Blokkerende systemkall

- Viktigste grunn for tråder: blokkerende I/O forespørsler
- Applikasjonen som ber om I/O blir satt på vent av operativsystemet til resultatet fra I/O returnerer
- Programmet kan da ikke kjøre videre før det får resultatet
- Generelt leder forespørsler om I/O til systemkall

Eksempler på blokkerende systemkall:

- read/write
- wait
- sleep

Eksempler på ikke-blokkerende systemkall:

- getpid
- gettimeofday
- setuid

Thread-modeller

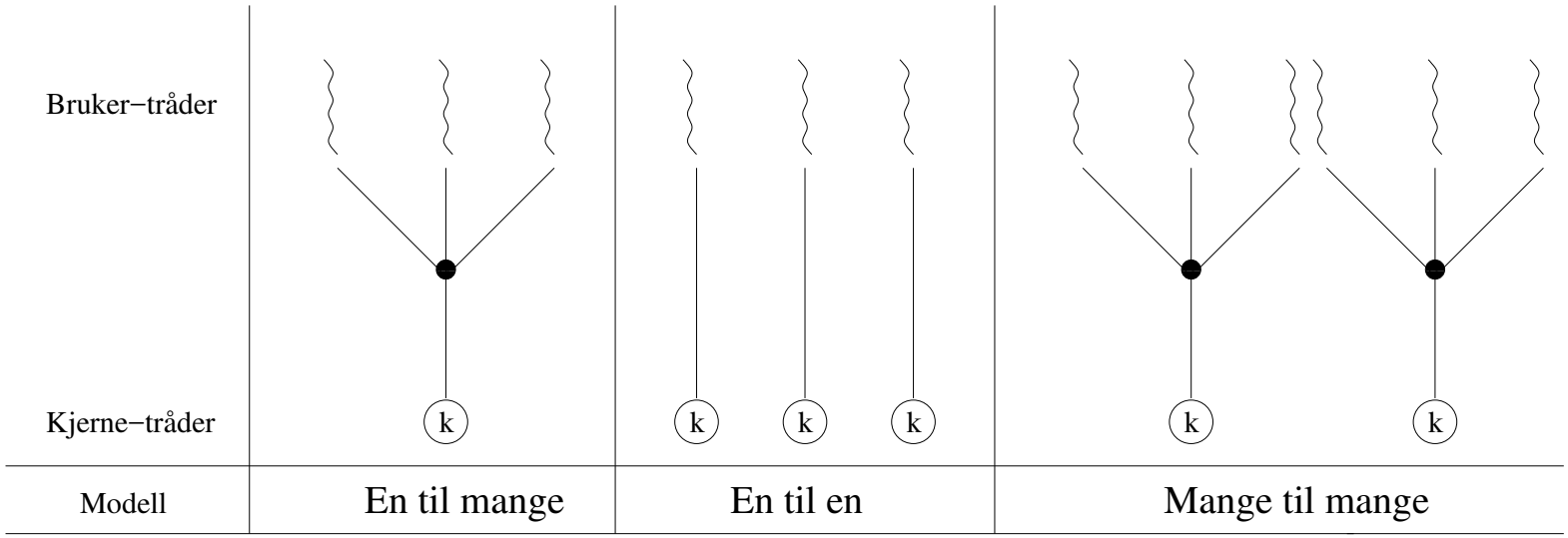

Figure: Thread-modeller i OS-kjernen

Thread-modeller

en til mange Alle trådene schedules som en prosess, en enhet. Java: green-threads, JVM sørger selv for scheduling; ingen multitasking. Default på gamle versjoner av Linux(Debian) og Solaris.

en til en Den mest vanlige. Hver tråd schedules uavhengig av de andre. Windows Java-threads, Linux native Java-threads, Linux Posix-threads (pthreads)

mange til mange Tråder schedules uavhengig om de ikke er for mange. Kjernen kan begrense antall tråder i RR-køen. Solaris, Digital Unix, IRIX pthreads

Synkronisering

Samtidige prosesser som deler felles ressurser/data må **synkroniseres**.

- prosesser må ikke endre felles data samtidig
- en prosess bør ikke lese felles data mens en annen endrer dem
- en prosess må kunne vente på (f. eks. resultater fra) en annen prosess

Serialisering

- Prosesser/tråder som aksesserer felles data må serialiseres
- De må jobbe en av gangen på felles data.
- Problemstillingen kalles **Race Condition** (konkurranse om felles ressurser).
- Programmereren må selv serialisere sine prosesser. OS legger mulighetene til rette.

Eksempel: To web-prosesser som skriver ut billetter

- På en web-side kan man kjøpe billetter
- En uavhengig prosess starter på web-server for hver bruker som bestiller en billett
- Disse prosessene er helt uavhengige; kan kjøre på hver sin CPU
- Men har felles variabel `LedigeBilletter` som er antall ledige billetter

```
if(LedigeBilletter > 0){
   LedigeBilletter--;
   SkrivUtBillett();
}
```

Mulig problem

P1-kode	P2-kode	LedigeBilletter
if(LedigeBilletter $>$ 0){		1
Context Switch$\Longrightarrow$	if(LedigeBilletter $>$ 0){	1
	LedigeBilletter- -;	0
	SkrivUtBillett();	0
	}	0
LedigeBilletter- -;	$\Longleftarrow$Context Switch	-1
SkrivUtBillett();		-1
}		-1

En Context Switch kan forekomme når som helst. Må serialiseres!

Race condition med èn kodelinje

- Èn linje høynivåkode oversettes ofte til mange linjer maskinkode av kompilatoren!
- En Context Switch kan oppstå mellom to hvilke som helst maskininstruksjoner.
- Dermed kan en Race Condition oppstå selv inni èn kodelinje
- CPU'en ser kun maskininstruksjoner og aner ikke noe om høynivåkoden som ligger bak.

Eksempel: to prosesser som oppdaterer en felles variabel

Anta at to prosesser P1 og P2 kjører følgende høynivåkode som oppdaterer en konto:

P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
.	.
.	.
saldo = saldo - mill;	saldo = saldo + mill;

Variabelen `saldo` er da en felles variabel begge kan endre.

Problem Hva skjer om OS switcher fra P1 til P2 mens P1 ufører `saldo = saldo - mill` ?

Hvorfor? Prosessen utfører maskinkode, linje for linje, og kan bli avbrutt etter en instruksjon.

Maskinkode for èn kodelinje

P1	P2
saldo = saldo - mill;	saldo = saldo + mill;

mov saldo, %ax
mov mill, %bx
sub %bx, %ax
mov %ax, saldo
	mov saldo, %ax
	mov mill, %bx
	add %bx, %ax
	mov %ax, saldo

En million forsvinner!

Prosess som kjører	Instruksjon (IR)	%ax	%bx	saldo
P1	mov saldo, %ax	5	0	5
P1	mov mill, %bx	5	1	5
OS	Context switch	0	0	5
P2	mov saldo, %ax	5	0	5
P2	mov mill, %bx	5	1	5
P2	add %bx, %ax	6	1	5
P2	mov %ax, saldo	6	1	6
OS	Context switch	5	1	6
P1	sub %bx, %ax	4	1	6
P1	mov %ax, saldo	4	1	4

Det burde ha blitt saldo $= 5$ og en mill er borte!! Konklusjon: må serialisere aksess til felles data!

Kritisk avsnitt

To prosesser P1 og P2 kjører:

P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
.	.
.	.
saldo = saldo - mill;	saldo = saldo + mill;

- Utregningen av saldo er et **kritisk avsnitt** i koden til P1 og P2.
- Kritisk avsnitt **må fullføres** av prosessen som utfører det uten at andre prosesser slipper til.
- Det medfører at prosessene må serialiseres.

Mulige måter å takle kritiske avsnitt

A Skru av scheduler før kritisk avsnitt. P1 kode:

```
disableInterupts();
saldo = saldo - mill;
enableInterupts();
```

OK for en OS-kjerne, men for farlig for brukerprosesser; de kan ta over styringen.

B Bruke en form for lås som gjør at bare en prosess av gangen har tilgang til felles data.

- MUTual EXclusion = MUTEX = gjensidig utelukkelse
- mest brukt
- mange implementasjoner

Linux-eksempel

File-lock for Linux-mail: hvis filen

`/var/mail/haugerud.lock`

eksisterer, kan inbox ikke leses/skrives til.

Sendmail og andre mailprogram lager denne filen før de skriver/leser mail og fjerner den når de er ferdige.

Windows-eksempel

Win 32 API'et har to funksjonskall

- EnterCriticalSection
- LeaveCriticalSection

som applikasjoner kan kalle før og etter et kritisk avsnitt.

Softwareløsning for P1/P2 med MUTEX

- Trenger to funksjoner `GetMutex(lock)` og `ReleaseMutex(lock)`
- Gjør at en prosess av gangen kan sette en lock.
- Gir følgende løsning:

```
GetMutex(lock);       // henter nøkkel
KritiskAvsnitt();     // saldo -= mill;
ReleaseMutex(lock);   // gir fra seg nøkkel
```

Software-mutex, forsøk 1

```
static boolean lock = false; // felles variabel

GetMutex(lock)
   {
   while(lock){} // venter til lock blir false
   lock = true;
   }
ReleaseMutex(lock)
   {
   lock = false;
   }
```

Dette burde sikre at to prosesser ikke er i kritisk avsnitt samtidig?