

Computer Arkitektur og Operativ Systemer

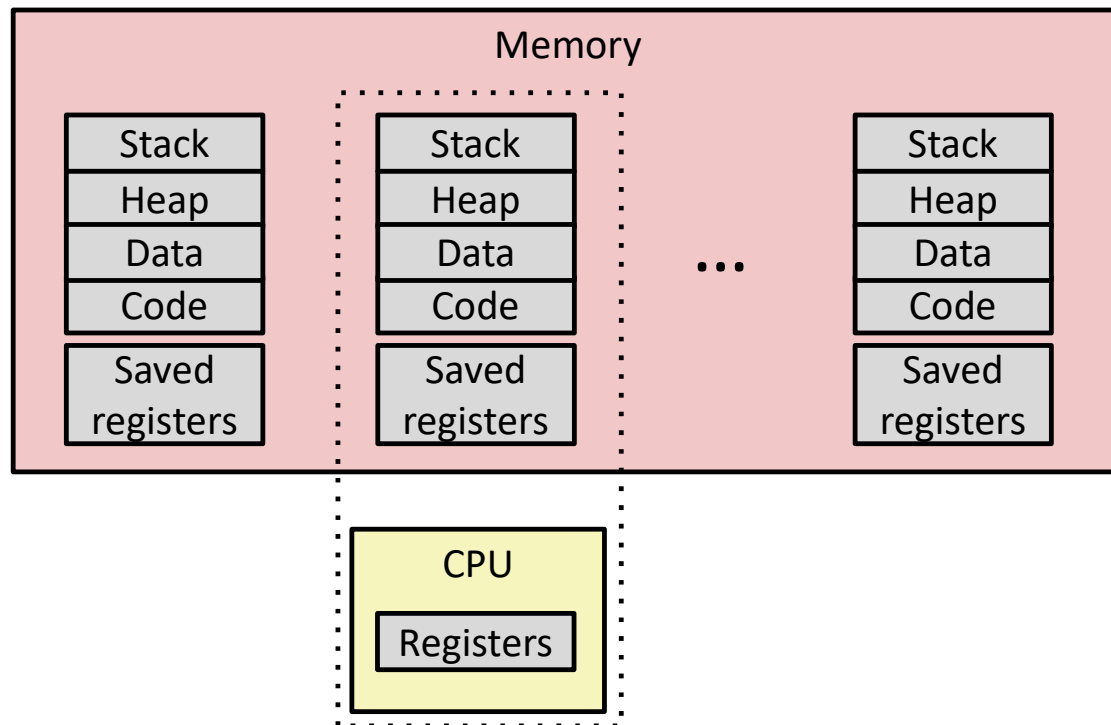
Concurrency 1

Forelæsning 11
Brian Nielsen

*Credits to
Randy Bryant & Dave O'Hallaron (CMU)
Youjip Won (KAIST)*

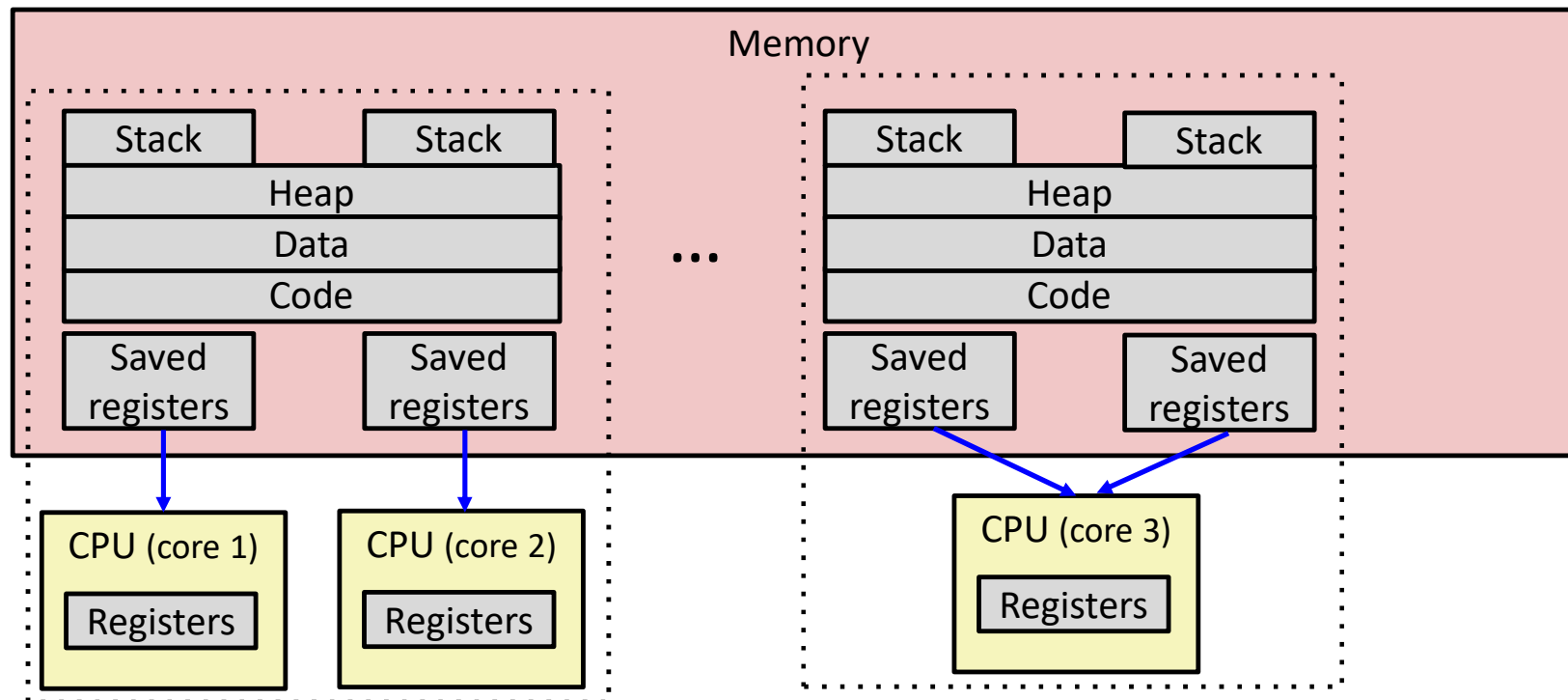
Parallelitet og Multi-threading

Den (traditionelle) realitet: Multi-programmering



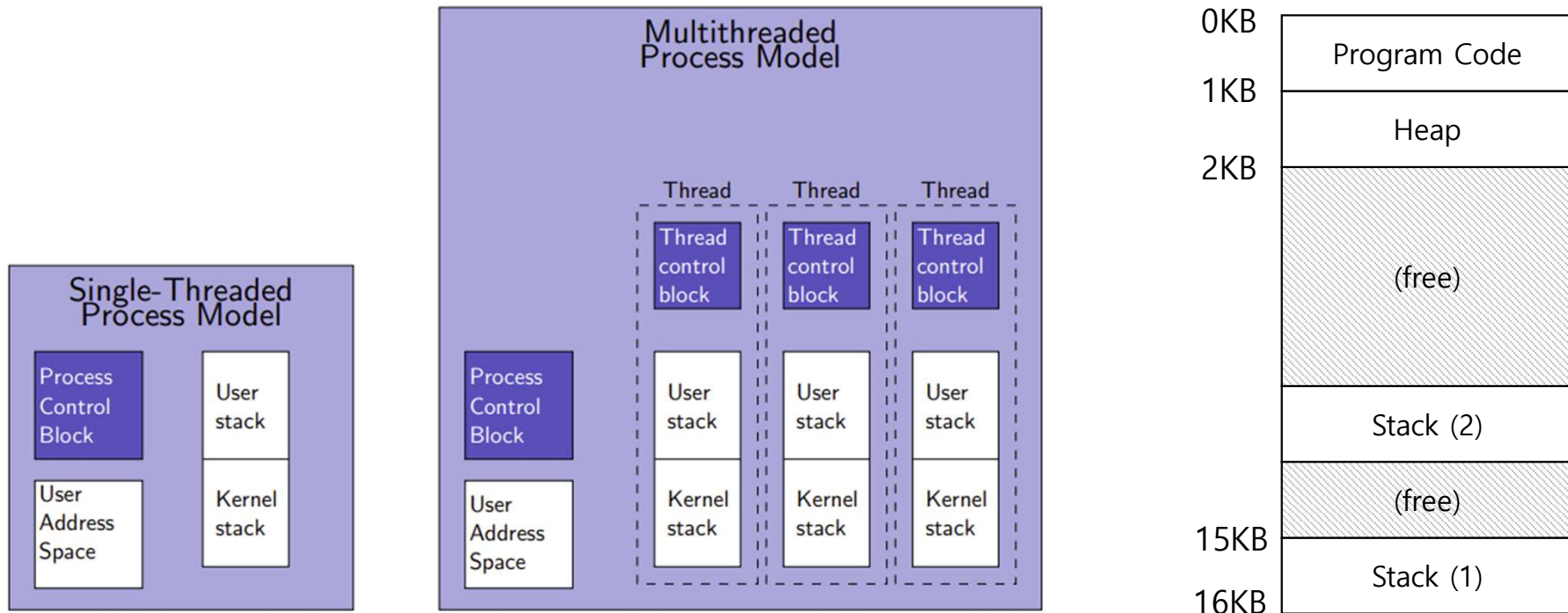
Et antal processer skiftes til at afvikle på CPUen

Multi-threading : Den (moderne) realitet 2



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

Fra enkelt-trådet proces til fler-trådet proces



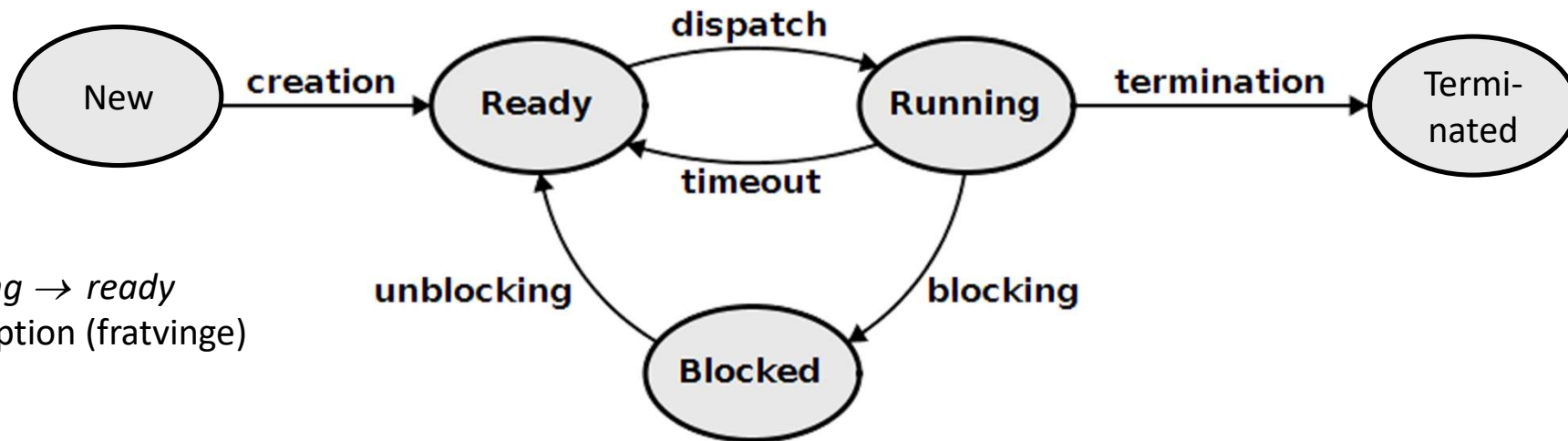
- Egen thread control block; egen stak
- Andre ressourcer i processen er DELT mellem processens tråde: øvrig hukommelse, åbne filer mv.
- Trådede er ikke isolerede: kan læse/(over) skrive hinandens data!

**(lille) Virtuelt Adresserum
m. 2 tråde**

Tråd kontrol blok: TCB

- **TCB: Thread-control-block** *Datastruktur som kernen vedligeholder pr. tråd for at styre dens afvikling*
- *En delmængde af PCB (alle processens ressourcer er jo delte mellem trådene og findes i PCB)*
 - Tråd identifikation
 - Unikt tråd ID, Process ID
 - Tråd tilstand
 - Gemt kontekst
 - Afviklingstilstand (i tilstandsdiagram)
 - Tråd kontrol
 - Prioritet, Stak,

Tilstandsmodel for 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)

Hvorfor fler-trådede programmer?

- **Parallelitet**

- Afvikle (lange) beregninger på flere CPU-kerner for at opnå en hastighedsgevinst
- **Parallelisering** : omskrivning af en sekventiel løsning/algorithm til en, der kan afvikles parallelt.

- **Øge hastighed / reaktions-evne** (responstid) af programmer

- I/O er laaaangsomt
- Tråde gør det muligt at **overlappe** I/O med beregninger i et enkelt program
- Iværksætte flere I/O operationer i parallelt på forskellige enheder: netværk og filsystem

- Også virkningsfuldt med een kerne

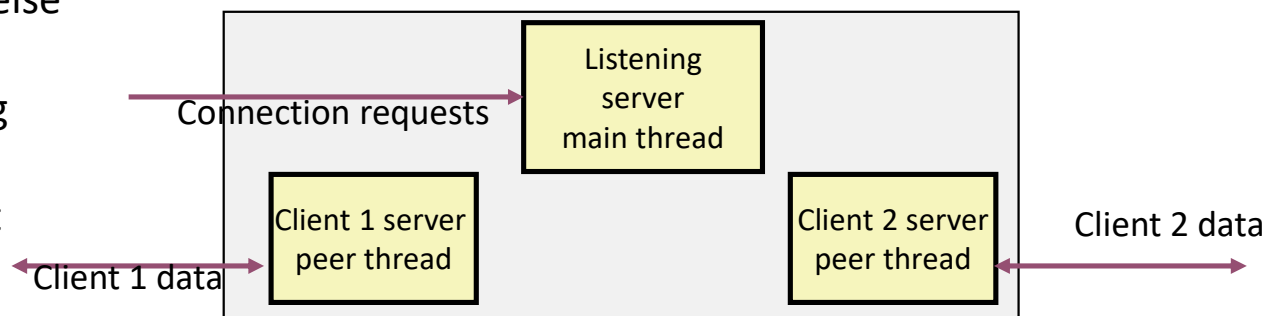
- Som **abstraktion** over aktivitet (som funktioner eller klasser) i simulation

Eksempler

- Web-server:

- En sekventiel web-server giver **dårlig svar tid** og **udnytter serveres ressourcer** dårligt

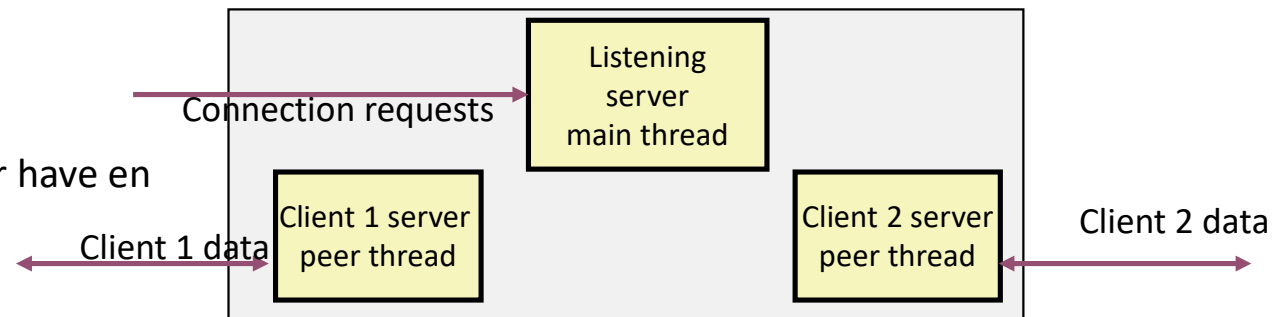
1. Accepter socket-forbindelse
2. Parse request-besked
3. Log request til access-log
4. Lav DB søgning
5. Lav beregning af resultat
6. Generér HTML/JSON
7. Send svar



- Fler-trådet server: Opret en tråd (eller have en pulje af tråde klar), som hver afvikler ovenstående)
- Billed-behandlingsprogram: tung beregning paralleliseres; en tråd styrer GUI
- Data-baser: opdatere (indexes) samtidigt med håndtering af forespørgsler
- Word: baggrundsopgaver som gemme til cloud og spell/syntax check
- Videnskabelige beregninger og simulationer: "talknusning"

Eksempler

- Web-server:
 - En **sekventiel** web-server giver **dårlig svar tid** og **udnytter serveres ressourcer** dårligt
 1. Acceptor socket-forbindelse
 2. Parse request-besked
 3. Log request til access-log
 4. Lav DB søgning
 5. Lav beregning af resultat
 6. Generér HTML/JSON
 7. Send svar
 - **Fler-trådet server**: Opret en tråd (eller have en pulje af tråde klar), som hver afvikler ovenstående)



- Billede-behandlingsprogram: tung beregning paralleliseres; en tråd styrer GUI
- Database: opdatere (indexes) samtidigt med håndtering af forespørgsler
- Word: baggrundsopgaver som gemme til cloud og stavekontrol/syntax check
- Videnskabelige beregninger og simulationer: "talknusing"

Hvorfor ikke parallelitet med processer?

- Processer er “tunge”;
 - Dyre at oprette
 - Dyrt proceskifte
- Svært bøvlet at dele data
 - Interprocess kommunikation
 - Delt-hukommelse kan sættes om
- Forkert abstraktion: program vs aktivitet

Grænser for speedup: Amdahl's Lov

Eksempel: 5 venner vil male en ny lejlighed med 5 rum



Speed-up: 5 personer vs. 1?
5 gange hurtigere end alene!

Hvad nu hvis et rum er dobbelt så stort som øvrige?



Antag: kun plads til en maler i et rum!

Så kan kun 5/6 enheder af arbejdet udføres i parallel (af 5 mand) ; rest sekventielt

Parallel afviklingstid: $5 \text{ enheder} / (5 \text{ mand} * 6 \text{ enheder}) + 1 \text{ enhed} / 6 \text{ enhed} * 1 \text{ mand}$
 $= 1/6 + 1/6 = 2/6 = 1/3$. **Kun 3 gange hurtiger!**

Speedup Limits: Amdahl's Law

- Indfange hvor svært der er at parallelisere en opgave
- $Speedup(n) = T_{sek} / T_{par}(n)$, n processorer

Amdahl's Law

$$S = \frac{1}{\underbrace{1-p}_{\text{sekventiel andel}} + \underbrace{p/n}_{\text{parallel andel}}}$$

sekventiel
andel

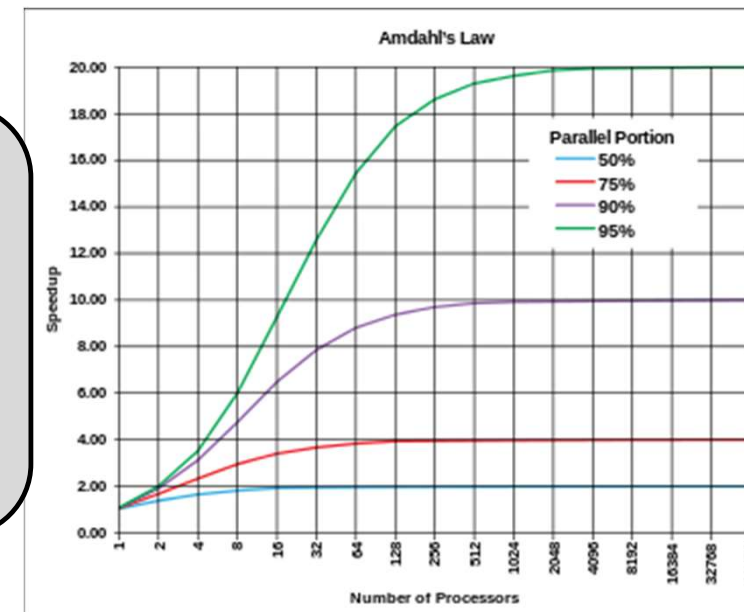
parallel andel

S = speedup

p = andel af arbejdet der kan
udføres parallelt ($0 \leq p \leq 1$)

n = antal processorer

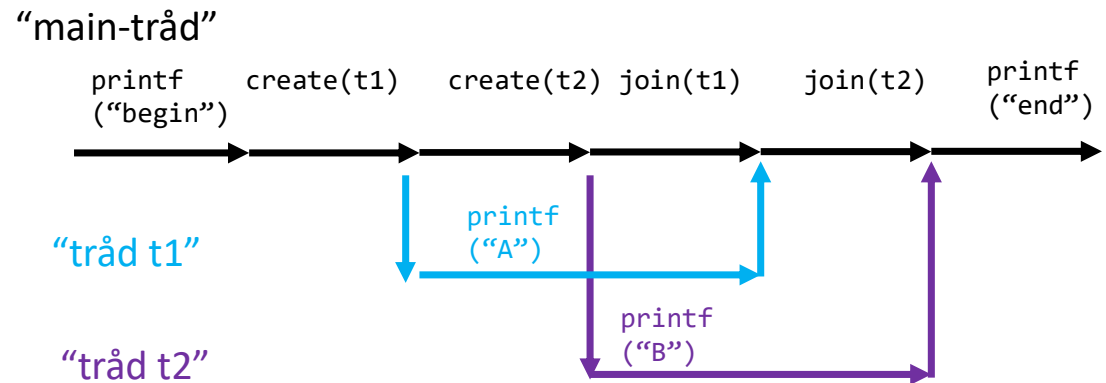
- Vigtigt at minimere den sekventielle andel!!
- God lineært speedup er svært at opnå
 - Afhængigheder mellem opgaver
 - Synkroniserings og kommunikations overhead
 - Kun få interessante "pinligt paralleliserbare" problemer



Tråde og Concurrency

Tråde og sideordnet afvikling (concurrency)

```
void* mythread (void *arg) {  
    printf ("%s\n", (char *) arg);  
    return NULL;  
}  
  
int main (int argc, char *argv[]) {  
    pthread_t t1, t2;  
  
    printf("main: begin\n");  
    Pthread_create(&t1, NULL, mythread, "A");  
    Pthread_create(&t2, NULL, mythread, "B");  
    // join waits for the threads to finish  
    Pthread_join(t1, NULL);  
    Pthread_join(t2, NULL);  
    printf("main: end\n");  
    return 0;  
}
```



Mulige printf-afviklingsrækkefølger

Begin	Begin
A	B
B	A
End	End

Hvilken er uforudsigelig; scheduler bestemmer!

Tråde og sideordnet afvikling (concurrency)

```
int counter=50;

void mythread(void) {
    counter++;
}

main() {
    create_thread(t1,mythread);
    create_thread(t2,mythread);
    join(t1); join(t2);
}
```

Forventer ved afslutning af main: Counter=52!

Kan faktisk ende med counter=51!



Tråde og sideordnet afvikling (concurrency)

counter++ oversættes til

```
tmp=counter;  
add 1, tmp;  
counter=tmp;
```

```
#counter ligger i 0x8049a1c  
105 movl (0x8049a1c), %eax  
108 addl $0x1, %eax  
113 movl %eax, (0x8049a1c)
```

```
int counter=50;  
  
void mythread(void) {  
    counter++;  
}  
  
main() {  
    create_thread(t1,mythread);  
    create_thread(t2,mythread);  
    join(t1); join(t2);  
}
```

Tråd1

```
tmp1=counter; # 50  
add 1, tmp1; # 51
```

tråd2

```
tmp2=counter; # 50  
add 1, tmp2; # 51  
counter=tmp2; # 51
```

```
counter=tmp1; # 51
```

RACE-CONDITION: "lost-update"

Interrupt/trådsifte



Tråde og sideordnet afvikling (concurrency)

```
#counter ligger i Mem[0x8049a1c]
105 movl (0x8049a1c), %eax
108 addl $0x1, %eax
113 movl %eax, (0x8049a1c)
```

OS	Thread1	Thread2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	mov (0x8049a1c), %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt	save T1's state				
	restore T2's state		100	0	50
		mov (0x8049a1c, %eax)	105	50	50
		add \$0x1, %eax	108	51	50
		mov (%eax, 0x8049a1c)	113	51	51
interrupt	save T2's state				
	restore T1's state		108	51	51
	mov %eax, 0x8049a1c		113	51	51

Eksempel

- Starter 2 tråde; tæller begge counter op med 1 ARG gange
- Forventet output: 2*ARG

```
>$ ./a.out 1000000
main: begin [counter = 0] [60208c]
A: begin [addr of i: 0x7f0ff3bfff3c]
B: begin [addr of i: 0x7f0ff33eff3c]
A: done
B: done
main: done
[counter: 1411341]
[should: 2000000]
```



```
...
int max;
volatile int counter = 0; // shared global variable

void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}

int main(int argc, char *argv[]) {
    ...
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
        (unsigned int) &counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    Pthread_join(p1, NULL); // join waits for the threads to finish
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n", counter, max*2);
    return 0;
}
```

<https://github.com/remzi-arpacidusseau/ostep-code/blob/master/threads-intro/t1.c>

Murphy og Concurrency

- Murphys 1. lov: kan det gå galt så går det galt
- Murphys 2. lov: det går galt på værst tænkelige tidspunkt (eksamensdemo)
- Tænk på scheduler som en ondsindet modstander der forsøger at afvikle dig program så det vil fejle!
- Fler-trådede programmer kan indeholde subtile fejle, som kun få afviklinger afslører
- “Concurrency bugs” / “Heisenbugs”
 - Kan ikke nødvendigvis gentages
 - Test besværligt/udueligt: hvordan kan vi afprøve alle “schedules”?????

Lidt terminologi

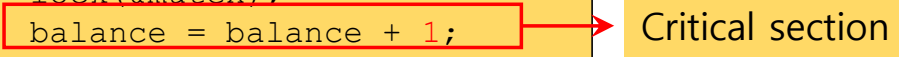
- **Race condition:**

- Resultatet afhænger af timing i afviklingen.
- Resultat er **ubestemt (indeterminate)**. Programmets udfald er **uforudsigelig** pga. ukendt **(non-deterministisk)** afviklingsrækkefølge.

- **Kritisk sektion (Critical section)**

- Et kode-fragment, der **tilgår delte variable (generelt ressourcer)** og som ikke må afvikles samtidigt (concurrently); dvs. må kun udføres af en tråd ad gangen
- Flere tråde i den kritiske region kan resultere i en race-condition
- Behov for at kritisk sektion kan afvikles **atomisk (udelt)**
- Behov for mekanismer til at lave gensidig udelukkelse (**mutual exclusion / mutex**)

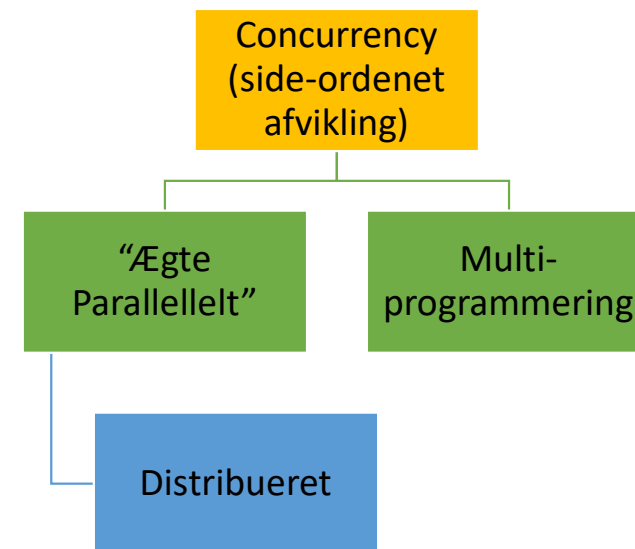
```
1  lock_t mutex;  
2  . . .  
3  lock(&mutex);  
4  balance = balance + 1;  
5  unlock(&mutex);
```



Critical section

Lidt terminologi

- **Concurrency**: afvikling sker samtidigt. Ved ikke om det (ønsker ikke at antage at) det er på en eller flere CPU-kerner.
- **(Ægte) Parallelt**: afvikling sker på flere kerner/processorer
- **Multi-programmeret**: samtidig afvikling på en kerne, fx vha timeslicing. Pseudo-parallelt.
- **Distribueret**: Beregning foregår på fysisk adskilte maskiner forbundet i netværk



Posix Threads API

Posix-threads

- En standardiseret definition af et API for programmering med tråde
- Implementationer på mange operativ systemer (inkl. diverse Unix-es, windows)
- `#include <pthread.h>`
 - `pthread_create`: opret en ny tråd
 - `pthread_join`: en tråd afventer terminering af en anden tråd
 - `pthread_exit`: selv-terminering (eller `return` fra tråd-funktionen)
 - `Pthread_self`: trådens eget trådID
 - `pthread_mutex_lock / unlock`: mutex
 - `pthread_condition_wait / signal`: synkronisering
- `ubuntu:~/ $ gcc -o myprog multithread.c -Wall -lpthread`
- `man -k pthread`

Oprettelse af tråd

```
int  
pthread_create(    pthread_t*    thread,  
                  const pthread_attr_t* attr,  
                  void*          (*start_routine)(void*),  
                  void*          arg);
```

- **thread**: Et håndtag brugt til styring af tråden.
- **attr**: Specificerer trådens attributer:
 - Stack size, Scheduling priority, ...
- **start_routine**: funktionen som tråden skal afvikle
 - Fx `void * mythread(void * arg) {...}`
- **arg**: argument til funktionen (`start routine`)
 - en void pointer tillader at en *vilkårlig* type kan overføres (casting).
- Pthread_create returnerer 0 hvis OK, ellers fejl-kode

Eksempel: pthread_create

- Definér struct-type til argumenter
- Tråd-funktionen
- Oprettelse af tråd
 - Thread handle **p**: reference parameter til pthread_create
 - Default attributter (**NULL**)
 - Funktionen **mythread**
 - Overførsel af argumenter

```
typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Afvent terminering af tråd

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- **thread**: Tråd handle, specificerer tråden, der skal afventes.
 - **Blokkerer** potentielt kalder.
 - Rydder op efter tråden
 - Terminerede tråde, der ikke afventes bliver “zombies” og optager ressourcer
- **value_ptr**: En pointer til returværdien
 - Da `pthread_join()` ændrer værdien, skal den have en pointer (reference parameter)
- “Man pages:”

There is no pthreads analog of `waitpid(-1, &status, 0)`, that is, “join with any terminated thread”. If you believe you need this functionality, you probably need to rethink your application design.

Terminering af tråd

- En tråd terminerer når tråd-funktionen
 - returnerer, eller
 - eksplicit kalder `pthread_exit()`;

```
void pthread_exit(void *retval);
```

```
void *mythread(void *arg) {  
    ...  
    return res;  
}
```

Eksempel: terminering

Korrekt

```
typedef struct __myarg_t {int a; int b;} myarg_t;
typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args; args.a = 10; args.b = 20;
    rc=pthread_create(&p, NULL, mythread, &args);
    rc=pthread_join(p, (void **) &m); //blocking
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Forkert

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

- Lokale variable allokeres på stak
- Samme levetid som funktions instansen
- Væk, når tråden terminerer
- Retur pointer udpeger garbage
- **Dualt** for argumenter ved **pthread_create!!!**
 - Hver tråd skal have sin egen-kopi af "args" som lever i trådens levetid!!!

Fejlhåndtering

- **Check altid returværdi for bibliotekskald og OS systemkald for fejl-retur-kode!**
 - Gælder derfor også fork(), exec(), open(),...
 - C sproget har ingen exception håndtering, så skal ske explicit med IF eller Assert
- Pthreads (med stort P) er bare bogens wrapper-makroer
 - Fejl-detektion, men ingen egentligt håndtering deraf!
 - Bedre end helt at lade være

```
void Pthread_xxx(...) {  
    int rc = pthread_xxx(...);  
    assert(rc == 0);  
}
```

```
#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include "common.h"  
#include "common_threads.h"  
  
int main(int argc, char *argv[]) {  
    ...  
    max = atoi(argv[1]);  
    Pthread_create(&p1, NULL, mythread, "A");  
    Pthread_create(&p2, NULL, mythread, "B");  
    Pthread_join(p1, NULL);  
    Pthread_join(p2, NULL);  
    return 0;  
}
```

<https://github.com/remzi-arpacidusseau/ostep-code/tree/master/include>

```
#define Pthread_create(thread, attr, start_routine, arg) assert(pthread_create(thread, attr, start_routine, arg) == 0);  
#define Pthread_join(thread, value_ptr) assert(pthread_join(thread, value_ptr) == 0);
```

Locks

- En abstract data-type, der har til formål at lave **mutex** (gensidig udelukkelse) til en **kritisk region**
 - Hvis låsen ikke er taget:
kalder-tråden tilegner sig låsen (acquire)
 - Hvis låsen holdes af anden tråd:
kalder-tråden blokkes indtil den låsen bliver frigivet, og tråden formår at tilegne sig låsen
- En lock-variabel skal erklæres og initialiseres!
- Alternativt: dynamisk initialisering

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock(&lock);  
x = x + 1; // or whatever your critical section is  
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_t lock;  
pthread_mutex_init(&lock, NULL);
```

Betingelsesvariable: Synkronisering af tråde

- En tråd skal afvente en hændelse i en anden tråd
- Condition variabel repræsenterer en venteliste
- `pthread_cond_wait`:
 - Sæt den kaldende tråd i blokkeret tilstand.
 - Afvent at en anden tråd signalerer at den kan fortsætte.
- `pthread_cond_signal`:
 - Væk (unblock) (mindst) en af de potentielt afventende tråde, som er blokkeret på angivne condition variabel
- Ex: tråd2 ønsker at signalere at en eller anden initialiserings-beregning er færdig:

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Tråd 1

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t init = PTHREAD_COND_INITIALIZER;  
int initialized=0;  
...  
pthread_mutex_lock(&lock);  
while (initialized == 0)  
    pthread_cond_wait(&init, &lock);  
pthread_mutex_unlock(&lock);
```

Tråd 2

```
pthread_mutex_lock(&lock);  
initialized = 1;  
pthread_cond_signal(&init);  
pthread_mutex_unlock(&lock);
```


Monitorer

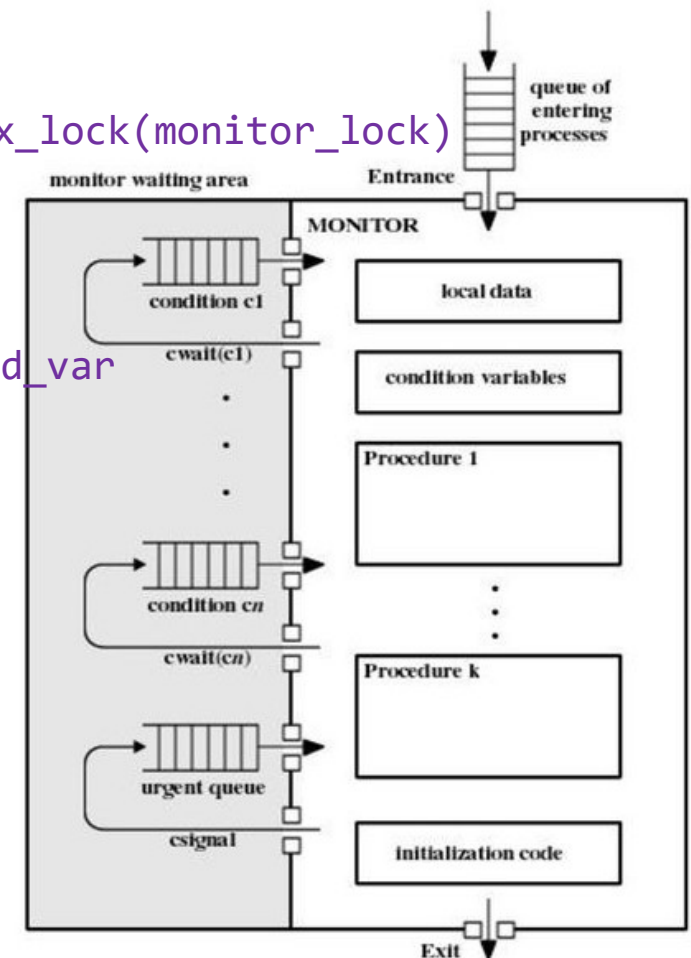
Hoare/Brinch Hansen 1973: Monitor = data abstraktion + atomicitet + synkronisering

- Samler data og operationer, der skal afvikles under mutex (kritiske regioner) i enhed
- Monitor lås:
 - Sikrer gensidig udelukkelse for tråde, der arbejder i monitor
 - Venteliste af tråde som ønsker adgang: (blokkeret på monitor lås)
- Condition variabel:
 - Repræsenterer en betingelse, der (potentielt) skal være opfyldt før en tråd kan fortsætte
 - Udtryk over variablene i monitoren
 - Holder en venteliste af blokerede tråde
- `cond_wait(cond_var, monitor_lock)`
 - Blokkérer kalder; som sættes i venteliste
 - Frigiver monitor lås
- `cond_signal(cond_var)`
 - Vækker en (eller flere) fra ventelisten
 - Bør kun kaldes mens tråden har monitor låsen
 - Nogle implementationer har "spurious wakeup"
 - Den vækkede tråd skal tilegne monitor låsen inden den kan afvikle
- Scheduler afgør hvilke af ventende tråde (ved indgang + vækkede fra condition ventelister) er skal køre næst
 - Uforudsigeligt hvilken tråd kører næst!
 - Pak evaluering af udtrykket in i en while!!!

`mutex_lock(monitor_lock)`

`cond_var`

`mutex_unlock(monitor_lock)`



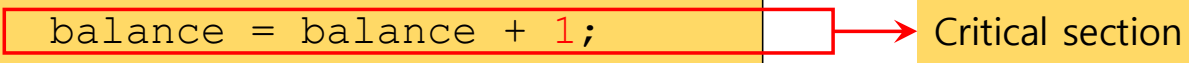
Hvordan laver vi gensidig-udelukkelse ?

- en “mutex” lock

Mutex-låse (locks)

- Mål: Garantere at en **kritisk region** udføres som om den var en enkelt atomisk (udelelig) enhed
 - Generisk eksempel: opdatering af en delt variable: `balance=balance+1;`

```
1  lock_t mutex;  
2  . . .  
3  lock(&mutex);  
4  balance = balance + 1;  
5  unlock(&mutex);
```



- Idé: lock-variabel holder står på låsens tilstand
 - **Ulåst/Ledig/fri**: ingen tråd holder låsen
 - **Låst/holdt/erhvervet**: en tråd ejer låsen og afvikler (potentielt) i kritisk region
 - Hvis en tråd forsøger at tage en lås som holdes (af en anden) så tvinges tråden til at afvente frigivelse!
- Hvordan laver vi "lock()" og "unlock()"
 - Konstruktion af en effektiv lås kræver hjælp fra både **hardware** og **OS**.

Ønsker til mutex locks

- **Sikre gensidig udelukkelse**

- Forhindrer mekanismen at flere tråde kan træde in i kritisk region?
- "Safety-krav"

- **Fairness**

- Når flere tråde konkurrerer om adgang, har alle så "lige" mulighed for at få låsen?
- Er der risiko for udsultning (Starvation)?
- Et "liveness krav"

- **Performance**

- Hvilket tids-overhead tilføjer låse programmet?
- Hvor mange "spildte" clock-cykler

Forsøg 1 : FLAG-metoden

- Brug et boolsk *flag* til at angive om låsen er holdt eller fri?

```
bool flag=0;

//lock
while (flag==1) ; //afvent (udfører tomt stmt)
    flag=1;

CRITICAL SECTION

//unlock
flag=0;
```

Forsøg 1: FLAG metoden

```
bool flag=0;

Tråd1                                Tråd2
//lock
while (flag==1) ;                    while (flag==1)
                                     flag=1;
                                     CRITICAL SECTION

                                     flag=0;

flag=1;
CRITICAL SECTION

flag=0;
```

Interrupt/trådsifte



1. Naiv brug af flag duer ikke: Garanterer ikke mutex!
2. Spin-wait (Busy-waiting): spilder mange clock-cycler.

Førsøg 2: styring af interrupts

- Idé: timer-interrupts forårsager trådskefe
 - **Slå Interrupts fra** når vi træder ind i CS
 - En af de første løsninger til mutex (opfundet for enkelt-processor systemer).

```
1  void lock() {  
2      DisableInterrupts();  
3  }  
4  void unlock() {  
5      EnableInterrupts();  
6  }
```

- Problemer:
 - *Kræver at vi stoler for meget på applikationer*
 - Grådigt/ondsindet program kan jo "glemme" at slå interrupts til igen og dermed eje al processortid.
 - Virker ikke på **multiprocessors maskiner**
 - Lange kritiske regioner vil give dårlig svartid til ydre enheder
 - Instruktioner til maskering/afmaskering af interrupts er langsomme på moderne CPUs.

Forsøg 3: TestAndSet

- Udvid processor med særlig TestAndSet (exchange/swap) instruktion
 - **Returner** (testing) gammel værdi af variable (`ptr`).
 - **Samtidigt updater** (skriver) variabel til `new`.
 - Sekvensen udføres **atomisk af HW**.

```
//pseudo-kode som beskriver adfærden af TestAndSet INSTRUKTION
//HW garanterer den udføres atomisk
1  int TestAndSet(int *ptr, int new) {
2  int old = *ptr;    // fetch old value at ptr
3  *ptr = new;        // store 'new' into ptr
4  return old;        // return the old value
5  }
```

```
bool flag=0;

//lock
while (TestAndSet(&flag,1)==1) ; //afvent (udfører tomt stmt)

CRITICAL SECTION

//unlock
flag=0;
```


Forsøg 3: TestAndSet

```
bool flag=0;

Tråd1                                Tråd2

//lock                                while (
while (                                TestAndSet(&flag,1)==1
                                        // fik retur 0
                                        ) ;

    TestAndSet(&flag,1)==1            CRITICAL SECTION
    //Øv, fik retur 1                 flag=0;
    ) ;

    //fik retur 0                     //Færdig: ude af CS
    CRITICAL SECTION
```

- Garanterer mutex: kun en tråd vinder ræset om at få flaget
- Spilder potentielt helt time-slice på spin-wait

Forsøg 4: Reduktion af busy-wait

- Spinning (busy-wait) kan reduceres ved frivilligt afgive CPU
- System kald til at "vige" `yield()`
 - OS flytter kalder fra *running tilstand* til *ready tilstand*.
 - Vi betaler stadig dyr pris for **context switch**
 - Mulighed for **udsultning** findes stadig (fx scheduler kan genaktivere den tråd, som netop kaldte `yield`)

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t * mutex) {
4      mutex->flag = 0;
5  }
6
7  void lock(lock_t * mutex) {
8      while (TestAndSet(&mutex->flag, 1) == 1)
9          yield(); // give up the CPU
10 }
11
12 void unlock() {
13     mutex->flag = 0;
14 }
```

Forsøg 5: Blokker tråd i kø

- Kan vi ikke sætte tråd i "venteliste" og bede OS om at blokkere tråden?
 - `park()` og `unpark(thread_id t)` systemkald
- Men nu bliver køen selv delte variable, og skal beskyttes af mutex => guard variabel fungerer nu som lås for mutex-data-strukturen! !

```
1  typedef struct __lock_t {
2  int flag; // lås er holdt eller fri
3  int guard; // beskyttelse af køen
4  queue_t *q; } lock_t;
5
6  void lock_init(lock_t *m) {
7      m->flag = 0;
8      m->guard = 0;
9      queue_init(m->q);
10 }
11
12 void lock(lock_t *m) {
13     while (TestAndSet(&m->guard, 1) == 1)
14         ; // acquire guard lock by spinning
15     if (m->flag == 0) {
16         m->flag = 1; // lock is acquired
17         m->guard = 0;
18     } else {
19         queue_add(m->q, gettid());
20         m->guard = 0;
21         park();
22     }
23 }
24 ...
```

```
22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
24         ; // acquire guard lock by spinning
25     if (queue_empty(m->q))
26         m->flag = 0; // let go of lock; no one wants it
27     else
28         unpark(queue_remove(m->q)); // hold lock (for next thread!)
29     m->guard = 0;
30 }
```

Forsøg 6: Blokker tråd



- Kode har race-condition!!!

```
1  Tråd 1:                                Tråd 2:
2  lock(lock_t *m) {
3      while (TestAndSet(&m->guard, 1) == 1)
4          ; // acquire guard lock by spinning
5      if (m->flag == 0) {
6          m->flag = 1; // lock is acquired
7          m->guard = 0;
8      } else {
9          queue_add(m->q, gettid());
10         m->guard = 0;
11
12         park();
13     }
14 }
15 ...
```

//lås frigivet
//preemption!
//tråd 2 laver unlock her!
//tråden parkeres alligevel og venter evigt!!

- **Løsning:** Lidt anderledes park() og unpark() systemkald
 - Muliggør at frigive lås og parkere tråd atomisk
- Solaris OS: "setpark"
- Linux OS: "futexes"



Lessons learned

- Det er tricky at “bygge” en mutex
 - Der er korrekt
 - Gerne fair (eller garanterer fremskridt)
 - Effektiv
- Vi kan ikke bygge mutex ud af “ingenting”; skal arve support fra HW
 - Atomisk testAndSet, disable interrupts
 - (Petersons algoritme løser problemet vha. delte variable, med busy-wait og visse antagelser om “memory coherency” i fler-kerne systemer)
- Brug de mutex mekanismer, der er given med bibliotek/sprog
 - Lav ikke egne fra bunden vha. flags
- Vi skal se at vi nogle gange kan/vil bygge andre mutex primitiver oven på de givne

Data-strukturer med concurrency

Problemstilling

- Abstrakt data-type (objekt)
- En samling data
- Opereres på vha et antal veldefinerede procedurer (metoder)
- Men kan du arbejdes på samtidigt (concurrently) af flerer tråde
- Vi skal sikre mutex vha. "locks"



```
struct my-data {  
    ...  
}  
function1 (struct my-data *d);  
function2 (struct my-data *d);
```

Eksempel 1: En simple optæller

```
typedef struct __counter_t {
    int value;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
}

void increment(counter_t *c) {
    c->value++;
}

int get(counter_t *c) {
    return c->value;
}
```

- En enkelt global lås tilføjes data-strukturen
- Låses og frigives i hver kritisk region: her operation



```
typedef struct __counter_t {
    int value;
    pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}

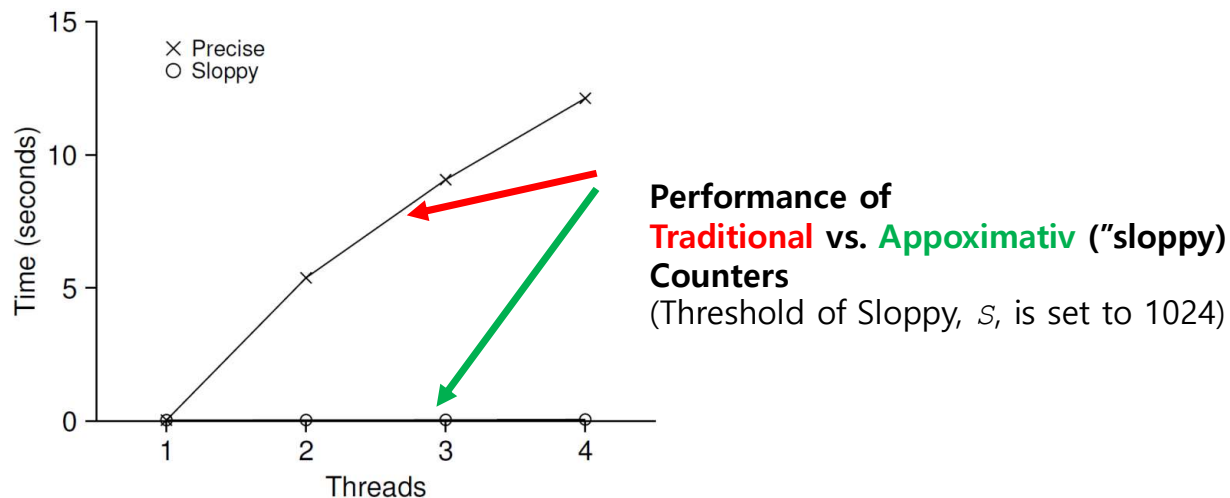
void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c->value;
    Pthread_mutex_unlock(&c->lock);
    return rc;
}
```

- Fx event-optælling

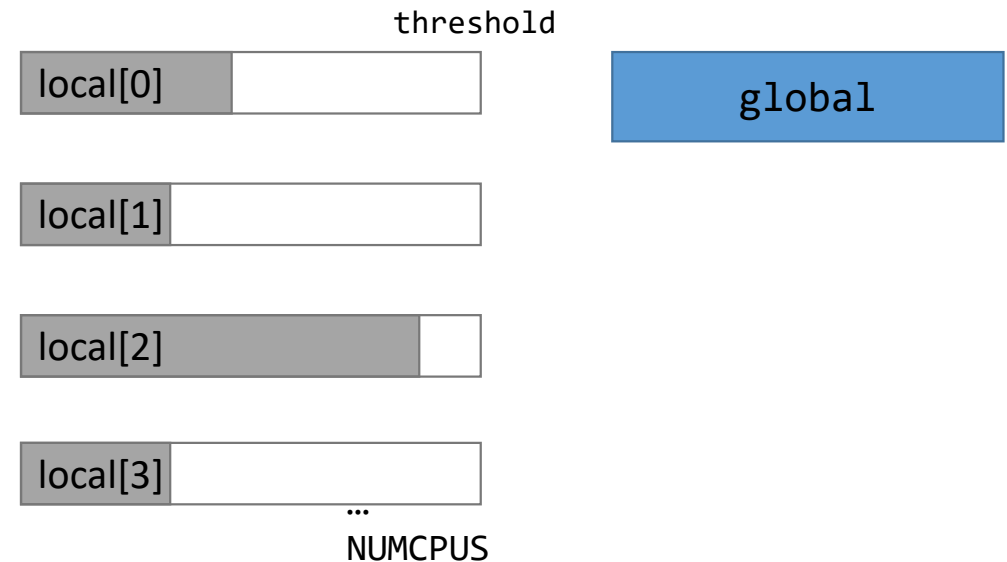
Eksempel 1: Problemstillinger

- Alle tråde deler enkelt global lås:
 - Ingen mulighed for at operere parallel
 - Skalerer dårligt
- Hver tråd opdaterer optæller 1M gange (Intel 2.7GHz i5 CPU).



Eksempel 2: En approximative tæller

- Lav en lokal tæller svarende til antal CPU/kerner NUMCPUS
- En "worker" tråd per kerne
- Når en tråd har talt sin local op til en given tærskel, akkumuleres det lokale bidrag i global tæller
- Optælling lokalt kan ske uden adgang til global mutex
- Bemærk:
 - NUMCPUS lokale mutex locks
 - 1 global



```
typedef struct __counter_t {  
    int global;           // global count  
    pthread_mutex_t glock; // global lock  
    int local[NUMCPUS];   // local count (per cpu)  
    pthread_mutex_t llock[NUMCPUS]; // ... and locks  
    int threshold;        // update frequency  
} counter_t;
```

Eksempel 2: En approximative tæller

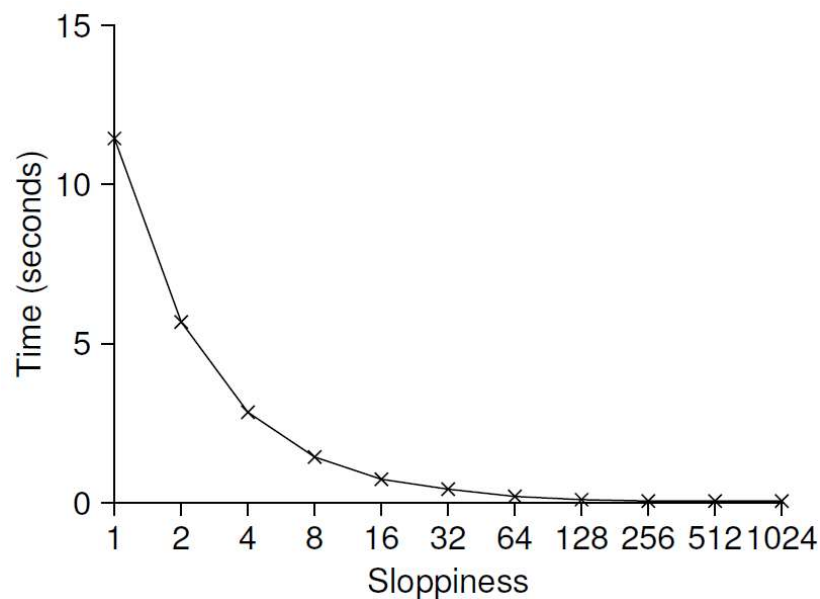
- Increment tager lokal lås
 - Bruger kun global (glock) lås når den har oversteget tærsklen
- NB Hvis antal tråde >> antal CPU
 - Tråde fordeles ligeligt ud på lokale tællere (threadID%NUMCPU)

```
// update: usually, just grab local lock and update local amount
//           once local count has risen by 'threshold', grab global
//           lock and transfer local values to it
void increment(counter_t *c, int threadID) {
    pthread_mutex_lock(&c->llock[threadID]);
    c->local[threadID] += 1;
    if (c->local[threadID] >= c->threshold) { // transfer to global
        pthread_mutex_lock(&c->glock);
        c->global += c->local[threadID];
        pthread_mutex_unlock(&c->glock);
        c->local[threadID] = 0;
    }
    pthread_mutex_unlock(&c->llock[threadID]);
}

// get: just return global amount (which may not be perfect)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
```

Eksempel 2 En approximative tæller

- Valg af tærskelværdi
 - Lav tærskel: dårlig performance, men global count er præcist
 - Høj tærskel: god performance, men stort efterslæb på count

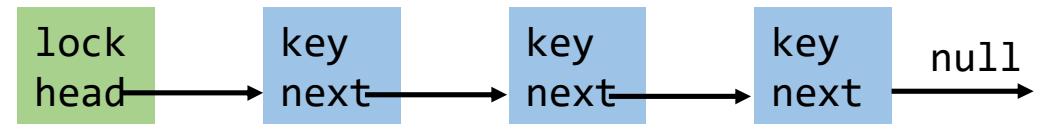


Simple Kædet liste (generelt træer mv)

```
// basic node structure
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) { ... }
int List_Insert(list_t *L, int key) { ... }
int List_Lookup(list_t *L, int key) { ... }
```

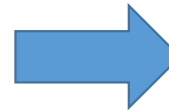


- Pak kritiske regioner i insert/lookup ind i lock()/unlock() på “head”-lås

Kædet liste (generelt træer mv)

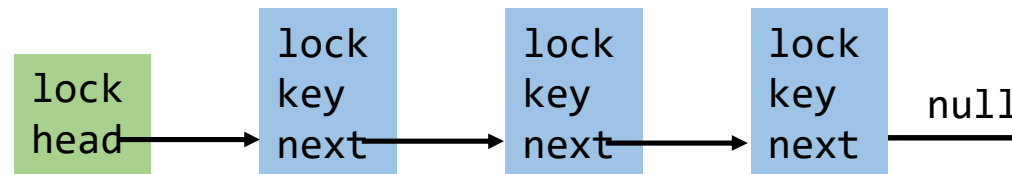
- NB1: Fejl i kritiske regioner: Husker du unlock i alle situationer
 - Omskriv kode til enkelt (eller så få) exit punkter som muligt
- NB2: lav kritisk region så kort som mulig

```
int List_Insert(list_t *L, int key) {  
    pthread_mutex_lock(&L->lock);  
    node_t *new = malloc(sizeof(node_t));  
    if (new == NULL) {  
        perror("malloc");  
        pthread_mutex_unlock(&L->lock);  
        return -1; // fail  
    }  
    new->key = key;  
    new->next = L->head;  
    L->head = new;  
    pthread_mutex_unlock(&L->lock);  
    return 0; // success  
}
```



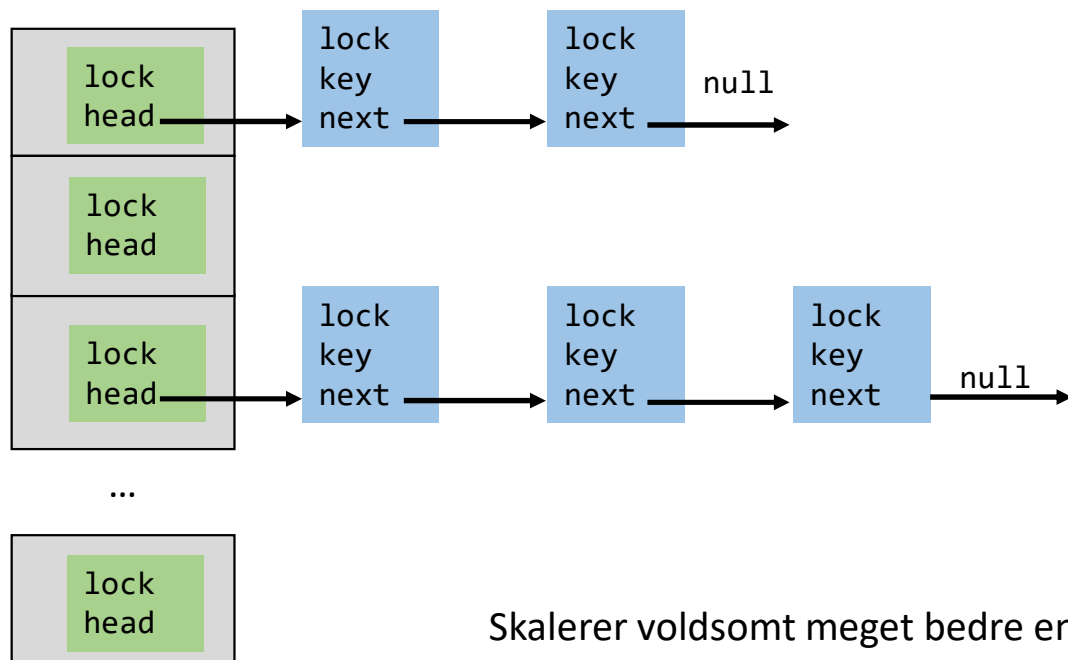
```
void List_Insert(list_t *L, int key) {  
    // synchronization not needed  
    node_t *new = malloc(sizeof(node_t));  
    if (new == NULL) {  
        perror("malloc");  
        return;  
    }  
    new->key = key;  
    // just lock critical section  
    pthread_mutex_lock(&L->lock);  
    new->next = L->head;  
    L->head = new;  
    pthread_mutex_unlock(&L->lock);  
}
```

- NB3: Øg paralleliteten ved at lave en lås pr. liste-element og udfør "hand-over-locking" (tag næste; frigiv forrige)



Concurrent Hash-tabel

- Statisk størrelse
- Kollisioner håndteres eksternt i (concurrent) kædet liste
- En lås per "bucket"



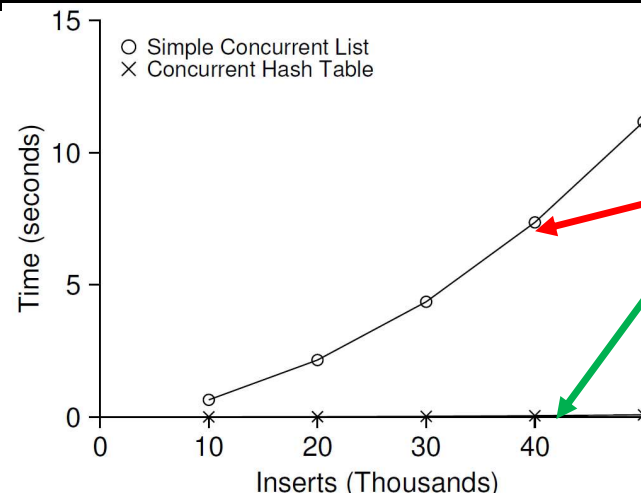
Skalerer voldsomt meget bedre end simpel (enkelt-lock concurrent) liste

```
#define BUCKETS (101)
typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);
    }
}

int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Lookup(&H->lists[bucket], key);
}
```



Låsings-strategi

- **Grov-kornet låsingsstrategi:** Lås en stor portion af en data-struktur af gangen og udfør mange operationer
 - Typisk mindre parallelitet mulig , men
 - Mindre overhead ved at låse/frigive
- **Fin-kornet-låsingsstrategi:** Lås så lidt som muligt og udfør få operationer af gangen
 - Typisk: meget parallelitet muligt
 - Større overhead ved at låse frigive
- En blanding af erfaring og eksperimenter afgør hvad der fungerer bedst i en given applikation.
 - Men pas på ikke at lave det for fin-kornet
 - Men enkelt global lås skalerer dårligt

