Semaphores

Mutual exclusion revisited

Recall the mutual exclusion problem for n threads

Coarse Grained Solution

We used an array of n boolean variables *in*

$$\text{define } \#in \triangleq \sum_{i \in \{1,..,n\}} toInt(in[i])$$
$$\text{where } toInt(\mathit{false}) = 0 \text{ and } toInt(\mathit{true}) = 1$$

```
bool $in[1:n] := ([n]\mathit{false});$
# global invariant: $0 \le \#in \le 1$
process CS[$i = 1$ to $n$] {
    while ($true$) {
        noncritical section
        $\langle \textbf{await}(\#in = 0)\ in[i] := true; \rangle$
        critical section
        ## $in[i]$        — by disjoint variables
        $\langle in[i] := \mathit{false}; \rangle$ } }
```

Data refining – augment, transform, diminish

We will augment with a *tracking variable* s which is 1 minus the number in. Its invariant is

$$s = 1 - \#in \qquad \text{(and hence } s \in \{0, 1\}\text{)}$$

So $s = 1$ means that no thread is in the critical section, while $s = 0$ means 1 thread is in the critical section.

```
## bool $in[1 : n] := ([n]false)$;
int $s := 1$;
# global invariant: $0 \leq \#in = 1 - s \leq 1$
process CS[$i = 1$ to $n$] {
    while ($true$) {
        noncritical section
        $\langle \textbf{await}(s > 0)\ in[i] := true; s := s - 1; \rangle$
        critical section
        ## $in[i]$        (and hence $s = 0$)
        $\langle in[i] := false; s := s + 1 \rangle$ } }
```

This is very similar to our lock variable solution. The relation is

$$lock = (s > 0)$$

Producer and consumer farms

Earlier we looked at a single producer sending information to a single consumer. Now we allow lots of produers and lots of consumers. Each message produced is to be consumed exactly once.

We have n threads; m are producers that produce things and the rest are consumers that use the things. They communicate by a shared buffer *buf*.

Each product should be consumed once and only once.

We now need mutual exclusion on the buffer.

Count the number of things produced (p) and the number consumed (c)

Global invariant: $0 \le p - c \le 1$

Therefore $p = c$ or $p = c + 1$.

Global invariant for mut. ex.: $0 \le \#in \le 1$

Exercise: Show that the algorithm on the next slide is interference free. Hint, from $0 \le \#in \le 1$ you can see that

$$in[i] \Rightarrow \neg in[j], \text{ for } i \neq j$$

```
**thing** $buf$;
**int** $p := 0, c := 0$ ;
**bool** $in[1 : n] := ([n]false)$
## Global Inv: $0 \le p - c \le 1$
## Global Inv: $0 \le \#in \le 1$

**process** Producer[$i = 1$ **to** $m$] {
        **while**(... ) {
                $\langle$**await**( $p = c \wedge \#in = 0$**)** $in[i] := true;\rangle$
                *fill buf*
                ## $p = c \wedge in[i]$
                $\langle p := p + 1; in[i] := false;\rangle$ } }

**process** Consumer[$i = m + 1$ **to** $n$] {
        **while**( ... ) {
                $\langle$**await**( $p > c \wedge \#in = 0$**)** $in[i] := true;\rangle$
                *empty buf*
                ## $p = c + 1 \wedge in[i]$
                $\langle c := c + 1; in[i] := false;\rangle$ } }
```

The protocol cycles through 4 states: empty, filling, full, emptying.

State	$\#in$	$p = c$	$p > c$
empty	0	$true$	$false$
filling	1	$true$	$false$
full	0	$false$	$true$
emptying	1	$false$	$true$

However, the difference between the emptying and filling states only matters to the thread that is doing the emptying and filling. All other threads only need to know that they are excluded from doing anything. The thread that is emptying or filling of course knows which it is doing. So we only need to represent three states: empty, full, busy.

We could now data refine using one variable to represent the state.

Instead, for reasons that you'll soon see, we use two variables (e and f) to represent the three states

State	$\#in$	$p = c$	$p > c$	e	f
empty	0	$true$	$false$	1	0
busy	1	?	?	0	0
full	0	$false$	$true$	0	1

The new global invariant is

$$\begin{aligned} & e, f \in \{0,1\} \\ \wedge\ & (e = 1 \Rightarrow p = c \wedge 0 = \#in) \\ \wedge\ & (f = 1 \Rightarrow p > c \wedge 0 = \#in) \\ \wedge\ & (e = f = 0 \Rightarrow 1 = \#in) \end{aligned}$$

As an aside, I'll note two consequences of our global invariants:

$$0 \leq e + f \leq 1$$

and

$$e + f = 1 - \#in$$

Now we data refine, replacing in, p, and c with e and f.

Producer and consumer farms: Final version

```
**thing** $buf;$
## **int** $p := 0, c := 0$ ; ## Global Inv: $0 \le p - c \le 1$
## **bool** $in[1 : n] := ([n]false)$
**int** $e := 1;$
**int** $f := 0;$## Global Inv: see previous page for $e$ and $f$.

**process** Producer[i = 1 **to** m] {
    **while**(... ) {
        # Wait until Empty; move to Busy
        $\langle$**await**( $e > 0$) $in[i] := true; e := e - 1;\rangle$
        *fill buf*
        # Move to Full
        $\langle p := p + 1; in[i] := false; f := f + 1;\rangle$ } }
**process** Consumer[i = m+1 **to** n] {
    **while**( ... ) {
        # Wait until Full; move to Busy
        $\langle$**await**($f > 0$) $in[i] := true; f := f - 1;\rangle$
        *empty buf*
        # Move to Empty
        $\langle c := c + 1; in[i] := false; e := e + 1;\rangle$ } }
```

An exercise

Note that all we have proved is that $0 \leq p - c \leq 1$. I.e. that the consumers never get ahead of the producers, and that the producers never get more than one message ahead of the consumers. Can you extend the proof to show that no message is ever received more than once and that no message is ever overwritten?

What have we learned?

What have we learned from these two examples?

In both cases we ended up using await statements of only two forms

$$\langle \textbf{await}(s > 0)\, T\ s := s - 1 \rangle$$
$$\langle U\ s := s + 1 \rangle$$

where s is an int variable initialized to a nonnegative integer and where T and U *change only ghost variables*.

It turns out that *all* syncronization problems can be solved using only such statements, thus, we give them a special name and syntax.

A *semaphore* is

a shared integer variable, s, manipulated only by three operations:

declare: sem s := i # $i \geq 0$.Default is $i = 0$.

P (*probeer te verlagen = try to decrease***):**

$$P_T(s) \triangleq \langle \mathbf{await}(s > 0)\ T\ s := s - 1\rangle$$

V (*verhoog* $\simeq$ *increase***):**

$$V_U(s) \triangleq \langle U\ s := s + 1\rangle$$

invariant $s \geq 0$

- s is non-negative
- U and T are sequences of statements that change only ghost variables
- General semaphore: can take on any nonnegative value.
- The implementation of semaphores using await statements is great for reasoning about semaphores. In practive, semaphores are often provided as a language or library primitive and synchronization constructs such as await statements are built using semaphores.

Fairness. V may release a waiting thread. Selection may be

- Weakly fair,
- Strongly fair, or
- FIFO

Derived Inference Rules

$$\frac{(R \wedge g > 0)\ \{T\}\ Q_{g \leftarrow (g-1)}}{\{R\}\ P_T(g)\ \{Q\}}$$

$$\frac{R\ \{U\}\ Q_{g \leftarrow (g+1)}}{\{R\}\ V_U(g)\ \{Q\}}$$

Binary semaphores

Binary Semaphore: either 0 or 1.

- **declare: binsem** s := i # $i \in \{0, 1\}$.Default is $i = 0$.
- Invariant $s \in \{0, 1\}$
- **P (***probeer te verlagen = try to decrease***):**
$$P_T(s) \triangleq \langle \mathbf{await}(s = 1)\ T\ s := 0\rangle$$
- **V (***verhoog* $\simeq$ *increase***):**
$$V_U(s) \triangleq \langle U\ s := 1\rangle$$

For this course we use general semaphores as

- They are only a little bit more complicated
- They are a bit more useful.

Mutual Exclusion

As noted above we use s to count 1 minus the number of threads in their critical section

```
int $s := 1$;
## Global Inv: $s \in \{0, 1\}$

process CS[$i = 1$ to $n$] {
        while ($true$) {
                noncrit. section
                < await($s > 0$)
                        $s -= 1$;
                > ## $s = 0$
                critical section
                < $s += 1$; > }
}
```

```
sem $s := 1$;
## Global Inv: $s \in \{0, 1\}$

process CS[$i = 1$ to $n$] {
        while ($true$) {
                noncrit. section
                $P(s)$ ;
                critical section
                $V(s)$ ; }
}
```

Semaphores and Rights

Often we can think about semaphores as repositories for **rights** (or **permissions**, if you prefer)
—much as a bank account is a repository for money.

- The value of the semaphore is the number of rights it holds.
- A thread can request to acquire a right from a semaphore by calling P.
- A right flows from the semaphore to the thread *when the call to P completes.*
- A thread can send a right to a semaphore by calling: V.

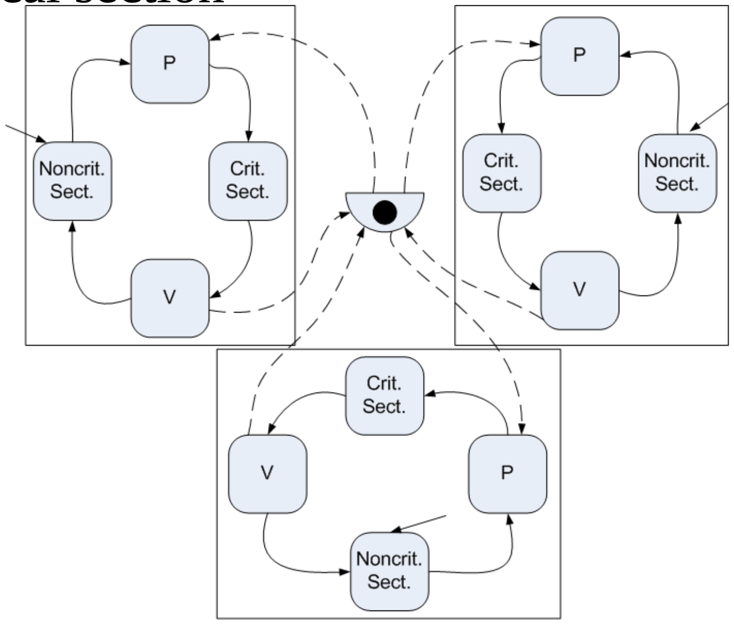
Mutual exclusion

In this case, there is one right, which is the right to enter a critical section.

Initially the semaphore holds the right.

Each thread obtains the right (P) before entering its critical section and at the same time deprives the other threads of the oportunity to obtain the right until it has been released back to the semaphore (V).

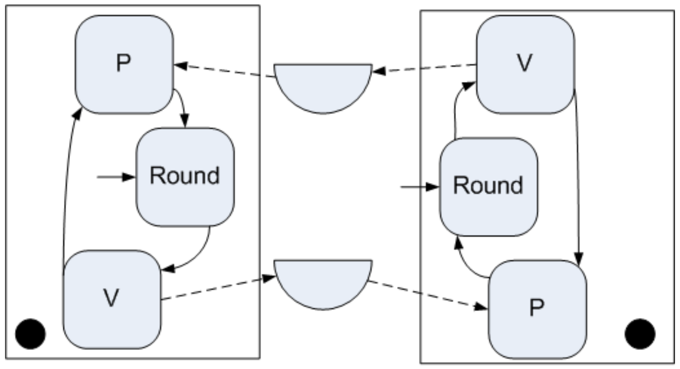
We can see this in a picture. The black dot is the right to execute a critical section



Mutual Exclusion (3 threads)

2 thread barrier synchronization

Initially each thread holds the right to execute its next round.



Two thread barrier

2 thread barrier synchronization

Two thread case:

```
sem arrive[2] := [0,0] ;
# int s[2] := [0,0], c[2] := [0,0] ;
## $arrive[i] = 1 \Rightarrow c[i] \geq c[1-i]$, for all $i \in \{0,1\}$
```

```
process Worker0 {
        while ($true$) {
                ## $c[0] = \min(c)$
                ++s[0];
                round
                ++c[0];
                V(arrive[0]);
                P(arrive[1]);}
}
```

```
process Worker1 {
        while ($true$) {
                ## $c[1] = \min(c)$
                ++s[1];
                round
                ++c[1];
                V(arrive[1]);
                P(arrive[0]);}
}
```

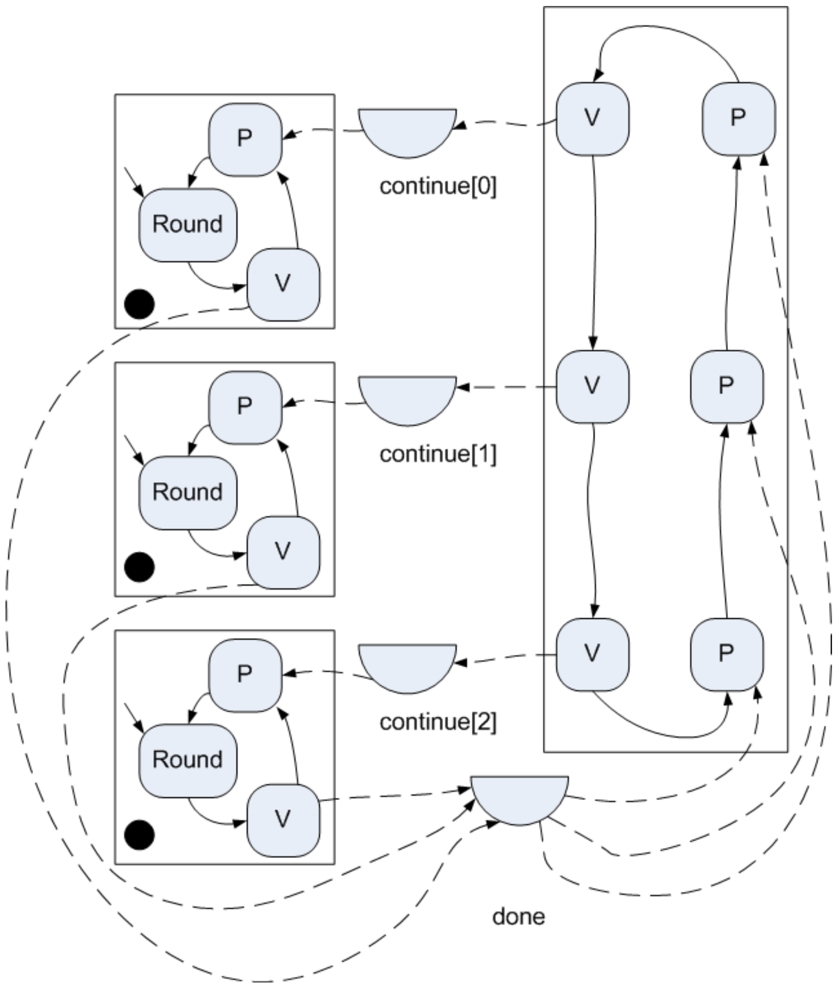
Can be extended to n threads by appropriate choice of semaphores.

Barrier Synchronization: Coordinator

- Workers
  - ∗ Signal arrival with $V(done)$
  - ∗ Wait on $P(continue[i])$
- Coordinator
  - ∗ Waits on n of $P(done)$
  - ∗ Releases all workers with $V(continue[i])$

```
**sem** done := 0 ;
**sem** continue[n] := ([n] 0) ;
**process** Worker[i = 0 **to** n-1] {
        **while** ($true$) {
                *round i*;
                V(done);
                P(continue[i]); } }
**process** Coordinator {
        **while** ($true$) {
                **for** [i = 0 **to** n-1] P(done);
                **for** [i = 0 **to** n-1] V(continue[i]); } }
```

Rights point of view. There are n rights to continue to the next round.



Split Binary Semaphore

- Use semaphores to signal data state rather than thread state.
- *split binary semaphore* — two or more binary semaphores that have the property that at most one is 1 at any time.
- Initially only one is 1.
- Invariant $0 \leq s_0 + s_1 + ... \leq 1$
- In **every** execution path, a P operation on one semaphore is followed (eventually) by a V on a (possibly different) semaphore.
- Code between P and V executed in mutual exclusion.

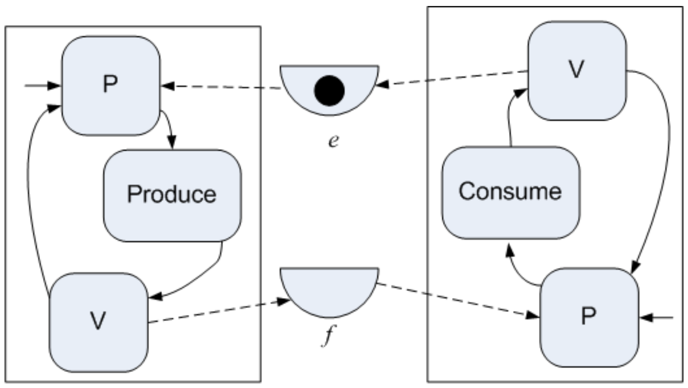
Example: Mutual exclusion on a buffer

Producer and consumer farms

```
thing $buf$;
int $e$ $:= 1$;
int $f$ $:= 0$;
## Global Inv: $0 \le e + f \le 1$

process Producer[i = 1 to m] {
    while(... ) {
        # Calculate the next value
        # Wait until Empty; move to Busy
        P($e$)
        fill $buf$
        # Move to Full
        V($f$)} }
process Consumer[i = m+1 to n] {
    while( ... ) {
        # Wait until Full; move to Busy
        P($f$)
        use $buf$
        # Move to Empty
        V($e$) } }
```

Rights point of view. There is one right, which is the right to use the buffer.



Producer & Consumer

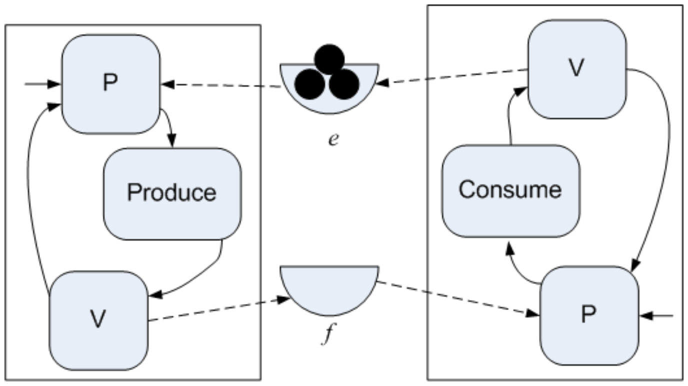
Semaphores as Counters

System with N (identical) resources that are to be shared.

- Use semaphore to represent number available,
- P to obtain one right to use a resource,
- V to release one right to use a resource,.

Consider producer-consumer with bounded buffer of size N and multiple producers and consumers.

Here there are N rights, representing the right to use the buffer.



Producer & Consumer

We also need 1 right representing the right to access the front of the buffer and one right representing the right to use the rear of the buffer.

Producer and Consumer

```
T buf[N];
int front = 0; # next cell to read
int rear = 0; # next cell to write
sem empty = N; # Num. empty cells
sem full = 0; # Num. full cells
sem mutexA = 1;
sem mutexF = 1;
```

```
void Add(int x) {
        P(empty);
        P(mutexA);
        buf[rear] := x;
        rear := (rear+1) % N;
        V(mutexA);
        V(full); }
```

```
T Fetch() {
        P(full);
        P(mutexF);
        T result := buf[front];
        front := (front+1) % N;
        V(mutexF);
        V(empty);
        return result; }
```

Implementing arbitrary synchronization with semaphores

Next we look at a technique that lets you implement any set of await statements using only (binary) semaphores and a few int variables.

This technique is called **passing the baton.**

I'll present the technique by example. The example is ...

The Readers and Writers Problem

- Several threads share a database,
- `Readers` — several can access concurrently.
- `Writers` — must have exclusive access.

Suppose that nr counts the number of readers and nw the number of writers.

Desired invariant

$$nw = 0 \vee (nw = 1 \wedge nr = 0)$$

Two solution forms:

1. Mutual exclusion — use semaphore for lock and count the readers.

  - ∗ First reader in acquires lock, last reader out releases it.
  - ∗ Writer acquires lock and releases when it’s done.

2. Conditional synchronization — Passing the Baton

Coarse-grained solution

```
int nw := 0 ;
int nr := 0 ;
## Global Inv: nw == 0 or nw == 1 and nr == 0

process Reader[i = 1 to M] {
        while ($true$) {
                ...
                $<$ await( nw == 0 ) nr := nr + 1 ; $>$
                read database
                $<$nr := nr - 1;$>$ } }

process Writer[j = 1 to N] {
        while ($true$) {
                ...
                $<$await( nr==0 and nw==0 ) nw := nw + 1; $>$
                write database
                $<$ nw := nw + 1 ; $>$ } }
```

Passing the Baton

A technique to implement a set of general **await** statements using (split binary) semaphores.

As with the producer and consumer farm example, there is one right, this is the right to run code inside the await statements. This is the baton.

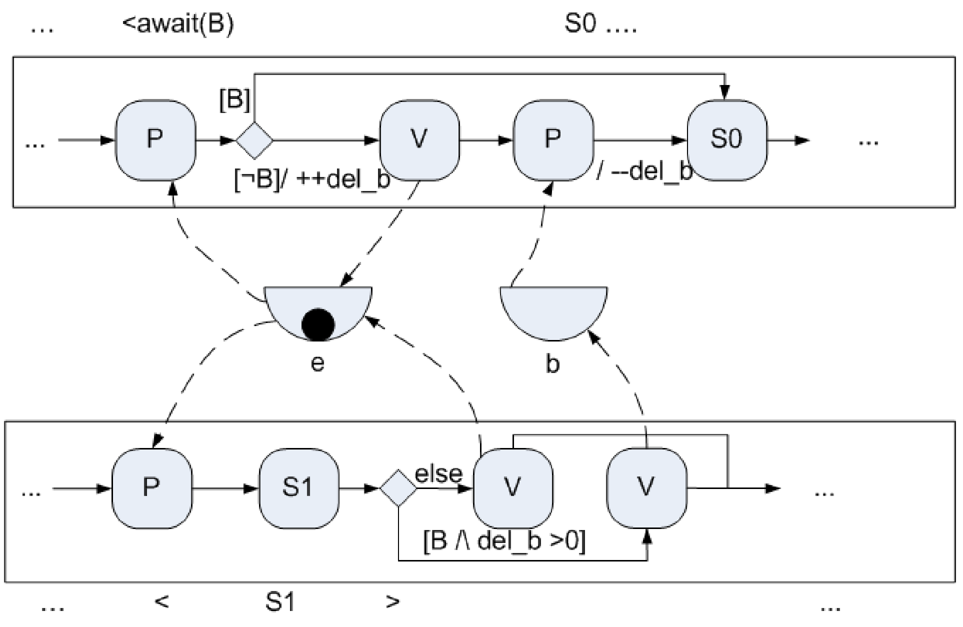
(The rights to read or write the database are represented by the program counters, not by semaphores. I.e a reader or writer has the right to read or write the data base if it has passed its first await statement, but not its second.)

- **sem** e := 1; — Control entry to atomic statements.
- For each condition (guard), B:
  - **sem** b := 0 ; — to delay threads that do **await**(B)
  - A counter **int** del_b := 0— counts the number of delayed threads.
- When a thread enters a critical section (an **await**) it obtains mutual exclusion.
  - If it needs to delay, it gives up exclusive access and waits on b.
- When a thread finishes a critical section (an **await**) it checks to see if there is a condition B that is now true and a

thread waiting on b.

∗ If so it wakes up a thread waiting on b but does not give up mutual exclusion. (This passes the baton). Thus the thread that was waiting on b has the right to execute. No other process can execute critical code in the mean-time, thus B will remain true. It is up to that thread to either give up exclusive access, or pass the baton again.

∗ If not, it gives up exclusive access.



Passing the baton, assuming one conditional critical section (top) and one unconditional critical section (bottom).
The ending of the conditional critical section (not shown) is the same as the end of the unconditional critical section.

Reader-Writer baton passing solution

Global data:

```
int nr := 0, nw := 0 ;
## Global Inv: nw == 0 or nw == 1 and nr == 0
sem e := 1; # exclusive access
sem r := 0; # the right to read
sem w := 0; # the right to write
int del_r := 0; # count of delayed readers
int del_w := 0; # count of delayed writers
```

At the close of each critical section, the following code is executed. It either passes the baton, or gives up exclusive access.

SIGNAL:

```
if (nw == 0 and del_r > 0) {
        V(r) ; } # Pass to a reader
else if (nr == 0 and nw == 0 and del_w > 0) {
        V(w) ; } # Pass to a writer
else { V(e) ; } # Release entry lock
```

```
process Reader[i = 1 to M] {
    while ($true$) { ...
        # $<$ await( nw == 0 ) nr := nr + 1 ; $>$
        P(e); if (nw != 0) {
                del_r++; V(e); P(r); del_w--; }
        nr := nr + 1;
        SIGNAL;
        read database
        # $<$nr := nr - 1;$>$
        P(e); nr := nr - 1; SIGNAL; } }
process Writer[j = 1 to N] {
    while ($true$) { ...
        # $<$await( nr==0 and nw==0 ) nw := nw + 1; $>$
        P(e); if (! (nr==0 and nw==0) ) {
                del_w++; V(e); P(w); del_w--;}
        nw := nw + 1;
        SIGNAL;
        write database
        #$<$nw := nw - 1;$>$
        P(e); nw := nw - 1; SIGNAL; } }
```