

Computer Arkitektur og Operativ Systemer

Concurrency 2

Forelæsning 12
Brian Nielsen

Credits to

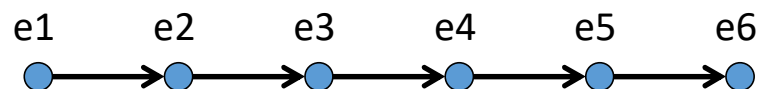
Randy Bryant & Dave O'Hallaron (CMU)

Youjip Won (KAIST)

Synkronisering med condition variable

Logisk tid

- En sekventiel) tråd består af en sekvens af hændelser
 - Hændelse fx udførelse af en maskinordre / instruction
 - Låsning af lock



- e1 sker-før e2 sker-før e3 osv (program rækkefølge)
 - En logisk tidsordning
- Lad T være et globalt ur; $T(e)$ er det reelle tidspunkt hvor hændelsen e sker.
 - Da har vi $T(e2)-T(e1)$ er uforudsigeligt!!

Synkronisering

```
int initialized=0;
```

- *En tråd skal afvente at en anden tråd når til et givent punkt*

- Initialisering færdig
- Mellemresultat er klart
- Afvent terminering (join)

- Fx. Tråd **p** skal afvente at tråd **q** er færdig initialiseret

q1 sker-før q2 sker-før q3

p1 sker-før p2 sker-før p3

(korrekt) Synkronisering sikrer at

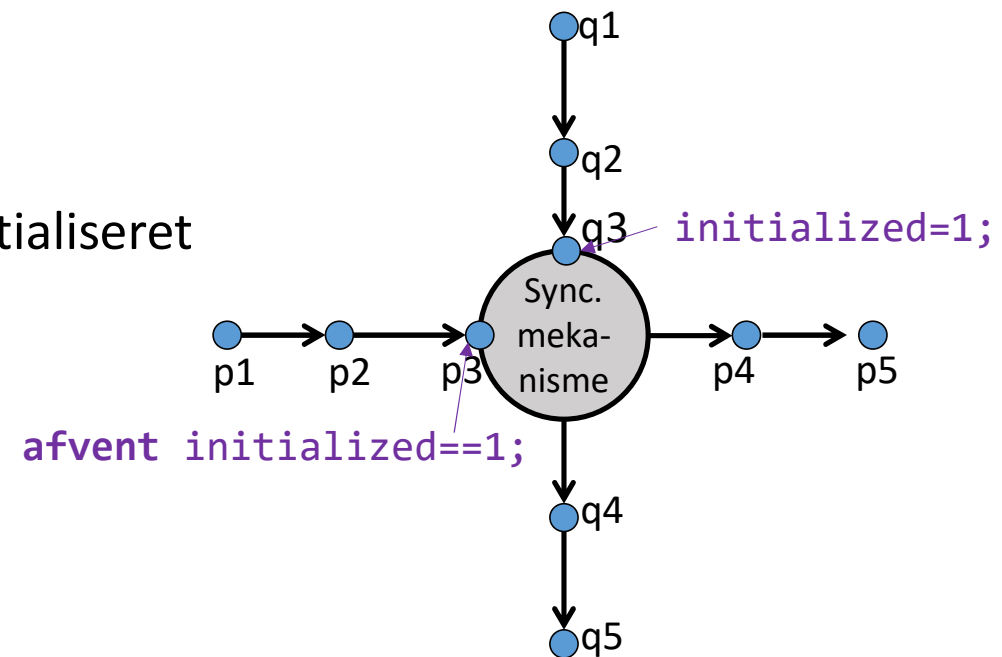
- q3 sker-før p4

(som følge q3 sker-før p5)

Giver en logisk tidsordning mellem forskellige tråde.

Hvis vi ikke kan sige om den ene hændelser sker-før den anden "ved at følge pilene" er de **concurrent**:

- q2 concurrent med p2
- p5 concurrent med p5



Synkronisering med betingelsesvariable

- En tråd skal afvente en hændelse i en anden tråd
- Condition variabel repræsenterer en venteliste
- `pthread_cond_wait`:
 - Sæt den kaldende tråd i blokeret tilstand.
 - Afvent at en anden tråd signalerer at den kan fortsætte.
 - Frigiver mutex
- `pthread_cond_signal`:
 - Væk (unblock) (mindst) en af de potentielt afventende tråde, som er blokeret 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);
```

Synkronisering med betingelsesvariable

- Hvad skal vi med "initialized" tilstandsvariabel ?
- Problematisk afviklingssekvens
 1. Tråd2 afvikler
 2. Signalerer conditon variabel
 3. Men ingen venter
 4. Tråd 1 afvikler
 5. Afventer på condition variabel
 6. Kommer aldrig videre
- En tråd ved ikke om den skal vente eller ej

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);
```

Synkronisering med betingelsesvariable

- Hvad skal vi med mutexen?
- Problematisk afviklingssekvens
 1. Tråd1 afvikler
 2. Tråd1 tester på initialized==0
 3. Kontext skift til tråd 2
 4. Signalerer afventende (pt ingen), så signalet "tabes"
 5. Tråd 1 fortsætter; blokerer på condition
 6. Kommer aldrig videre :-)

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

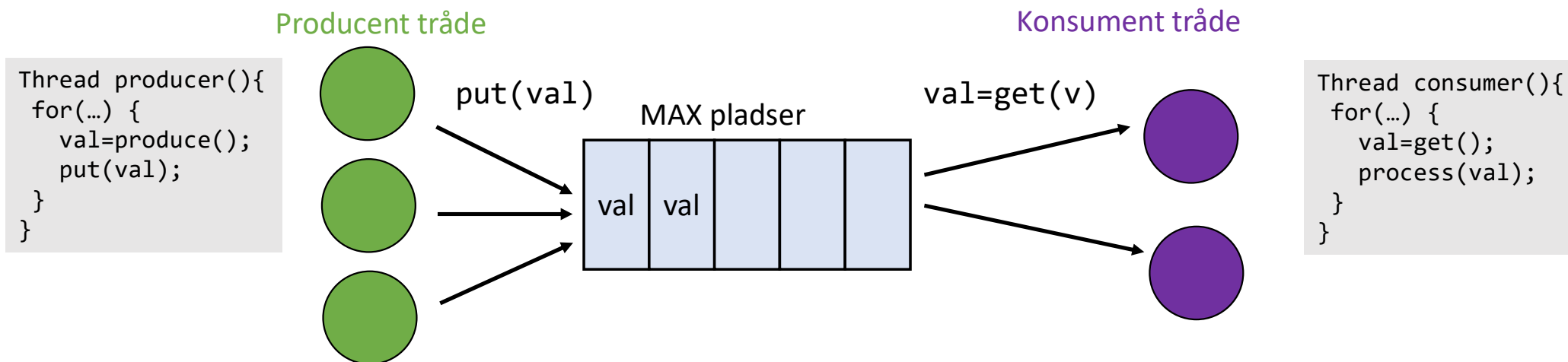
```
pthread_mutex_lock(&lock);  
initialized = 1;  
pthread_cond_signal(&init);  
pthread_mutex_unlock(&lock);
```

- Test og wait skal være atomisk

Producer-consumer problemet

Producer-Consumer problemet

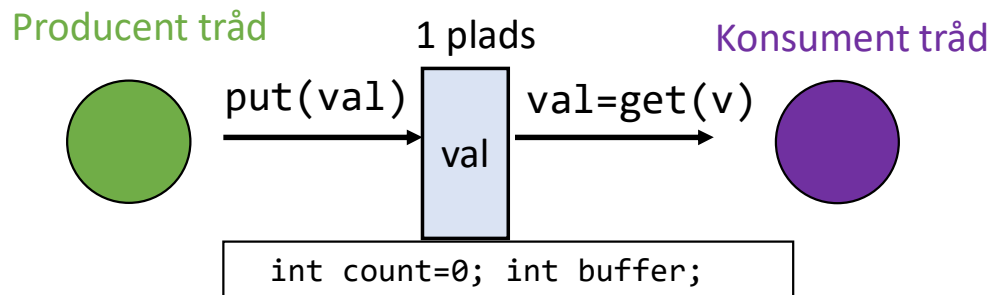
- Concurrent bounded buffer



- En producent, som forsøger `put` på en fyldt buffer skal *blokere* indtil der bliver plads
- En konsument, som forsøger `get` på en tom buffer skal *blokere* indtil der ankommer et element
- Bufferen er en delt ressource: adgang under gensidig udelukkelse
 - En kø af mellemresultater
 - `grep foo file.txt | wc -l`
- Et generelt mønster som ofte bruges i samtidige programmer
- Et skole eksempel til studie af synkronisering

Producer-Consumer v1: enkelt-plads buffer

- 1 producent, 1-plads buffer

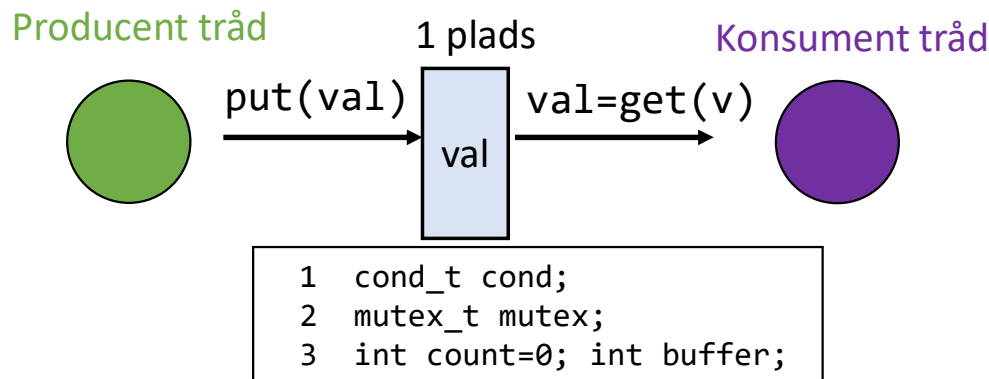


```
1 int buffer;  
2 int count = 0; //tom  
3  
4 void put(int value) {  
5     assert(count == 0);  
6     count = 1;  
7     buffer = value;  
8 }  
9  
10 int get() {  
11     assert(count == 1);  
12     count = 0;  
13     return buffer;  
14 }
```

- Skriv kun data når `count` er 0.
 - Dvs. Når bufferen er *tom*.
- Udlæs kun når `count` er 1.
 - Dvs. Når bufferen *fylt*.

Producer-Consumer v1: enkelt-plads buffer

- 1 producent, 1 konsument, 1-plads buffer



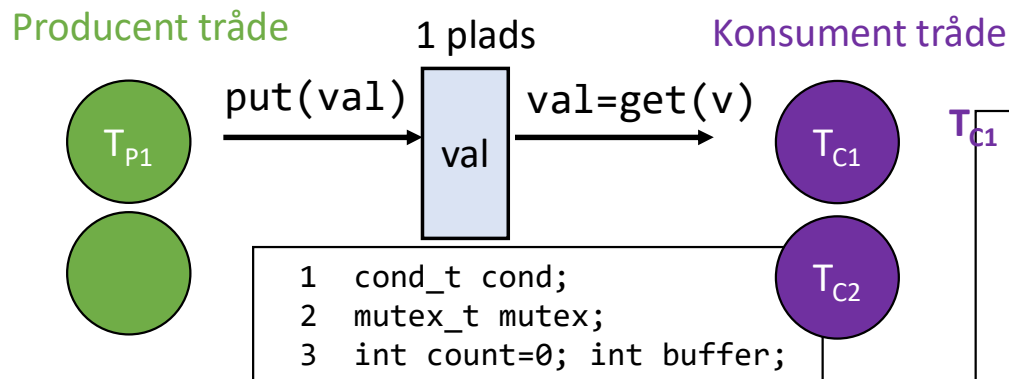
```
4 void *producer(void *arg) {  
5     int i;  
6     for (i = 0; i < loops; i++) {  
7         Pthread_mutex_lock(&mutex);           // p1  
8         if (count == 1)                       // p2  
9             Pthread_cond_wait(&cond, &mutex); // p3  
10        put(i);                                // p4  
11        Pthread_cond_signal(&cond);            // p5  
12        Pthread_mutex_unlock(&mutex);          // p6  
13    }  
14 }  
15
```

```
16 void *consumer(void *arg) {  
17     int i;  
18     for (i = 0; i < loops; i++) {  
19         Pthread_mutex_lock(&mutex);           // c1  
20         if (count == 0)                       // c2  
21             Pthread_cond_wait(&cond, &mutex); // c3  
22         int tmp = get();                       // c4  
23         Pthread_cond_signal(&cond);            // c5  
24         Pthread_mutex_unlock(&mutex);          // c6  
25         printf("%d\n", tmp);  
26     }  
27 }
```

- Vent på cond, hvis buffer er fyldt
- Signalér på cond at buffer ikke længere er tom
- OBS: Højst én ventende process: enten producenten eller konsumenten!
- Vent på cond, hvis buffer er tom
- Signalér på cond at buffer ikke længere er fyldt

Producer-Consumer v2a: enkelt-plads buffer

- **Flere** producenter/konsumenter, 1-plads buffer



1. T_{C1}, T_{C2} er ready $\langle \rangle$
2. T_{C1} : forsøger, blokerer på tom buffer $\langle T_{C1} \rangle$
3. T_{p1} : Producerer element: T_{C1} ready $\langle \rangle$
4. T_{p1} : Producerer element: blokerer $\langle T_{p1} \rangle$
5. T_{C2} : afvikler, tager elementet i buffer
Signalerer $\langle \rangle$
6. T_{C1} : fortsætter og tager fra tom buffer :-)

Vente-betingelsen er ændret af anden tråd

T_{C1}

```

16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         if (count == 0)                       // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                      // c4
23         Pthread_cond_signal(&cond);           // c5
24         Pthread_mutex_unlock(&mutex);         // c6
25         printf("%d\n", tmp);
26     }
27 }
    
```

T_{C2}

```

16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         if (count == 0)                       // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                      // c4
23         Pthread_cond_signal(&cond);           // c5
24         Pthread_mutex_unlock(&mutex);         // c6
25         printf("%d\n", tmp);
26     }
27 }
    
```

Et mod-eksemplets eksekveringsspor

Consumer 1		Consumer 2		Produce2		Count	Comment
	State		State		State		
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Block		Ready		Ready	0	Nothing to get
	Block		Ready	p1	Running	0	
	Block		Ready	p2	Running	0	
	Block		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Block	1	Buffer full; Block
	Ready	c1	Running		Block	1	sneaks in ...
	Ready	c2	Running		Block	1	
	Ready	c4	Running		Block	0	... and grabs data
	Ready	c5	Running		Ready	0	awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Condition K \emptyset : <T_{c1}>

Condition K \emptyset : <>

Condition K \emptyset : <T_{p1}>

Ready <T_{c1}:c4, T_{c2}:c1>


Ready <T_{c1}:c4, T_{c2}:c6>

Semantik af condition variable

- Problemet opstår af simpel årsag
 - Efter producent vækker T_{c1} , men før T_{c1} fik change for at afvikle, ændrede bufferens tilstand af T_{c2} .
 - Den/de vækkede tråd konkurrerer med andre tråde om låsen til kritisk region
 - Ingen garanti for at tilstanden er som den vækkede kræver
- → Mesa semantiks.
 - Næsten alle praktiske systemer med condition variable følger denne semantik
 - (Navngivet efter programmeringssproget Mesa)
 - Incl. Java-monitorer
- Hoare semantik giver den stærkerer garanti at den vækkede tråd afvikler umiddelbart efter opvækning



```
pthread_mutex_lock(&mutex);
```



```
if(...)  
pthread_cond_wait(&cond, &mutex);
```

```
pthread_mutex_unlock(&mutex);
```

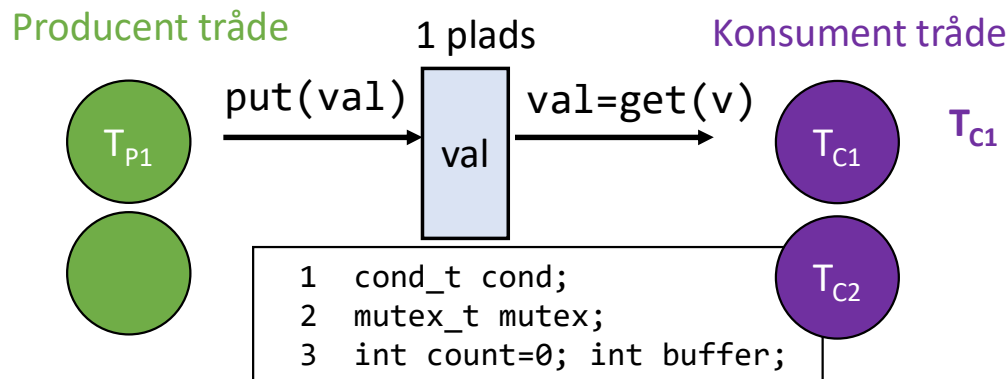
```
pthread_mutex_lock(&mutex);
```

```
if()  
pthread_cond_signal(&cond, &mutex);
```

```
pthread_mutex_unlock(&mutex);
```

Producer-Consumer v2b: enkelt-plads buffer

- **Flere** producenter/konsumenter, 1-plads buffer



```
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex);           // p1
8         while (count == 1)                    // p2
9             Pthread_cond_wait(&cond, &mutex); // p3
10        put(i);                                // p4
11        Pthread_cond_signal(&cond);            // p5
12        Pthread_mutex_unlock(&mutex);          // p6
13    }
14 }
15
```

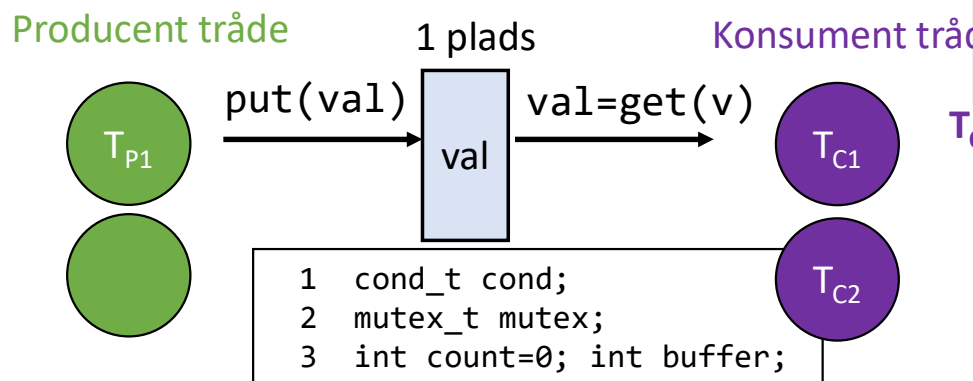
- WHILE til gen-check af betingelsen

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                    // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&cond);            // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

- Vent på cond, hvis buffer er tom
- Signalér på cond at buffer ikke længere er fyldt

Producer-Consumer v2b: enkle

- **Flere** producenter/konsumenter, 1-plads buffer



1. T_{C1}, T_{C2}, T_{P1} er ready $0 <>$
2. T_{C1} : forsøger, blokerer på tom buffer $0 < T_{C1} >$
3. T_{C2} : forsøger, blokerer på tom buffer $0 < T_{C1}, T_{C2} >$
4. T_{P1} : Producerer element: T_{C1} ready $1 < T_{C2} >$
5. T_{P1} : Producerer element: blokerer $1 < T_{C2}, T_{P1} >$
6. T_{C1} : afvikler, tager elem og signalerer $0 < T_{P1} >$
7. T_{C2} : fortsætter og gen-check: $0 < T_{C2}, T_{P1} >$
8. T_{C1} : forsøger, blokerer på tom buffer $0 < T_{C2}, T_{P1}, T_{C1} >$

En konsument må ikke vække andre konsumenter

T_{P1}

```

4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex);           // p1
8         while (count == 1)                    // p2
9             Pthread_cond_wait(&cond, &mutex); // p3
10        put(i);                                // p4
11        Pthread_cond_signal(&cond);           // p5
12        Pthread_mutex_unlock(&mutex);         // p6
13    }
14 }
    
```

T_{C1}

```

16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                    // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22        int tmp = get();                       // c4
23        Pthread_cond_signal(&cond);           // c5
24        Pthread_mutex_unlock(&mutex);         // c6
25        printf("%d\n", tmp);
26    }
27 }
    
```

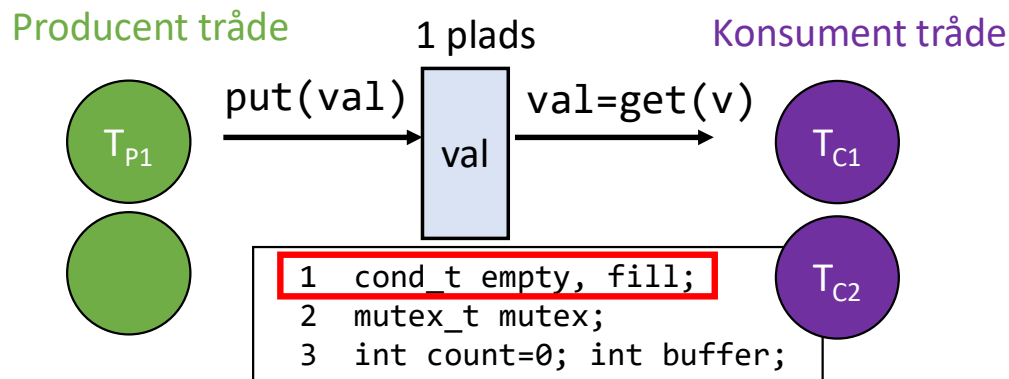
T_{C2}

```

16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                    // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22        int tmp = get();                       // c4
23        Pthread_cond_signal(&cond);           // c5
24        Pthread_mutex_unlock(&mutex);         // c6
25        printf("%d\n", tmp);
26    }
27 }
    
```


Producer-Consumer v2c: enkelt-plads buffer

- **Flere** producenter/konsumenter, 1-plads buffer



```
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex);           // p1
8         while (count == 1)                    // p2
9             Pthread_cond_wait(&fill, &mutex); // p3
10        put(i);                                // p4
11        Pthread_cond_signal(&empty);           // p5
12        Pthread_mutex_unlock(&mutex);          // p6
13    }
14 }
15
```

- WHILE til gen-check af betingelsen

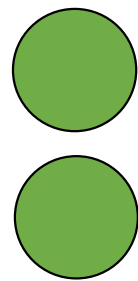
```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                    // c2
21             Pthread_cond_wait(&empty, &mutex); // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&fill);           // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

- Vent på cond, hvis buffer er tom
- Signalér på cond at buffer ikke længere er fyldt

Producer-Consumer V3

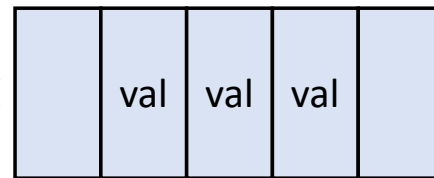
- Flere bufferpladser, højere grad af samtidighed, færre kontekst skift

Producent tråde



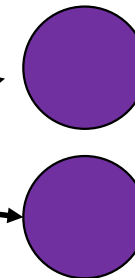
put(val)

MAX pladser



val=get(v)

Konsument tråde

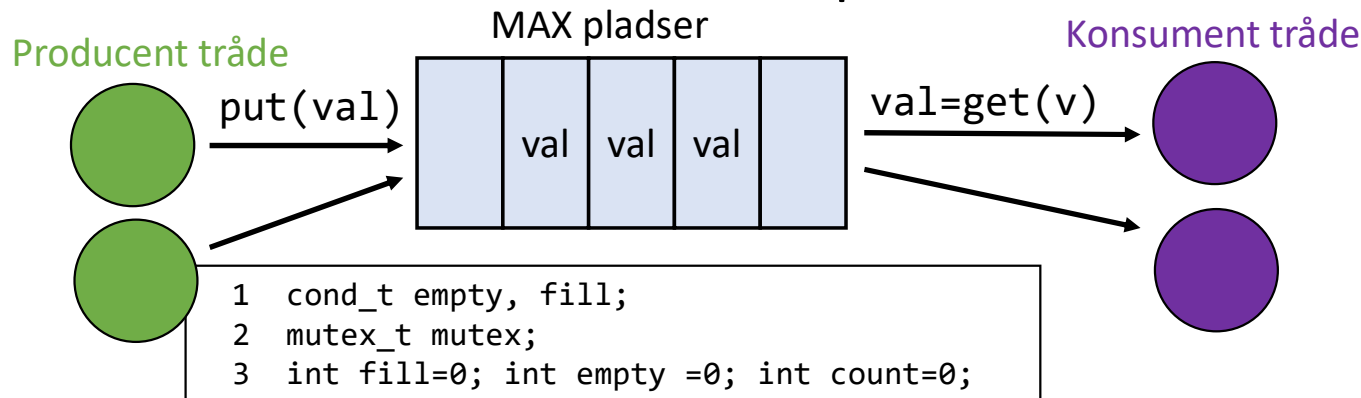


```
1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
```

```
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

Producer-Consumer v3:

- **Flere** producenter/konsumenter, MAX-plads buffer



```
4 void *producer(void *arg) {
5     int i;
6     for (i = 0; i < loops; i++) {
7         Pthread_mutex_lock(&mutex);           // p1
8         while (count == MAX)                  // p2
9             Pthread_cond_wait(&fill, &mutex); // p3
10        put(i);                               // p4
11        Pthread_cond_signal(&empty);          // p5
12        Pthread_mutex_unlock(&mutex);         // p6
13    }
14 }
15
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                   // c2
21             Pthread_cond_wait(&empty, &mutex); // c3
22         int tmp = get();                     // c4
23         Pthread_cond_signal(&fill);          // c5
24         Pthread_mutex_unlock(&mutex);         // c6
25         printf("%d\n", tmp);
26     }
27 }
```

p2: En producent blokerer kun hvis alle buffer pladser er fyldt.

c2: En konsument blokerer kun hvis alle buffer pladser er tomme.

Dækning af betingelser

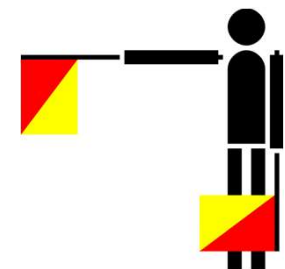
- Antag en hukommelsesallokator har 0 bytes ledige
 - Tråd T_a kalder `allocate(100)`.
 - Tråd T_b kalder `allocate(10)`.
 - Både T_a og T_b averter på en condition og blokerer.
 - Tråd T_c kalder `free(50)`.
- Hvilken tråd skal afvikle?
 - Hvis tråd T_a vækkes, blokerer den bare igen!!!
- Erstat `pthread_cond_signal()` med `pthread_cond_broadcast()`
 - **Vækker alle ventende tråde.**
 - Pris: for mange tråde vækkes
 - Tråde, som ikke kan fortsætte blokerer når de gen-checker betingelsen
 - Altid sikkert (pga. while!)
- (Java monitorer: `notify()` vs. `notifyAll()`)

```
1 // how many bytes of the heap are free?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // need lock and condition too
5 cond_t c;
6 mutex_t m;
7
8
9 void * allocate(int size) {
10     Pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         Pthread_cond_wait(&c, &m);
13     void *ptr = ...; // get mem from heap
14     bytesLeft -= size;
15     Pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     Pthread_mutex_lock(&m);
21     bytesLeft += size;
22     Pthread_cond_signal(&c); // whom to signal??
23     Pthread_mutex_unlock(&m);
24 }
```

Semaforer

Semaforer

- En anden klassisk *synkroniserings- og mutex* mekanisme
- Et objekt som indeholder en heltallig tællerværdi (og venteliste) med operationerne:
 - `sem_init`: initialiserer semaforen og dens værdi
 - `sem_wait()`
 - Blokkerer kaldende tråd hvis tællerværdi ≤ 0
 - Fratrækker tællerværdien 1
 - `sem_post()`
 - Øger værdien med 1
 - Hvis afventende (blokerede) tråde, vækkes én
- Operationerne er atomiske
- Alternative navne
 - Wait / Signal
 - P / V (efter Dijkstra: probieren eller Verhoeen)



Semafor ~signaleringsflag

Semaforer: Posix API

```
1  #include <semaphore.h>
2  sem_t s;
3  sem_init(&s, 0, 1); // initialize s to the value 1
```

- Erklærer semaforen `s` og initialiserer tællerværdien til **1**
- Andet argumentet **0** angiver at semaforen deles blandt tråde *internt i samme proces*. (ikke delt med tråde i andre processer)

```
4  int sem_wait(sem_t *s);
```

- Hvis tæller er ≥ 1 ; dekrementér tæller og returner
- Hvis tæller ≤ 0 ; dekrementér og blokér (afvent et post)
- Når tælleren er negativ, angiver værdien antallet af afventende tråde

```
5  int sem_post(sem_t *s);
```

- Inkrementerer tællerværdien med **1**
- Vækker én af afventende tråd (hvis nogen)

Semaforer til mutex 1

- En semafor initialiseres til 1: een tråd kan passere wait

```
1 sem_t m;  
2 sem_init(&m, 0, 1); //start værdi: 1  
3  
4 sem_wait(&m);  
5 //critical section here  
6 sem_post(&m);
```

Semaforens værdi	Tråd 1	Tråd 2
1		
1	kald sema_wait(&m)	
0	sem_wait(&m) returner	
0	(kritisk region)	
0	kald sem_post(&m)	
1	sem_post(&m) returner	
1		kald sem_wait(&m)
0		sem_wait(&m) returner
0		(kritisk region)
0		kald sem_post(&m)
1		sem_post(&m) returner

Semaforer til mutex 2

- En semafor initialiseres til 1: een tråd kan passere wait

```
1 sem_t m;  
2 sem_init(&m, 0, 1); //start værdi: 1  
3  
4 sem_wait(&m);  
5 //critical section here  
6 sem_post(&m);
```

Semaforens værdi	Tråd 1	Tråd 2
1		
1	kald sema_wait(&m)	
0	sem_wait(&m) returner	
0	(kritisk region)	kald sem_wait(&m)
-1		sem<=0: bloker
-1	kald sem_post(&m)	
0	sem_post(&m) returner	//ready
0		sem_wait(&m) returner
0		(kritisk region)
0		kald sem_post(&m)
1		sem_post(&m) returner

Semaforer til synkronisering

- Tråd P skal afvente at tråd Q er færdig med en initialiserings-procedure ?
 - Lav en semafor med startværdien **0**.

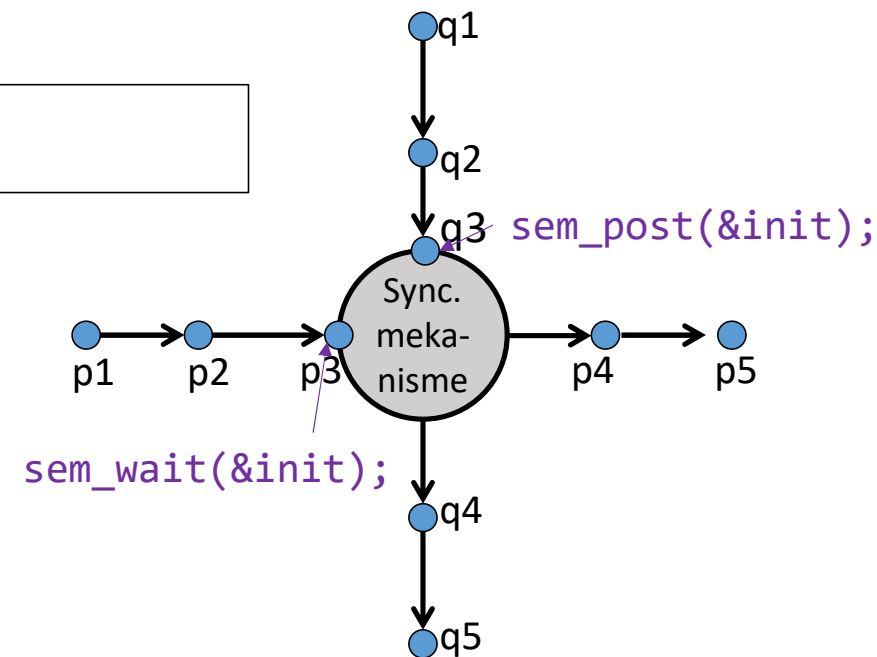
```
1 sem_t init; //semafor til signalering af endt initialisering
2 sem_init(&init, 0, 0); //start værdi: 0
```

Tråd P

```
...
p2 sem_wait(&init);
p4 ...
```

Tråd Q

```
...
q3 sem_post(&init);
q4 ...
```



Semaforer til synkronisering

Tråd P
 p2 sem_wait(&init);
 p4 ...

Tråd Q
 q3 sem_post(&init);
 q4 ...

- Tråd P kører først.

init	Tråd P	Tråd Q
0	... p2	
0	p3 kald sem_wait(&init)	
-1	//init<=0: bloker	
-1		... q2
-1		q3:kald sem_post(&init)
0	//ready	sem_post(&init)retur
0		p4...
0	sem_wait(&m) retur	
0	p4 ...	

- Tråd Q kører først

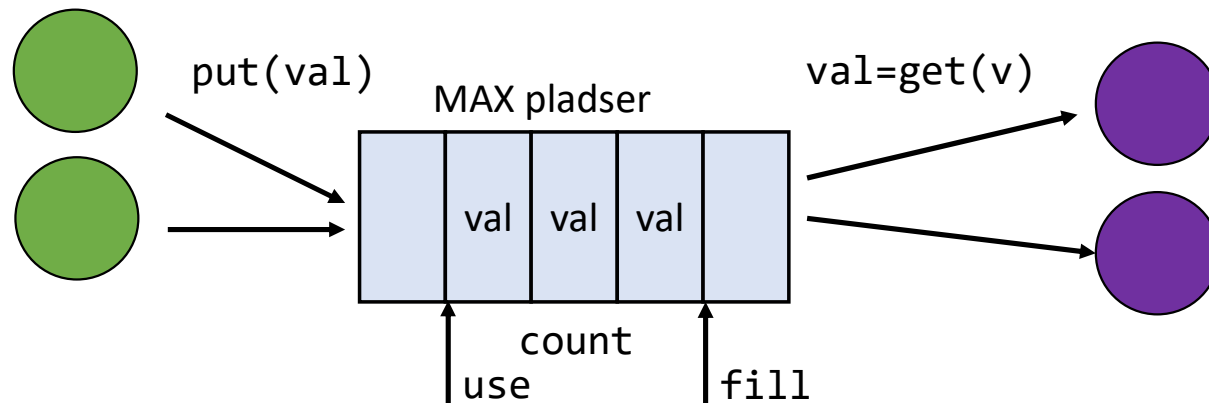
init	Tråd P	Tråd Q
0		... q2
0		q3: kald sem_post(&init)
1		sem_post(&init) retur
1		p4...
1	... p2	
1	p3 kald sem_wait(&init)	
0	//init>0: fortsæt	
0	sem_wait(&m) retur	
0	p4 ...	

Producer-consumer med semaforer

Producer-Consumer med semaforer

Producent tråde

Konsument tråde

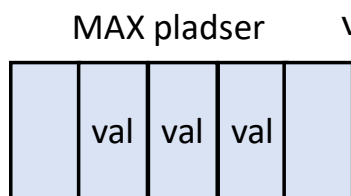
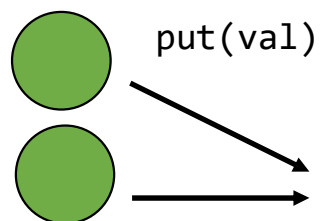


```
1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
```

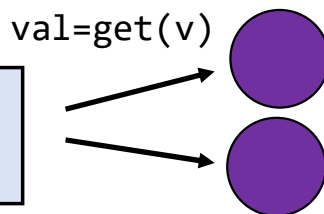
```
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

Producer-Consumer med semaforer v1?

Producent tråde



Konsument tråde



```
1 sem_t empty; //Vent hvis buffer tom
2 sem_t full; //Vent hvis buffer fuld
3 sem_init(&empty, 0, MAX); // MAX ledige pladser til start
4 sem_init(&full, 0, 0); // 0 er brugte
5 sem_t muxtex; sem_init(&muxtex,0,1); //mutex på buffer
```

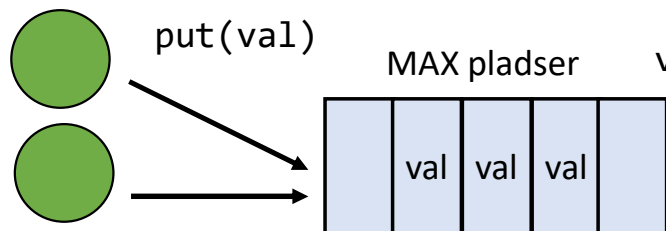
- Lad en semafor (empty) tælle antallet n af ledige pladser: lad n producenter passere
- Lad en semafor (full) tælle antallet m af brugte pladser: lad m konsumenter passerer
- $m+n==MAX$

```
6 void *producer(void *arg) {
7     int i;
8     for (i = 0; i < loops; i++) {
9         sem_wait(&muxtex);
10        sem_wait(&empty);
11        put(i);
12        sem_post(&full);
13        sem_post(&muxtex);
14    }
15 }
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&muxtex);
20         sem_wait(&full);
21         int tmp = get();
22         sem_post(&empty);
23         sem_post(&muxtex);
24         printf("%d\n", tmp);
25     }
26 }
```

Producer-Consumer med semaforer v1?

Producent tråde



Konsument tråde

```
1 sem_t empty;           //Vent hvis buffer tom
2 sem_t full;            //Vent hvis buffer fuld
3 sem_init(&empty, 0, MAX); // MAX ledige pladser til start
4 sem_init(&full, 0, 0);   // 0 er brugte
5 sem_t mutex; sem_init(&mutex,0,1); //mutex på buffer
```

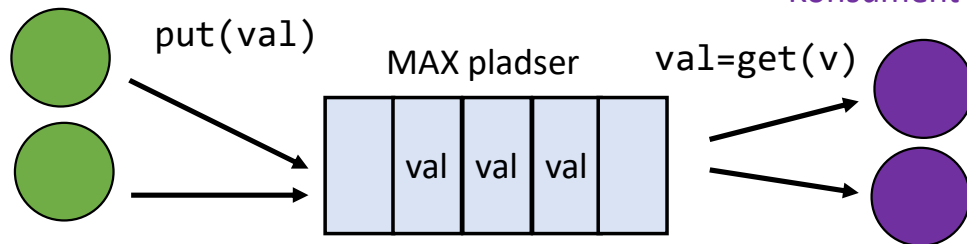
```
16 void *consumer(void *arg)
17 {
18     int i;
19     for (i = 0; i < loops; i++) {
20         sem_wait(&mutex);
21         sem_wait(&full);
22         int tmp = get();
23         sem_post(&empty);
24         sem_post(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

1. Buffer er tom
2. Konsument **tager og holder mutex** (linie 19).
3. Konsument **kald**er `sem_wait()` på `full` semaforen (linie 20).
4. Konsument blokeres og CPU frigives.
 - Men consumer holder stadig mutex!
5. Producer **kald**er `sem_wait()` på `mutex` semafor (line 9).
6. Producer blokerer da den allerede er optaget!
 - **En klassisk baglås (deadlock): en evig indbyrdes venten!**

Producer-Consumer med semaforer v2

Producent tråde

Konsument tråde



```
1 sem_t empty; //Vent hvis buffer tom
2 sem_t full; //Vent hvis buffer fuld
3 sem_init(&empty, 0, MAX); // MAX ledige pladser til start
4 sem_init(&full, 0, 0); // 0 er brugte
5 sem_t mutex; sem_init(&mutex,0,1); //mutex på buffer
```

- Byt rundt på full og mutex (ditto empty og mutex i producer)
- OK?
- Ja, da full ikke slipper flere konsumenter igennem end at de kan gennemføre den kritiske region

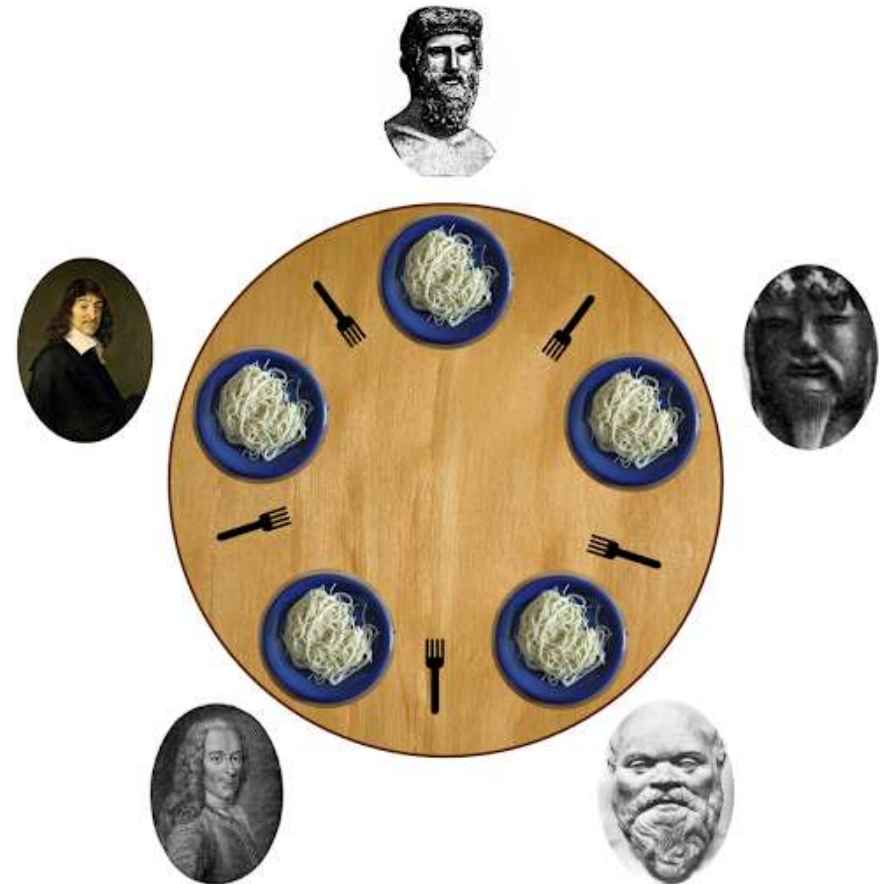
```
6 void *producer(void *arg) {
7     int i;
8     for (i = 0; i < loops; i++) {
9         sem_wait(&empty);
10        sem_wait(&mutex);
11        put(i);
12        sem_post(&mutex);
13        sem_post(&full);
14    }
15 }
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);
20         sem_wait(&mutex);
21         int tmp = get();
22         sem_post(&mutex);
23         sem_post(&empty);
24         printf("%d\n", tmp);
25     }
26 }
```


De spisende filosofen

De spisende filosoffer

- 5 filosoffer sidder ved et rundt bord og spiser spaghetti
- Mellem hver filosof er der én gaffel!
- De skal bruge 2 gafler (hhv. til højre og til venstre) bestik for at kunne spise
- Dvs. der er konkurrence om at få disse



<http://devzone.connectivelogic.co.uk/default/The%20Dining%20Philosophers%20Problem.html>

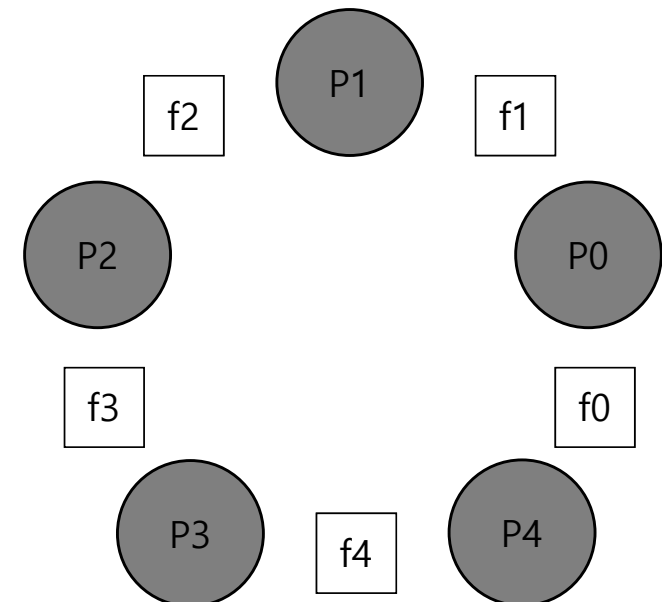
De spisende filosoffer

- En løsning kræver:
 - Ingen deadlock.
 - Alle spiser; ingen **udsultes**.
 - Høj grad af **samtidighed**.
- Filosofs p 's venstre gaffel: \rightarrow kald `left(p)`.
- Filosofs p 's højre gaffel: \rightarrow call `right(p)`.

```
// helper functions
int left(int p) { return p; }

int right(int p) {
    return (p + 1) % 5;
}
```

```
Thread philosopher(int p) {
    while (1) {
        think();
        getforks();
        eat();
        putforks();
    }
}
```



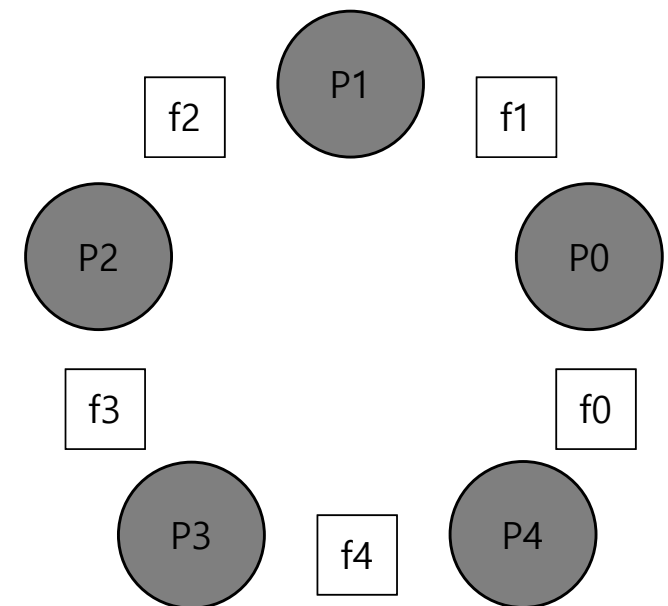
De spisende filosoffer

- En gaffel repræsenteres som en semafor

```
sem_t forks[N];  
for(int s=0;s<N;s++) sem_init(&forks[s],0,1);
```

```
1 void getforks(int p) {  
2     sem_wait(forks[left(p)]);  
3     sem_wait(forks[right(p)]);  
4 }  
5  
6 void putforks() {  
7     sem_post(forks[left(p)]);  
8     sem_post(forks[right(p)]);  
9 }
```

```
Thread philosopher(int p) {  
    while (1) {  
        think();  
        getforks();  
        eat();  
        putforks();  
    }  
}
```



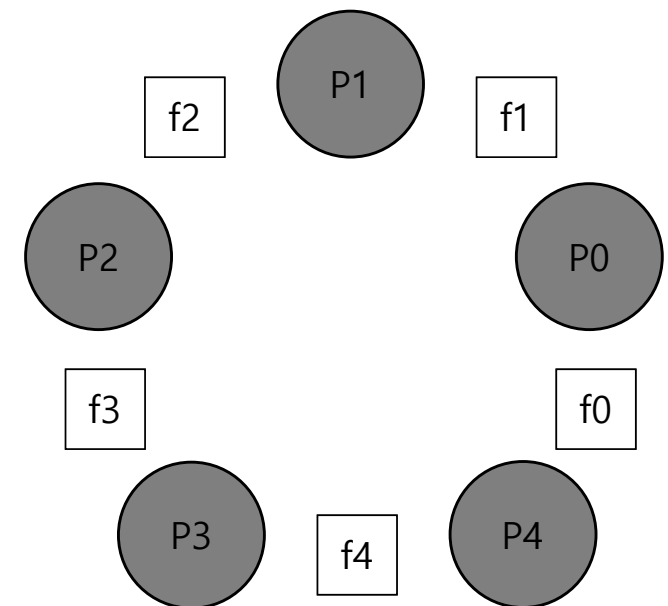
De spisende filosoffer

- En gaffel repræsenteres som en semafor

```
sem_t forks[N];  
for(int s=0;s<N;s++) sem_init(&forks[s],0,1);
```

```
1 void getforks(int p) {  
2     sem_wait(forks[left(p)]);  
3     sem_wait(forks[right(p)]);  
4 }  
5  
6 void putforks() {  
7     sem_post(forks[left(p)]);  
8     sem_post(forks[right(p)]);  
9 }
```

```
Thread philosopher(int p) {  
    while (1) {  
        think();  
        getforks();  
        eat();  
        putforks();  
    }  
}
```



Deadlock kan indtræffe!

- Filosoferne vil spise ca. samtidigt, og hver filosof formår at **låse venstre gaffel** inden nogen anden når at tage højre gaffel .
- Hver sidder fast med en gaffel og afventer den anden: **forevigt!**

De spisende filosoffer

- En gaffel repræsenteres som en semafor

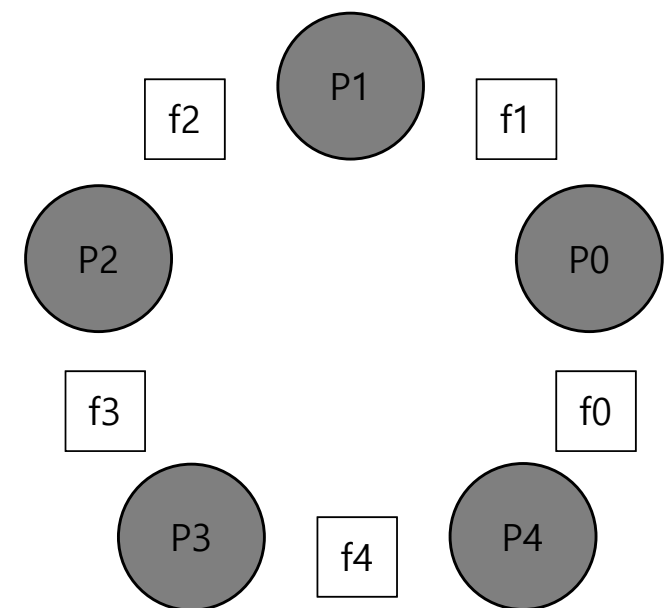
```
sem_t forks[N];  
for(int s=0;s<N;s++) sem_init(&forks[s],0,1);
```

```
1 void getforks(int p) {  
2     if(p==4 ) { //filo Freud  
3         sem_wait(forks[right(p)]);  
4         sem_wait(forks[left(p)]);  
5     }  
6     else {  
7         sem_wait(forks[left(p)]);  
8         sem_wait(forks[right(p)]);  
9     }
```

Deadlock kan undgås ved at en filo låser i en anden rækkefølge

Mulighed at cirkulær venten opstår brydes

```
Thread philosopher(int p) {  
    while (1) {  
        think();  
        getforks();  
        eat();  
        putforks();  
    }
```



Baglås / Deadlock

Deadlocks (Baglåse)

- Der er baglås, når der er opstået et cirkel blandt en mængde tråde hvor de hver venter på en ressource, som fastholdes af en anden

Thread 1:

lock(L1);

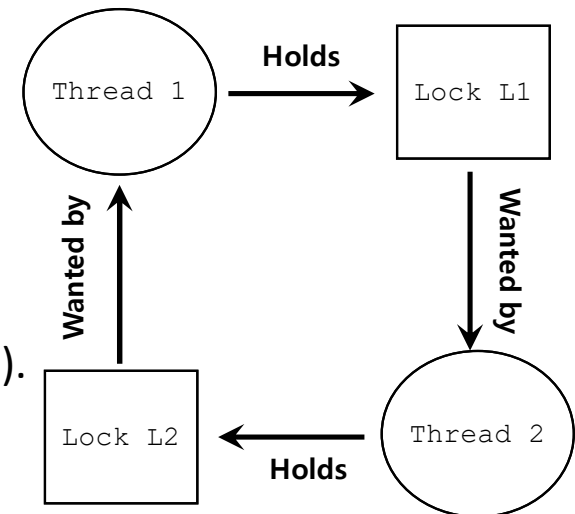
lock(L2);

Thread 2:

lock(L2);

lock(L1);

- Thread1 holder lås **L1** og venter på en anden, **L2**.
- Thread2 holder lås **L2** og venter på **L1** (som holdes af **Thread 1**).



Betingelser for deadlock

Betingelse	Beskrivelse
Mutual Exclusion	Tråde kan kræve eksklusiv adgang til de ressourcer de skal bruge
Hold-and-wait	En tråd kan holde fast i en tildelt ressource, mens den venter på adgang til flere. (<i>"Stykvis allokering"</i>)
No preemption	Ressources kan ikke magtfuldt fratvinges de tråde som holder dem.
Circular wait	Der findes en cirkulær kæde af tråde, således at hver tråd holder en eller flere ressourcer som efterspørges af næste tråd i kæden

- **Nødvendige** betingelser for at deadlock potentielt kan opstå: Mutex, hold&wait, no-preemption
- **Tilstrækkelig** betingelse: cirkulær venten er opstået.

Overordnede strategier mod deadlock

- (Prevention) Forebygge at deadlock kan opstå
 - Bryde en af de 4 betingelser for deadlock:
 - Fjerne muligheden for mutex, hold&wait, eller no-preemption
 - Forhindre at cirkulær venten kan opstå
- (Avoidance) Undvige deadlock:
 - Scheduler bruge information om hvilke ressourcer der findes, kender trådenes behov, og undviger de afviklingssekvenser som potentielt leder til baglås
- (Recovery): Genopretning
 - Detektér om deadlock er opstået og ryd op.

Forebyggelse: cirkulær venten

- Lav en **total ordning** aflåsningsrækkefølge, som alle tråde overholder
 - En låse protokol
 - Ulempe: låse forrest i ordning låses i (unødigt) lang tid
- **Ex:**
 - To låse L1 og L2: Tag altid L1 før L2.

Thread 1:	Thread 2:
lock(L1);	lock(L2);
lock(L2);	lock(L1);



Thread 1:	Thread 2:
lock(L1);	lock(L1);
lock(L2);	lock(L2);

- EX: de spisende filosoffer: fork0 → fork1 → fork2 → fork3 → fork4

- ~~• P0: fork0 → fork1~~
- ~~• P1: fork1 → fork2~~
- ~~• P2: fork2 → fork3~~
- ~~• P3: fork3 → fork4~~
- ~~• P4: fork4 → fork0~~

P0: fork0 → fork1

P1: fork1 → fork2

P2: fork2 → fork3

P3: fork3 → fork4

P4: fork0 → fork4 //Freud

Forebyggelse: hold&vent

- Tag alle ressourcer som tråden skal bruge på én gang atomisk
 - `lockAll(<list of locks>);`
- Kan emuleres med:
 - Ingen anden tråd kan begynde låsning
- Problem:
 - Låsebehov skal kendes på forhånd
 - Mindsker samtidighed (låse holdes unødigt længe)

```
1    lock(prevention);  
2    lock(L1);  
3    lock(L2);  
4    ...  
5    unlock(prevention);
```

Forebyggelse: preemption

- Generel fra-tvingning er problematisk, da tråden jo kan være i en kritisk region og opdatere mange forskellige ressourcer på program specifik måde:
 - Tilbageførsel ("undo") af ændringer ("roll-back") så delte ressourcer kommer i kendt og konsistent tilstand
 - Transaktioner a la databaser
- Hvad med andre ressource typer? Printer?

```
1  lock(mutex);  
3  kontoA+=100  
4  kontoB-=100;  
5  unlock(mutex);
```

Forebyggelse: "preemption light"

- "Preemption light"

- tryLock(L):

- returnerer 0 hvis låsen var ledig
 - returnerer fejlkode, hvis låsen var taget

- Problem 1: Mulighed for **Livelock**

- To tråde afvikler samme sekvens om og om igen, uden at gøre fremskridt

```
1  while(retry){
2      lock(L1);
3      if( tryLock(L2) != 0 )
4          unlock(L1);
5      else retry=0;
6  }
```

```
1  while(retry){
2      lock(L1);
3      if( tryLock(L2) != 0 )
4          unlock(L1);
5      else retry=0;
6  }
```

```
1  while(retry){
2      lock(L2);
3      if( tryLock(L1) != 0 )
4          unlock(L2);
5      else retry=0;
6  }
```

- Risiko kan mindskes ved at tilføje tilfældigt forsinkelse når tryLock fejler: sleep(rand());
- Problem 2: Hvad hvis trådene havde lavet ændringer i variable efter lock, men tryLock fejler? Roll-back?
- Problem 3: større programmer: hvornår hhv lock() og tryLock() ???

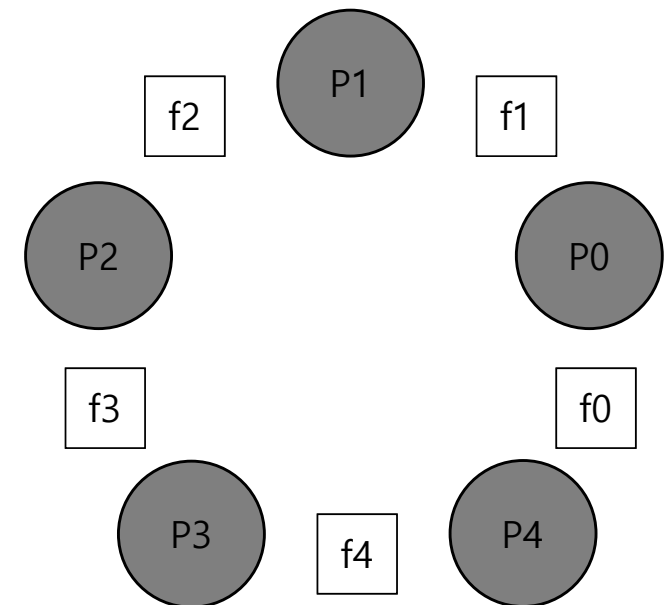
Forebyggelse: preemption light: De spisende filosoffer

- En gaffel repræsenteres som en semafor

```
sem_t forks[N];  
for(int s=0;s<N;s++) sem_init(&forks[s],0,1);
```

```
1 void getforks(int p) {  
2     int retry=1;  
3     while(retry) {  
4         sem_wait(forks[left(p)]);  
5         if(sem_tryWait(forks[right(p)])!=OK)  
6             sem_post(forks[left(p)]);  
7         else retry=0;  
8     }  
9 }  
10
```

```
Thread philosopher(int p) {  
    while (1) {  
        think();  
        getforks();  
        eat();  
        putforks();  
    }  
}
```



Ingen deadlock, men mulighed for Livelock

Forebyggelse: “mutex”

- Låse-frie datastrukturer
- Anvender særlige maskin-instruktioner (atomiske)
- Fx atomisk optælling af x:
AtomicIncrement(&x,42);
- Problem:
 - Generalitet ???
 - Hvad med ressourcer som printer og filer??
 - Hvilke datastrukturer kan laves lock-frie?
 - Busy-wait
 - Livelock

*//Pseude-kode beskriver adfærd af instuktion vha. C-kode
//Dette er ikke en C-funktion!!!*

```
1  int instruction CompareAndSwap(  
    int *address,  
    int expected,  
    int new){  
2      if(*address == expected){  
3          *address = new;  
4          return 1; // success  
5      }  
6      return 0;  
7  }
```

```
1  void AtomicIncrement(int *value, int amount){  
2      do{  
3          int old = *value;  
4      }while( CompareAndSwap(value, old, old+amount)==0);  
5  }
```


(Avoidance) Undvigelse:

- Scheduler "smart" deadlock undviges
- Fx: Antag OS kender:
 - Hvilke låse der findes,
 - Hvilke tråde der findes,
 - Hvert tråds max-behov for hvilke låse
 - Antagelse: hvis trådene låser op til max, så vil de frigive ressourcerne igen på et eller andet tidspunkt
 - På et hvert tidspunkt: hvilke tråde holder hvilke låse
- OS undersøger om en lock(1) er "sikker" at udføre

Låse: L1, L2

Tråde: T1, T2, T3, T4

Max ressource-**behov** matrice:

Tråde/låse	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

Allokeret/holder/Ejer matrice:

Tråde/låse	T1	T2	T3	T4
L1	no	yes	no	no
L2	no	no	no	no

1. T1 kommer og vil have L2: **Nægtes / Blokeres da**,
T2 skal bruge L2 inden (vi er sikre på at) L1 leveres tilbage, og
T1 skal bruge L1 inden (vi er sikre på at) L2 leveres tilbage
2. T3 vil have L2: **OK**
T3 har ikke behov for yderligere låse inden L2 kan leveres tilbage

Bank-mandens algoritme: Banken udlåner aldrig flere ressourcer, end den er sikker på at den kan få tilbage

Detektion og Genopretning

- Lad deadlocks indtræffe
- OS afvikler deadlock detektions algoritme: find cykler i ressource allokeringsgraf
 - Genstart systemer?!
 - Dræb en af processerne: inkonsistent ressource tilstand
- Anvendes tit i databaser:
 - Abortér en af transaktionerne og roll-back
- En optimistisk strategi,
 - potentielt effektiv hvis deadlock er sjælden
 - Ret ineffektiv hvis det sker relativt ofte

Deadlock resumé

- Deadlock opstår når der er en cirkulær indbyrdes evig venten
- Der er flere strategier til at forhindre deadlock
 - Ingen uden ulemper
 - “pick your poison”
- Lav en total låsningsorden er ofte et nyttigt

De 9 bud

1. Tråde's relative hastigheder er uforudsigelig
2. Beskyt delte ressourcer vha. gensidig udelukkelse
 - Brug ikke scheduling (prioriteter + frivillig tidsdeling) istedet for mutex
3. Brug synkroniserings-primitiver, ikke "dovent" flag=1 og IF(flag)
4. Antag ikke ordning af vækkede tråde ved mutexes (semaforer, locks, conditions)
5. Er biblioteket thread safe?
 - Nu om dage mest relevant for hjemmelavede biblioteker / ikke standart pakker
6. Brug kun tryLock (og lign. med timeout) *ekstremt* undtagelsesvist
7. Undgå busy waiting
8. Check for race-conditions, baglås, udsultning med en dosis paranoia
9. Forhindre baglås i at opstå
 - Lav fx en låseprotokol med aftalt totalt ordnet låsningsrækkefølge