# Semaphores

We see that solving the critical section problem is difficult when only using regular variables. And we were trying to synchronize just two processes ! Imagine trying to synchronize three or more !

One additional problem is that the processes that are waiting to enter their critical section are using CPU cycles that otherwise could be better spent elsewhere.

We need better tools to help synchronizing. One of the first tools to help us, and one of the most important, is the semaphore.

A semaphore is essentially a counter. Two operations are allowed, one to increment the counter, and another to decrement the counter. These two operations are traditionally called `P` and `V`.

The added feature is that decrementing the counter is not allowed if the counter is zero. In this case, the process attempting to decrement the counter is blocked, and does not proceed until the counter is incremented (by some other process).

The definition of these `P` and `V` operations are:

```
P(s):  < await(s > 0) { s--; } >
V(s): < s++;  >
```

Note that both these operations are atomic; the `V` operation increments the semaphore `s` in a single CPU cycle (as seen from the "outside"); the `P` operation tests the value of the semaphore `s`, and only if it is positive will `s` be decremented. If `s` is zero, the process stays blocked. Again, the testing and decrementing of `s` is done atomically (as seen from the "outside").

Solving the critical section problem then becomes easy:

```
Semaphore mutex = 1;   // mutex for mutual exclusion


private static process p1 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      P(mutex);
      // critical section
      x += 3;
      // exit protocol
      V(mutex);
   }
}
```

```
private static process p2 {
   for (int s = 1; s <= S; s++) {
      // non-critical section
      ...
      // entry protocol
      P(mutex);
      // critical section
      x += 3;
      // exit protocol
      V(mutex);
   }
}
```

Now that we have such a wonderful tool we can use it for many more applications ! In particular, a semaphore is often used to count resources, and to make processes wait if a particular resource is temporarily unavailable.

An illustration of this concept can be found in the Producers – Consumers problem.

# Producers – Consumers problem

In this problem, we have a process which produces messages of some type. On the other side, we have a process which takes this message, and "consumes it", meaning that the message is read and further processes.

Since a producer and a consumer do not generally execute at the same speeds, we generally provide the space for several messages, usually called a message buffer. The data structure is often called a bounded buffer. *Nota Bene: here we assume that the data is copied to the buffer – not the reference to the data.*

**With only 1 producer/consumer -** notice how the semaphores **do not** provide mutual exclusion !

```
T buf[N];               // the message buffer
int front= 0, rear= 0;  // pointers to empty/full spots in buffer
sem empty= N, full= 0;  // semaphores for synchronisation

process Producer {
   T message;
   while (true) {
      produce(message); // a new message
      // deposit
      P(empty);
      buf[rear] = message;
      rear = (rear+1) % N;
      V(full);
   }
}
```

```
process Consumer {
   T message;
   while (true) {
      //fetch:
      P(full);
      message= buf[front];
      front = (front+1) % N;
      V(empty);
      consume(message);
   }
```

What needs to be changed when the number of consumers and/or producers is greater that 1 ?

# Semaphores in JR

(See chapter 6 JR book)

Declaring a semaphore:

```
sem sem_id = non-negative-int; // 0 by default, but
                                    // better to be explicit
```

Using a semaphore:

```
V(sem_id);              // as seen above, increment
P(sem_id);              // decrement or block
```

Arrays of semaphores: this is a little tricky in JR, because of the way semaphores are implemented. You'll need to declare a special kind of array, and then initialize each semaphore (the reason for this awkward syntax will be seen next semester).

Arrays of semaphores:

```
cap void () sem_array[] = new cap void() [SIZE];
for (int i=0; i < SIZE; ++i) {
      sem_array[i] = new sem; // or new sem(initial_value)
}
```

Some semaphore implementations have other operations available, but <u>all</u> have a form of **`P`** or **`V`**.