

Computation structures

Support for problem-solving lesson #6

Exercise 1

Three producer processes (P_1 , P_2 and P_3) write some integers into a buffer of N slots (`int x = produce()`) in such a way that a consumer process C can read them (`consume(Buffer[i])`).

The producers must access the buffer in an orderly fashion : P_1 , then P_2 , then P_3 , then P_1 an so on.

Once the buffer is full, the consumer reads the produced numbers and empties the buffer.

Give a simple solution to this problem by using semaphores and a shared memory zone containing only the buffer and a counter of the number of elements in it.

Exercise 1

Things to pay attention to:

- The producers cannot produce any value if the buffer is full.
- The consumer cannot consume any value if the buffer is *not* full.
- Each producer must wait its turn before producing a value.
- We must ensure that no deadlock (nor livelock) will ever happen.

Exercise 1

Let's first write the code for features, without any synchronization control

```
//The producers (id = 0,1 or 2)
prod(int id) {
    while(true) {

        buffer[i] = produce();
        i = i+1;
        if(i == N) {

            i = 0;
        }
    }
}

shared int i = 0;
shared int buffer[N];

//The consumer
cons() {
    while(true) {

        for(int k = 0; k < N; ++k) {
            consume(buffer[k]);
        }
    }
}
```

Exercise 1

Each producer must wait its turn before producing a value.

```
shared semaphore p[3] = {1,0,0};
```

```
shared int i = 0;  
shared int buffer[N];
```

```
//The producers (id = 0,1 or 2)  
prod(int id) {  
    while(true) {  
        wait(p[id]);  
        buffer[i] = produce();  
        i = i+1;  
        if(i == N) {  
            i = 0;  
        }  
        signal(p[(id+1)%3]);  
    }  
}
```

```
//The consumer  
cons() {  
    while(true) {  
        for(int k = 0; k < N; ++k) {  
            consume(buffer[k]);  
        }  
    }  
}
```

Exercise 1

The consumer cannot consume any value if the buffer is *not* full.

```
shared semaphore p[3] = {1,0,0};  
shared semaphore full = 0;
```

```
shared int i = 0;  
shared int buffer[N];
```

```
//The producers (id = 0,1 or 2)  
prod(int id) {  
    while(true) {  
        wait(p[id]);  
        buffer[i] = produce();  
        i = i+1;  
        if(i == N) {  
            signal(full);  
  
            i = 0;  
        }  
        signal(p[(id+1)%3]);  
    }  
}
```

```
//The consumer  
cons() {  
    while(true) {  
        wait(full);  
        for(int k = 0; k < N; ++k) {  
            consume(buffer[k]);  
        }  
    }  
}
```

Exercise 1

The producers cannot produce any value if the buffer is full.

```
shared semaphore p[3] = {1,0,0};
shared semaphore full = 0;
shared semaphore clear = 0;
shared int i = 0;
shared int buffer[N];
```

```
//The producers (id = 0,1 or 2)
prod(int id) {
    while(true) {
        wait(p[id]);
        buffer[i] = produce();
        i = i+1;
        if(i == N) {
            signal(full);
            wait(clear);
            i = 0;
        }
        signal(p[(id+1)%3]);
    }
}
```

```
//The consumer
cons() {
    while(true) {
        wait(full);
        for(int k = 0; k < N; ++k) {
            consume(buffer[k]);
        }
        signal(clear);
    }
}
```

Exercise 1

Is **i** protected enough?

Yes, because only 1 semaphore in **p** will be equal to 1 at a time

```
shared semaphore p[3] = {1,0,0};
shared semaphore full = 0;
shared semaphore clear = 0;
shared int i = 0;
shared int buffer[N];
```

```
//The producers (id = 0,1 or 2)
prod(int id) {
    while(true) {
        wait(p[id]);
        buffer[i] = produce();
        i = i+1;
        if(i == N) {
            signal(full);
            wait(clear);
            i = 0;
        }
        signal(p[(id+1)%3]);
    }
}
```

```
//The consumer
cons() {
    while(true) {
        wait(full);
        for(int k = 0; k < N; ++k) {
            consume(buffer[k]);
        }
        signal(clear);
    }
}
```


Exercise 1

Could we improve even further?

Yes, because we strictly respect the statement by waiting for the consumer to empty the buffer.

We could let the producers produce more values when the consumer has started to read.

//The producers (id = 0,1 or 2)

```
prod(int id) {  
    while(true) {  
        wait(p[id]);  
        buffer[id] = produce();  
        buffer[i] = produce();  
        if(i == N) {  
            if(i == signal(full));  
            signal(full);  
            i = 0;  
        }  
        signal(p[(id+1)%3]);  
    }  
}
```

```
shared semaphore p[3] = {1,0,0};  
shared semaphore full = 0;  
shared semaphore clear = 0;  
shared int i = 0; ←  
shared int buffer[N];
```

But then *i* will no longer represent the number of unread elements in the buffer at each time.

//The consumer

```
cons() {  
    while(true) {  
        wait(full);  
        for(int k = 0; k < N; ++k) {  
            consume(buffer[k]);  
        }  
        signal(clear);  
        signal(clear);  
    }  
}
```

Exercise 1

Can I switch these two waits?

No, because the producers must wait their turn before taking a job to perform

//The producers (id = 0,1 or 2)

```
prod(int id) {  
    while(true) {  
        wait(p[id]);  
        wait(clear);  
        buffer[i] = produce();  
        i = i+1;  
        if(i == N) {  
            signal(full);  
            i = 0;  
        }  
        signal(p[(id+1)%3]);  
    }  
}
```

```
shared semaphore p[3] = {1,0,0};  
shared semaphore full = 0;  
shared semaphore clear = N;  
shared int i = 0;  
shared int buffer[N];
```

//The consumer

```
cons() {  
    while(true) {  
        wait(full);  
        for(int k = 0; k < N; ++k) {  
            consume(buffer[k]);  
            signal(clear);  
        }  
    }  
}
```

Exercise 2

One producer P and two consumers C_1 and C_2 want to communicate through a buffer mechanism.

A buffer of N slots is thus placed between the producer P and consumer C_1 while a second buffer, of M slots, is placed between the producer P and consumer C_2 .

Each consumer reads the elements in its buffer, except when the buffer is empty.

The producer writes each elements into either the first or the second buffer and must be blocked only when the two buffers are full.

Give a simple solution to this problem. You can use `int x = rand(2);` to obtain a random integer in $[0,1]$;

Exercise 2

Things to pay attention to:

- The producer cannot produce any value if *both* buffers are full.
 - But if one buffer is full and the other one has room inside, the producer must use the one with available space.
- Each consumer cannot consume any value if its own buffer is empty.
- We must ensure that no deadlock (nor livelock) will ever happen.

Exercise 2

What you shouldn't do:

```
shared int buffer1[M]; shared int buffer2[N];
shared semaphore s[2] = {M,N};
shared semaphore go[2] = {0,0};
```

```
void producer() {
    int in[2] = {0,0};
    while(true) {
        int x = rand(2);
        wait(s[x]);
        int val = produce();
        if(x == 0) { buffer1[in[x]] = val; }
        else {buffer2[in[x]] = val;}
        in[x]++;
        signal(go[x]);
    }
}
```

Why shouldn't you do that?

Worst case scenario :

- **buffer1** is full but **buffer2** still has room.
- The call to **rand(2)** returns 0.
- You will wait on **s[0]**
(thus, be blocked while the two buffers were not full).

Or, the dual case (**buffer2** is full, **buffer1** is not and **rand(2)** returned 1).

Exercise 2

When should the producer be blocked?

- The producer cannot produce any value if *both* buffers are full.
- So, how many elements can the producer produce at most if no consumer ever got the chance to progress?
 - $M+N$ (the sizes of the two buffers)
- How to transcribe this using semaphores?

```
shared semaphore s = M+N;  
producer() {  
    wait(s);  
    ...  
}
```

Exercise 2

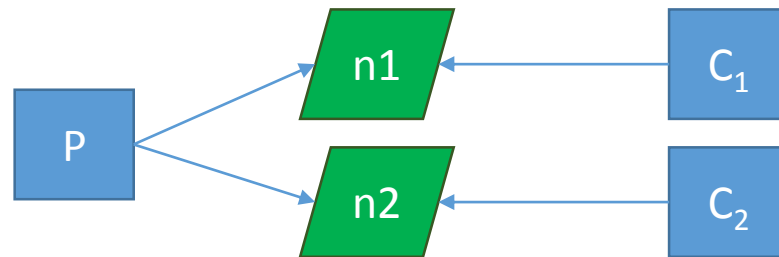
How do I know which buffer to use?

- Semaphores?
 - No, because I would have the same problem
- I must put the number of available slots of each buffer in shared memory (e.g. $n1$ and $n2$).
- I then test these numbers. If $n1 == 0$, I use buffer 2. If $n2 == 0$, I use buffer 1. If $n1 > 0$ **AND** $n2 > 0$, I select one buffer at random.
- What if $n1 == 0$ **AND** $n2 == 0$?
 - In that case, I am still waiting on the semaphore.
- I then don't forget to update either $n1$ or $n2$.

Exercise 2

Protecting **n1** and **n2**.

- Should I protect these shared variables?
 - Yes, because several processes will manipulate them (possibly at the same time).
- What processes are accessing these variables?



- So, 1 mutex to protect both **n1** and **n2**?
 - No, it's too restrictive. E.g. C_2 could not access **n2** while C_1 is accessing **n1**, although they could update these variables simultaneously with no interference.
 - 1 mutex for **n1** **AND** 1 mutex for **n2**.

Exercise 2

As usual, let's start by the code logic, without any synchronization.

```
//The producer
prod() {
    int out[2] = {0,0};
    while(true) {

        if(n[0] == 0) {
            produceIn(buffer2,&out[1],1, M);
        } else if (n[1] == 0) {
            produceIn(buffer,&out[0],0, N);
        } else {
            int x = rand(2);
            if(x == 1) {
                produceIn(buffer2,&out[1],1, M);
            } else {
                produceIn(buffer,&out[0],0, N);
            }
        }
    }
}

shared int n[2] = {N,M};
shared int buffer[N];
shared int buffer2[M];

//The production function
produceIn(int buffer[ ], int* out,
        int type, int max) {
    int x = generate();
    buffer[*out] = x;
    *out = ((*out)+1)%max;

    n[type] = n[type]-1;
}

//The consumer (id = 0 or 1)
//max = N or M
cons(int buffer[ ], int id, int max) {
    int in = 0;
    while(true) {
        consume(buffer[in]);

        n[id] = n[id]+1;

        in = (in+1)%max;
    }
}
```

Exercise 2

The producer cannot produce any value if both buffers are full.

```
shared semaphore s = M+N;
```

```
shared int n[2] = {N,M};  
shared int buffer[N];  
shared int buffer2[M];
```

```
//The producer
```

```
prod() {  
    int out[2] = {0,0};  
    while(true) {  
        wait(s);  
        if(n[0] == 0) {  
            produceIn(buffer2,&out[1],1, M);  
        } else if (n[1] == 0) {  
            produceIn(buffer,&out[0],0, N);  
        } else {  
            int x = rand(2);  
            if(x == 1) {  
                produceIn(buffer2,&out[1],1, M);  
            } else {  
                produceIn(buffer,&out[0],0, N);  
            }  
        }  
    }  
}
```

```
//The production function
```

```
produceIn(int buffer[ ], int* out,  
        int type, int max) {  
    int x = generate();  
    buffer[*out] = x;  
    *out = ((*out)+1)%max;  
  
    n[type] = n[type]-1;  
}
```

```
//The consumer (id = 0 or 1)
```

```
//max = N or M
```

```
cons(int buffer[ ], int id, int max) {  
    int in = 0;  
    while(true) {  
        consume(buffer[in]);  
  
        n[id] = n[id]+1;  
  
        signal(s);  
        in = (in+1)%max;  
    }  
}
```

Exercise 2

Each consumer cannot consume any value if its own buffer is empty.

```
shared semaphore s = M+N;

shared semaphore ready[2] = {0,0};
shared int n[2] = {N,M};
shared int buffer[N];
shared int buffer2[M];

//The producer
prod() {
    int out[2] = {0,0};
    while(true) {
        wait(s);
        if(n[0] == 0) {
            produceIn(buffer2,&out[1],1, M);
        } else if (n[1] == 0) {
            produceIn(buffer,&out[0],0, N);
        } else {
            int x = rand(2);
            if(x == 1) {
                produceIn(buffer2,&out[1],1, M);
            } else {
                produceIn(buffer,&out[0],0, N);
            }
        }
    }
}

//The production function
produceIn(int buffer[ ], int* out,
        int type, int max) {
    int x = generate();
    buffer[*out] = x;
    *out = ((*out)+1)%max;

    n[type] = n[type]-1;

    signal(ready[type]);
}

//The consumer (id = 0 or 1)
//max = N or M
cons(int buffer[ ], int id, int max) {
    int in = 0;
    while(true) {
        wait(ready[id]);
        consume(buffer[in]);

        n[id] = n[id]+1;

        signal(s);
        in = (in+1)%max;
    }
}
```

Exercise 2

The number of available slots needs to be protected.

```
shared semaphore s = M+N;
shared semaphore sn[2] = {1,1};
shared semaphore ready[2] = {0,0};
shared int n[2] = {N,M};
shared int buffer[N];
shared int buffer2[M];
```

//The producer

```
prod() {
    int out[2] = {0,0};
    while(true) {
        wait(s);
        if(n[0] == 0) {
            produceIn(buffer2,&out[1],1, M);
        } else if (n[1] == 0) {
            produceIn(buffer,&out[0],0, N);
        } else {
            int x = rand(2);
            if(x == 1) {
                produceIn(buffer2,&out[1],1, M);
            } else {
                produceIn(buffer,&out[0],0, N);
            }
        }
    }
}
```

//The production function

```
produceIn(int buffer[ ], int* out,
        int type, int max) {
    int x = generate();
    buffer[*out] = x;
    *out = ((*out)+1)%max;
    wait(sn[type]);
    n[type] = n[type]-1;
    signal(sn[type]);
    signal(ready[type]);
}
```

//The consumer (id = 0 or 1)

//max = N or M

```
cons(int buffer[ ], int id, int max) {
    int in = 0;
    while(true) {
        wait(ready[id]);
        consume(buffer[in]);
        wait(sn[id]);
        n[id] = n[id]+1;
        signal(sn[id]);
        signal(s);
        in = (in+1)%max;
    }
}
```

Exercise 2

Why don't we protect the access to `n[]` here?

Because it's only a *read* operation, and the worse that could happen would be that a consumer increases one of these values.

```
shared semaphore s = M+N;
shared semaphore sn[2] = {1,1};
shared semaphore ready[2] = {0,0};
shared int n[2] = {N,M};
shared int buffer[N];
shared int buffer2[M];
```

//The producer

```
prod() {
    int out[2] = {0,0};
    while(true) {
        wait(s);
        if(n[0] == 0) {
            produceIn(buffer2,&out[1],1, M);
        } else if (n[1] == 0) {
            produceIn(buffer,&out[0],0, N);
        } else {
            int x = rand(2);
            if(x == 1) {
                produceIn(buffer2,&out[1],1, M);
            } else {
                produceIn(buffer,&out[0],0, N);
            }
        }
    }
}
```

//The production function

```
produceIn(int buffer[ ], int* out,
         int type, int max) {
    int x = generate();
    buffer[*out] = x;
    *out = ((*out)+1)%max;
    wait(sn[type]);
    n[type] = n[type]-1;
    signal(sn[type]);
    signal(ready[type]);
}
```

//The consumer (id = 0 or 1)

//max = N or M

```
cons(int buffer[ ], int id, int max) {
    int in = 0;
    while(true) {
        wait(ready[id]);
        consume(buffer[in]);
        wait(sn[id]);
        n[id] = n[id]+1;
        signal(sn[id]);
        signal(s);
        in = (in+1)%max;
    }
}
```

Exercise 2

Why don't we protect the access to the buffers?

For the same reason that we don't need `out[]` and `in` to be in shared memory. A slot in a buffer is only accessed/updated by one process at a time.

```
shared semaphore s = M+N;
shared semaphore sn[2] = {1,1};
shared semaphore ready[2] = {0,0};
shared int n[2] = {N,M};
shared int buffer[N];
shared int buffer2[M];
```

//The producer

```
prod() {
  int out[2] = {0,0};
  while(true) {
    wait(s);
    if(n[0] == 0) {
      produceIn(buffer2,&out[1],1, M);
    } else if (n[1] == 0) {
      produceIn(buffer,&out[0],0, N);
    } else {
      int x = rand(2);
      if(x == 1) {
        produceIn(buffer2,&out[1],1, M);
      } else {
        produceIn(buffer,&out[0],0, N);
      }
    }
  }
}
```

//The production function

```
produceIn(int buffer[ ], int* out,
         int type, int max) {
  int x = generate();
  buffer[*out] = x;
  *out = ((*out)+1)%max;
  wait(sn[type]);
  n[type] = n[type]-1;
  signal(sn[type]);
  signal(ready[type]);
}
```

//The consumer (id = 0 or 1)

//max = N or M

```
cons(int buffer[ ], int id, int max) {
  int in = 0;
  while(true) {
    wait(ready[id]);
    consume(buffer[in]);
    wait(sn[id]);
    n[id] = n[id]+1;
    signal(sn[id]);
    signal(s);
    in = (in+1)%max;
  }
}
```

Exercise 2

Shouldn't I use %M or %N in these operations?

No, because `n[]` represents the number of available slots in the buffer, and that value is between `[0,M]` (or `[0,N]`).

```
shared semaphore s = M+N;
shared semaphore sn[2] = {1,1};
shared semaphore ready[2] = {0,0};
shared int n[2] = {N,M};
shared int buffer[N];
shared int buffer2[M];
```

//The producer

```
prod() {
    int out[2] = {0,0};
    while(true) {
        wait(s);
        if(n[0] == 0) {
            produceIn(buffer2,&out[1],1, M);
        } else if (n[1] == 0) {
            produceIn(buffer,&out[0],0, N);
        } else {
            int x = rand(2);
            if(x == 1) {
                produceIn(buffer2,&out[1],1, M);
            } else {
                produceIn(buffer,&out[0],0, N);
            }
        }
    }
}
```

//The production function

```
produceIn(int buffer[ ], int* out,
         int type, int max) {
    int x = generate();
    buffer[*out] = x;
    *out = ((*out)+1)%max;
    wait(sn[type]);
    n[type] = n[type]-1;
    signal(sn[type]);
    signal(ready[type]);
}
```

//The consumer (id = 0 or 1)

//max = N or M

```
cons(int buffer[ ], int id, int max) {
    int in = 0;
    while(true) {
        wait(ready[id]);
        consume(buffer[in]);
        wait(sn[id]);
        n[id] = n[id]+1;
        signal(sn[id]);
        signal(s);
        in = (in+1)%max;
    }
}
```

Exercise 2

Should I still protect this operation with semaphores if I used `n[type]--` (or `n[id]++`)?

Yes, because `++` and `--` are not atomic operations. They require both a read and a write and the process could be paused between the read and the write.

```
shared semaphore s = M+N;
shared semaphore sn[2] = {1,1};
shared semaphore ready[2] = {0,0};
shared int n[2] = {N,M};
shared int buffer[N];
shared int buffer2[M];
```

//The producer

```
prod() {
    int out[2] = {0,0};
    while(true) {
        wait(s);
        if(n[0] == 0) {
            produceIn(buffer2,&out[1],1, M);
        } else if (n[1] == 0) {
            produceIn(buffer,&out[0],0, N);
        } else {
            int x = rand(2);
            if(x == 1) {
                produceIn(buffer2,&out[1],1, M);
            } else {
                produceIn(buffer,&out[0],0, N);
            }
        }
    }
}
```

//The production function

```
produceIn(int buffer[ ], int* out,
        int type, int max) {
    int x = generate();
    buffer[*out] = x;
    *out = ((*out)+1)%max;
    wait(sn[type]);
    n[type] = n[type]-1;
    signal(sn[type]);
    signal(ready[type]);
}
```

//The consumer (id = 0 or 1)

```
//max = N or M
cons(int buffer[ ], int id, int max) {
    int in = 0;
    while(true) {
        wait(ready[id]);
        consume(buffer[in]);
        wait(sn[id]);
        n[id] = n[id]+1;
        signal(sn[id]);
        signal(s);
        in = (in+1)%max;
    }
}
```


Exercise 3

The following C code offers a solution to the problem of the shared buffer zone between several producers and several consumers (each element is read by only one consumer) :

```
shared int in = 0, out = 0;  
shared int buf[N];  
shared semaphore n = 0, e = N, s = 1;
```

//The producer (called in a while loop)

```
append(int v) {  
    wait(e);  
    wait(s);  
    buf[in] = v;  
    in = (in + 1) % N;  
    signal(s);  
    signal(n);  
}
```

//The consumer (called in a while loop)

```
int take() {  
    int v;  
    wait(s);  
    wait(n);  
    v = buf[out];  
    out = (out + 1) % N;  
    signal(e);  
    signal(s);  
    return v;  
}
```

- (a) Is this solution acceptable ? Explain.
- (b) If not, what code modifications are required ? Justify.
- (c) Considering that there are K producers and L consumers, modify this program in such a way that each consumer is able to read, at its own pace, all the produced elements.


Exercise 3

(a) Is this solution acceptable ? Explain

```
shared int in = 0, out = 0;  
shared int buf[N];  
shared semaphore n = 0, e = N, s = 1;
```

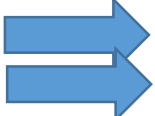
//The producer (called in a while loop)

```
append(int v) {  
    wait(e);  
    wait(s);  
    buf[in] = v;  
    in = (in + 1) % N;  
    signal(s);  
    signal(n);  
}
```



//The consumer (called in a while loop)

```
int take() {  
    int v;  
    wait(s);  
    wait(n);  
    v = buf[out];  
    out = (out + 1) % N;  
    signal(e);  
    signal(s);  
    return v;  
}
```



- This solution is not acceptable.
- Worst case scenario:
 - Consumer gets the hand and executes `wait(s)`, succeeds, and now `s` equals `0`
 - Consumer executes `wait(n)` and is **blocked**, because `n` equals `0`
 - Producer gets the hand and executes `wait(e)`, succeeds, and now `e` equals `N-1`
 - Producer executes `wait(s)` and is **blocked**, because `s` equals `0`
 - Producer and consumer are **both blocked**, this is a deadlock situation.

Exercise 3

(b) If not, what code modifications are required? Justify.

```
shared int in = 0, out = 0;  
shared int buf[N];  
shared semaphore n = 0, e = N, s = 1;
```

//The producer (called in a while loop)

```
append(int v) {  
    wait(e);  
    wait(s);  
    buf[in] = v;  
    in = (in + 1) % N;  
    signal(s);  
    signal(n);  
}
```

//The consumer (called in a while loop)

```
int take() {  
    int v;  
    wait(s);  
    wait(n);  
    v = buf[out];  
    out = (out + 1) % N;  
    signal(e);  
    signal(s);  
    return v;  
}
```



- The problem comes from the fact that the consumer tries to access the mutex before checking that there's work to do.
- So, a very simple solution would be to switch the two waits in the consumer.
- Now, the consumer first waits for jobs to be done before entering the mutual exclusion section, and no deadlock can happen.
- Switching these two signals could also be a good idea.

Exercise 3

(c) Considering that there are K producers and L consumers, modify this program in such a way that each consumer is able to read, at its own pace, all the produced elements.

```
shared int in = 0, out = 0;  
shared int buf[N];  
shared semaphore n = 0, e = N, s = 1;
```

//The producer (called in a while loop)

```
append(int v) {  
    wait(e);  
    wait(s);  
    buf[in] = v;  
    in = (in + 1) % N;  
    signal(s);  
    signal(n);  
}
```

//The consumer (called in a while loop)

```
int take() {  
    int v;  
    wait(n);  
    wait(s);  
    v = buf[out];  
    out = (out + 1) % N;  
    signal(s);  
    signal(e);  
    return v;  
}
```

If there are K producers, should we change something?

No, because each producer will indeed write one element at a time into the buffer.

Exercise 3

Considering L consumers, with each consumer able to read, at its own pace, all the produced elements.

```
shared int in = 0, out = 0;
shared int buf[N];
shared semaphore n = 0, e = N, s = 1;
```

```
//The producer (called in a while loop)
append(int v) {
    wait(e);
    wait(s);
    buf[in] = v;
    in = (in + 1) % N;
    signal(s);
    signal(n);
}
```

```
//The consumer (called in a while loop)
int take() {
    int v;
    wait(n);
    wait(s);
    v = buf[out];
    out = (out + 1) % N;
    signal(s);
    signal(e);
    return v;
}
```

The problem is not really about the L consumers, but more that each consumer must read each produced element.

Thus, I need:

- A counter, per element, that counts how many consumers must still read that element;
- A semaphore per consumer, that is used to unlock it when there's something new to read;
- An array of **out** values, one for each consumer, since each consumer can read at its own pace.

Exercise 3

What I have so far

```
shared int in = 0, out = 0;  
shared int buf[N];  
shared semaphore n = 0, e = N, s = 1;
```

```
//The producer (called in a while loop)  
append(int v) {  
    wait(e);  
    wait(s);  
    buf[in] = v;  
  
    in = (in + 1) % N;  
    signal(s);  
  
    signal(n);  
}
```

```
//The consumer (called in a while loop)  
int take(int pid) {  
    int v;  
    wait(n);  
    wait(s);  
    v = buf[out];  
  
    signal(e);  
  
    out = (out + 1) % N;  
    signal(s);  
    return v;  
}
```

Exercise 3

I need a counter, per element, that counts how many consumers must still read that element;

```
shared int in = 0, out = 0;  
shared int buf[N], read[N] = {L,L,..., L};  
shared semaphore n = 0, e = N, s = 1;
```

```
//The producer (called in a while loop)  
append(int v) {  
    wait(e);  
    wait(s);  
    buf[in] = v;  
    read[in] = L;  
    in = (in + 1) % N;  
    signal(s);  
  
    signal(n);  
}
```

```
//The consumer (called in a while loop)  
int take(int pid) {  
    int v;  
    wait(n);  
    wait(s);  
    v = buf[out];  
    read[out]--;  
    if(read[out] == 0) {  
        signal(e);  
    }  
    out = (out + 1) % N;  
    signal(s);  
    return v;  
}
```

Exercise 3

I need a semaphore per consumer, that is used to unlock it when there's something new to read;

```
shared int in = 0, out = 0;  
shared int buf[N], read[N] = {L,L,..., L};  
shared semaphore n[L] = {0,0,...,0}, e = N, s = 1;
```

//The producer (called in a while loop)

```
append(int v) {  
    wait(e);  
    wait(s);  
    buf[in] = v;  
    read[in] = L;  
    in = (in + 1) % N;  
    signal(s);  
    for(int i = 0; i < L; i++) {  
        signal(n[i]);  
    }  
}
```

//The consumer (called in a while loop)

```
int take(int pid) {  
    int v;  
    wait(n[pid]);  
    wait(s);  
    v = buf[out];  
    read[out]--;  
    if(read[out] == 0) {  
        signal(e);  
    }  
    out = (out + 1) % N;  
    signal(s);  
    return v;  
}
```


Exercise 3

I need an array of **out** values, one for each consumer, since each consumer can read at its own pace.

```
shared int in = 0, out[L] = {0,0,...,0};
shared int buf[N], read[N] = {L,L,..., L};
shared semaphore n[L] = {0,0,...,0}, e = N, s = 1;
```

//The producer (called in a while loop)

```
append(int v) {
    wait(e);
    wait(s);
    buf[in] = v;
    read[in] = L;
    in = (in + 1) % N;
    signal(s);
    for(int i = 0; i < L; i++) {
        signal(n[i]);
    }
}
```

//The consumer (called in a while loop)

```
int take(int pid) {
    int v;
    wait(n[pid]);
    wait(s);
    v = buf[out[pid]];
    read[out[pid]]--;
    if(read[out[pid]] == 0) {
        signal(e);
    }
    out[pid] = (out[pid] + 1) % N;
    signal(s);
    return v;
}
```