# Barrier synchronization

Often, processes executing together to solve a problem must wait at specific steps in order to proceed. For example, if several processes are performing matrix multiplication, they must wait until all have completed in order to proceed to the next step in the computations.

The idea is to provide a function, called wait, which will allow the processes to continue only once all have terminated their task.

Solution 1:

```
final int N="the number of processes needed to synchronize"
Semaphore mutex=1, block=0;
int count=0;

void wait() {

   P(mutex);
   count++;
   if (count == N) {
      count--;
      while (count > 0){V(block); count --; }
      V(mutex);
   }
   else {
      V(mutex);
      P(block);
   }
}
```

Solution 2:

```
final int N="the number of processes needed to synchronize"
Semaphore mutex=1, block=0;
int count=0;

void wait() {

   P(mutex);
   count++;
   if (count == N) {
      count--;
      V(block);
   }
   else {
      V(mutex);
      P(block);
      count--;
      if (count > 0) V(block);
      else V(mutex);
   }
}
```

## Readers and Writers – a simple first solution

```
sem mutexR= 1, rw=1;     // semaphores for mutual exclusion and blocking writers respectively
int nr = 0, nw=0;        // number of active readers and writers, respectively
```

```
process readers ((int r=0; r <= NR; r++)) {
   while (true) {
      ...
      // want to read DB
      P(mutexR);
      nr++;
      if (nr == 1) P(rw);
      V(mutexR);
      read;
      // done reading
      P(mutexR);
      nr--;
      if (nr == 0) V(rw);
      V(mutexR);
   }
}
```

```
process writers ((int w=0; w <= NW; w++)) {
   while (true) {
      ...
      // want to write data
      P(rw);
      write;
      // done writing
      V(rw);
   }
}
```

Fair solution ?

In this solution of the Readers and Writers problem, we keep track of how many processes are waiting to either read or write. Using this state information, we can then freely decide which type of processes to favor, either the readers, the writers, or none at all (therefore achieving a "fair" solution).

Below is an example where the readers are favored:

```
int nr= 0, nw= 0;          // number of active readers and writers, respectively
sem mutex= 1, r= 0, w= 0; // semaphores for mutual exclusion, blocking readers, blocking writers respectively
int dr= 0, dw= 0;          // number of delayed readers and writers, respectively
```

```
process readers ((int r=0; r <= NR; r++)) {
   while(true) {
      ...
      P(mutex);
      if (nw > 0) {
         dr++; V(mutex); P(r);
      }
      nr++;
      if (dr > 0) {
         dr--; V(r);
      }
      else if (dr == 0){
         V(mutex);
      }
      read the database
      P(mutex);
      nr--;
      if (nr == 0 && dw > 0) {
         dw--; V(w);
      }
      else if (nr > 0 || dw == 0) {
         V(mutex);
      }
   }
}
```

```
process writers ((int w=0; w <= NW; w++)) {
   while (true) {
      ...
      P(mutex);
      if (nr > 0 || nw > 0) {
         dw ++; V(mutex); P(w);
      }
      nw ++;
      V(mutex);
      write the database
      P(mutex);
      nw--;
      if (dr > 0) {
         dr--; V(r);
      }
      else if (dw > 0) {
         dw--; V(w);
      }
      else if (dr == 0 && dw == 0) {
         V(mutex);
      }
   }
}
```

To ensure preference for writers, need to delay new readers if a writer is waiting and awaken a delayed reader only if no writer is waiting. **In bold the changes to the code presented above.**

```
int nr= 0, nw= 0;          // number of active readers and writers, respectively
sem mutex= 1, r= 0, w= 0; // mutual exclusion, blocking readers, blocking writers respectively
int dr= 0, dw= 0;          // number of delayed readers and writers, respectively
```

```
process readers ((int r=0; r <= NR; r++)) {
   while(true) {
      ...
      P(mutex);
      if (nw > 0 || dw > 0) {
         dr++; V(mutex); P(r);
      }
      nr++;
      if (dr > 0) {
         dr--; V(r);
      }
      else if (dr == 0){
         V(mutex);
      }
      read the database
      P(mutex);
      nr --;
      if (nr == 0 && dw > 0) {
         dw--; V(w);
      }
      else if (nr > 0 || dw == 0) {
         V(mutex);
      }
   }
}
```

```
process writers ((int w=0; w <= NW; w++)) {
   while (true) {
      ...
      P(mutex);
      if (nr > 0 || nw > 0) {
         dw ++; V(mutex); P(w);
      }
      nw ++;
      V(mutex);
      write the database
      P(mutex);
      nw--;
      if (dr > 0 && dw == 0) {
         dr--; V(r);
      }
      else if (dw > 0) {
         dw--; V(w);
      }
      else if (dr == 0 && dw == 0) {
         V(mutex);
      }
   }
}
```

To make it fair (assuming semaphores themselves are fair) - we could force readers and writers to alternate. In particular, when a writer finishes, all waiting readers get a turn; and when the readers finish, one waiting writer gets a turn.

```
int nr= 0, nw= 0;          // number of active readers and writers, respectively
sem mutex= 1, r= 0, w= 0;  // mutual exclusion, blocking readers, blocking writers respectively
int dr= 0, dw= 0;          // number of delayed readers and writers, respectively
```

```
process readers ((int r=0; r <= NR; r++)) {
   while(true) {
      ...
      P(mutex);
      if (nw > 0 || dw > 0) {
         dr++; V(mutex); P(r);
      }
      nr++;
      if (dr > 0) {
         dr --; V(r);
      }
      else if (dr == 0){
         V(mutex);
      }
      read the database
      P(mutex);
      nr --;
      if (nr == 0 && dw > 0) {
         dw--; V(w);
      }
      else if (nr > 0 || dw == 0) {
         V(mutex);
      }
   }
}
```

```
process writers ((int w=0; w <= NW; w++)) {
   while (true) {
      ...
      P(mutex);
      if (nr > 0 || nw > 0) {
         dw ++; V(mutex); P(w);
      }
      nw ++;
      V(mutex);
      write the database
      P(mutex);
      nw--;
      if (dr > 0) {
         dr--; V(r);
      }
      else if (dw > 0) {
         dw--; V(w);
      }
      else if (dr == 0 && dw == 0) {
         V(mutex);
      }
   }
}
```