COMP 4735: Operating Systems

Lesson 8.3: Semaphores

Rob Neilson

rneilson@bcit.ca

Semaphores

Cooperating Multiple Processes

- we use the **`fork(proc, N, arg`$_1$`, arg`$_2$`, …,arg`$_N$`)`** command to create a process with the N arguments specified
- two cooperating processes can be defined and created as follows:

```
proc_0() {                          proc_1(int x, int y) {
  while(TRUE) {                       while(TRUE) {
    <compute section>;                  <compute section>;
    <critical section>;                 <critical section>;
  }                                   }
}                                   }

<shared global declarations>
<initial processing>
fork(proc_0, 0);
fork(proc_1, 2, 6, 11);
```

- only one process at a time can be executing in its critical section
- each process also has a compute section, which may be executed concurrently with the other process

What is a Semaphore?

- a *semaphore*, s, is a nonnegative integer variable that can only be changed or tested by these two indivisible functions:

```
P(s): [while(s == 0) {wait}; s = s - 1]
V(s): [s = s + 1]
```

down P(s): accomplishes the “test and set” of a variable in a single indivisible instruction

up V(s): signals any blocked processes to allow execution to continue

- these functions operate in a similar manner to the enter(lock) and exit(lock) functions described earlier, although they offer a more general solution

To Solve a General Problem ...

```
Proc_0() {
  while(TRUE) {
    <compute section>;
    P(s);
      <critical section>;
    V(s);
  }
}
```

```
proc_1() {
  while(TRUE {
    <compute section>;
    P(s);
      <critical section>;
    V(s);
  }
}
```

```
semaphore s = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

Insert `P(s)` immediately before the critical section.

- decrements the variable (s) so that next process has to wait

Insert `V(s)` immediately after the critical section.

- increments the variable (s) so that next process can continue

Solving the Account Balance Problem

```
proc_0() {                          proc_1() {
  . . .                               . . .
  P(s);                               P(s);
    balance += amount;                  balance -= amount;
  V(s);                               V(s);
  . . .                               . . .
}                                   }

semaphore s = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

Remember:

```
P(s): [while(s == 0) {wait}; s = s - 1]
V(s): [s = s + 1]
```

Account Balance Timing ...

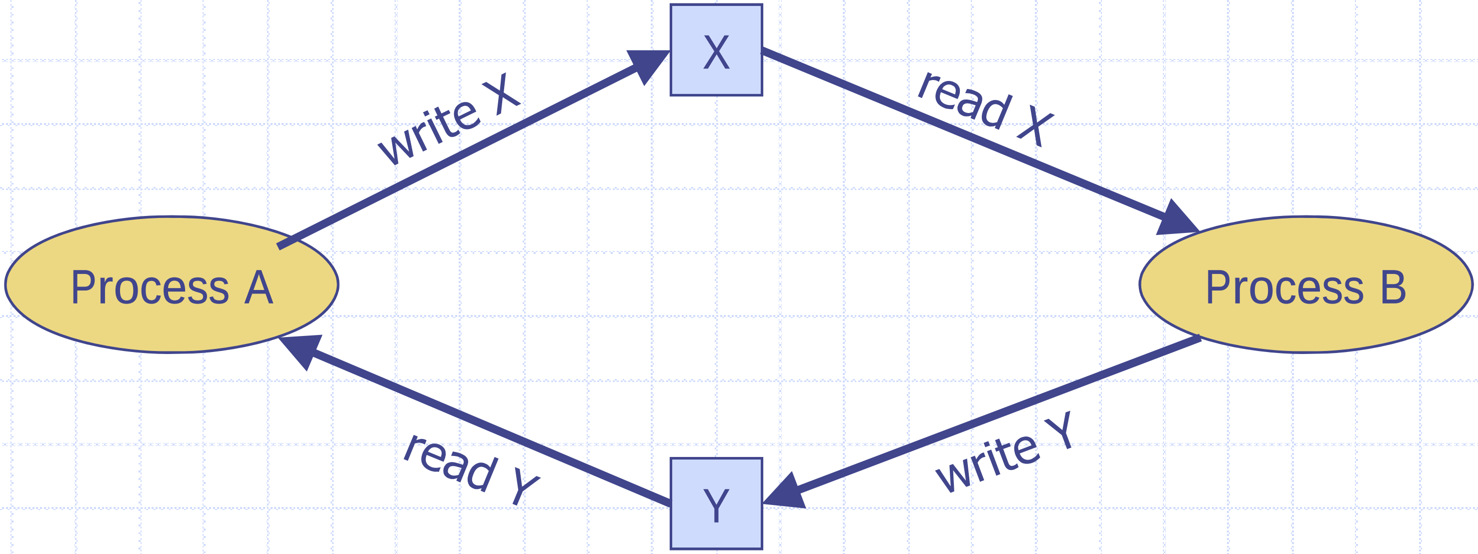
	proc_0	proc_1	
			s: 1
	...		
ts=1	P(s); [while(s == 0) {wait}; s--]		s: 0
ts=2		... P(s); [while(s == 0) {wait}; s--]	|WAIT
ts=3	balance += amount;		
ts=4		P(s);	|WAIT
ts=5	V(s); [s++]		s: 1
ts=6		P(s);	s: 0
		balance -= amount;	
		V(s);	s: 1
	...	...	

Binary Semaphore

- the semaphore variable can be initialized to any non-negative value
- when a semaphore can only have values of 0 or 1, we call it a ***binary semaphore***
  - binary semaphore can be initialized to 0 or 1, depending on problem
    - most critical section problems initialize to 1
    - some synchronization problems initialize to 0 (see next example)
- the previous “critical section synchronization problem” was solved using a binary semaphore
- another general type of problem is the need to synchronize the actions of two threads, for example ….

General Synchronization Problem

- consider the case where one process write a shared variable, and another reads it
- the process that is reading must wait for the data to be placed in the memory, or its read will obtain stale data



- in this problem there is ***no critical section***
- instead, the problem is to 'coordinate the actions' of the processes

Formulating the General Sync Problem

we need this ...

to happen before this ...

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    <compute section A2>;
    retrieve(y);
  }
}

fork(proc_A, 0);
fork(proc_B, 0);
```

```
proc_B() {
  while(TRUE) {
    retrieve(x);
    <compute section B1>;
    update(y);
    <compute section B2>;
  }
}
```

... and we need this ...

... to happen before this ...

General Sync Problem (1)

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    <compute section A2>;
    retrieve(y);
  }
}
```

```
proc_B() {
  while(TRUE) {
    retrieve(x);
    <compute section B1>;
    update(y);
    <compute section B2>;
  }
}
```

```
semaphore s1 = 0; // proc_a will use this to tell proc_b
                  // when it has written X

semaphore s2 = 0; // proc_b will use this to tell proc_a
                  // when it has written Y

fork(proc_A, 0);
fork(proc_B, 0);
```

- The solution will use two binary semaphores.
- The semaphores are initialized to 0 which means:
  - *wait until it is OK to proceed.*
- Binary semaphores that are initialized to 0 are sometimes called **blocking semaphores**

General Sync Problem (2)

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    <compute section A2>;
    retrieve(y);
  }
}

semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);
```

```
proc_B() {
  while(TRUE) {
    P(s1); // wait for proc_a
    retrieve(x);
    <compute section B1>;
    update(y);
    <compute section B2>;
  }
}
```

Step 1:

- make proc_b wait for proc_a to write X
- s1 is initialized to 0, so proc b cannot proceed until explicitly told to
  - ie: it has to wait for proc_a to do V(s1)

General Sync Problem (3)

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    V(s1); // signal proc_b
    <compute section A2>;
    retrieve(y);
  }
}
```

```
proc_B() {
  while(TRUE) {
    P(s1); // wait for proc_a
    retrieve(x);
    <compute section B1>;
    update(y);
    <compute section B2>;
  }
}
```

```
semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);
```

Step 2:

- make proc_a signal proc_b after it writes X
- V(s1) increments the semaphore to 1
- this allows proc_b to proceed the next time it gets the CPU

General Sync Problem (4)

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    V(s1); // signal proc_b
    <compute section A2>;
    P(s2); // wait for proc_b
    retrieve(y);
  }
}
```

```
proc_B() {
  while(TRUE) {
    P(s1); // wait for proc_a
    retrieve(x);
    <compute section B1>;
    update(y);
    <compute section B2>;
  }
}
```

```
semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);
```

Step 3:

- make proc_a wait for proc_b to write Y
- note that we are using the other semaphore
  - ie: P(s2)

General Sync Problem (5)

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    V(s1); // signal proc_b
    <compute section A2>;
    P(s2); // wait for proc_b
    retrieve(y);
  }
}
```

```
proc_B() {
  while(TRUE) {
    P(s1); // wait for proc_a
    retrieve(x);
    <compute section B1>;
    update(y);
    V(s2); // signal proc_a
    <compute section B2>;
  }
}
```

```
semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);
```

Step 4:

- make proc_b signal proc_a after it writes Y
- V(s2) increments the semaphore to 1
- this allows proc_a to proceed the next time it gets the CPU

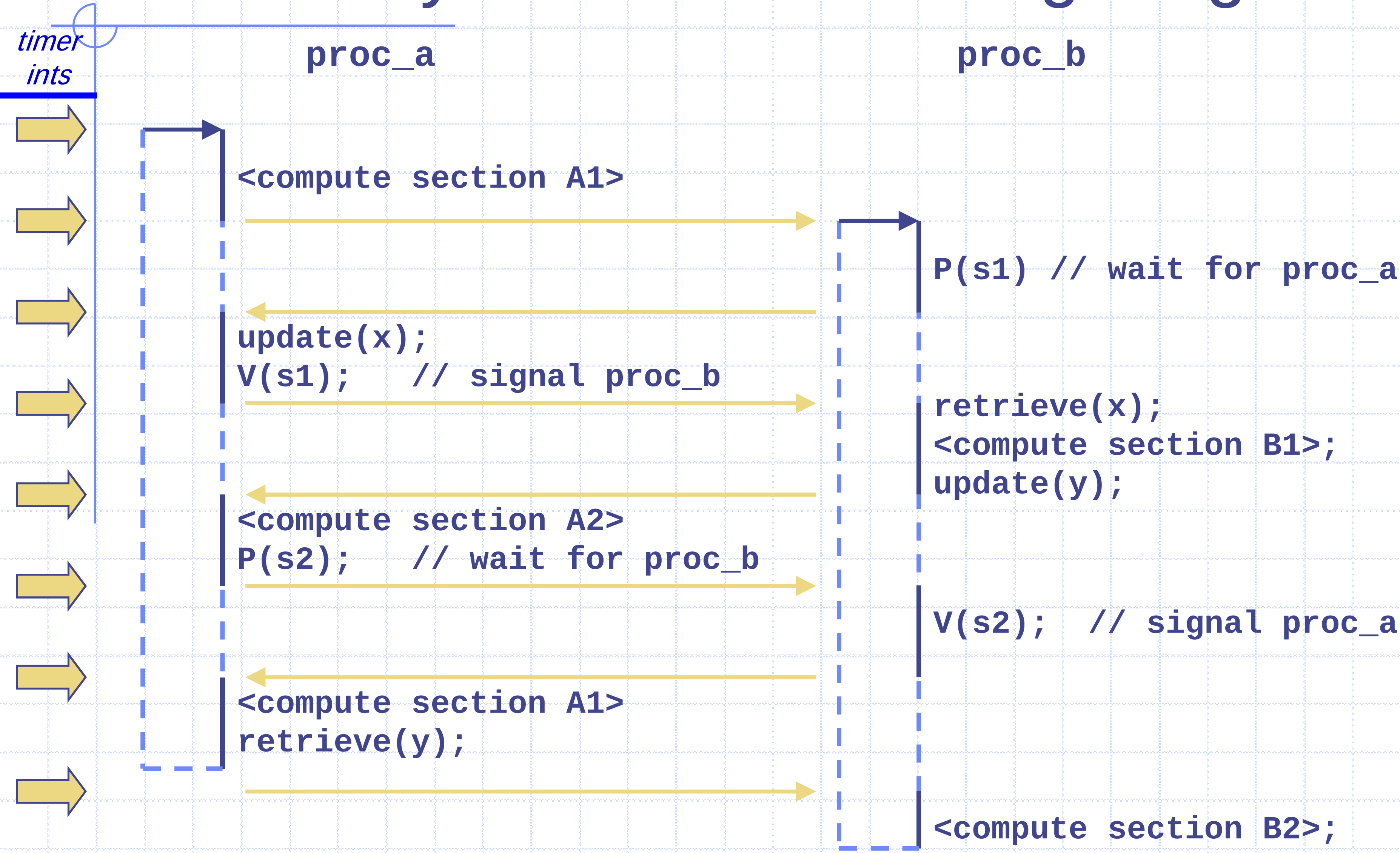
Final Solution to General Problem

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
    V(s1); // signal proc_b
    <compute section A2>;
    P(s2); // wait for proc_b
    retrieve(y);
  }
}
```

```
proc_B() {
  while(TRUE) {
    P(s1); // wait for proc_a
    retrieve(x);
    <compute section B1>;
    update(y);
    V(s2); // signal proc_a
    <compute section B2>;
  }
}
```

```
semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);
```

General Sync Problem: Timing Diag



Counting Semaphore

So far we have considered two types of semaphores:

1. binary semaphore:
   - used to control access to critical sections
2. blocking semaphores:
   - used to make threads wait for events before proceeding

There is a more general type of semaphore, called a ***counting semaphore***, which can take on values other than 0 or 1.

- counting semaphores are useful for situations where you want to limit the number of threads that are in some state, but the limit is greater than 1
  - for example, to access or use a resource

Counting Semaphores: Resource Control

Problem:

- you have a finite amount of some resource
- many threads can acquire and use a resource
- if more threads request resources than there are resources, we want threads to block until a resource becomes available

Solution:

- create a **counting semaphore**, shared among all threads
- **initialize** it to the **number of resources**
- when a thread wants a resource it executes P(s)
- if s is 0 (no resources available) the thread blocks
- if s>0, s is decremented and one of the resources is assigned to the thread
- when a thread is done with the resource it executes V(s) to increment the semaphore (put the resource back in the pool)

Resource Control Solution

```
proc() {
  while(TRUE) {
    <compute section>;
    P(res); // block if no resources available
      <use the resource>
    V(res); // free a resource
  }
}
semaphore res = 5;  // there are 5 resources to share
for (i=1 to 100)
   fork(proc, 0);   // there are 100 processes
```

Classic Semaphore Problems

- there are a number of classic semaphore problems that are worth studying as they provide insight into how semaphores can be used

- these problems include
  - bounded-buffer (aka producer-consumer) problem
  - readers-writers problem
  - dining philosophers problem

- we will consider some of these problems now …

Bounded Buffer Problem Statement

There are two threads that share data using N buffers.

- the buffers exist in and are accessed via shared memory

1. producer thread operates as follows …
   - get an empty buffer
   - write some information into the buffer
   - put the full buffer somewhere that the consumer process can get it

   Note: the producer must block if there are no empty buffers to fill

2. consumer thread operates as follows …
   - get a full buffer
   - extract (and process) the information in it
   - put the empty buffer somewhere that the producer process can get it

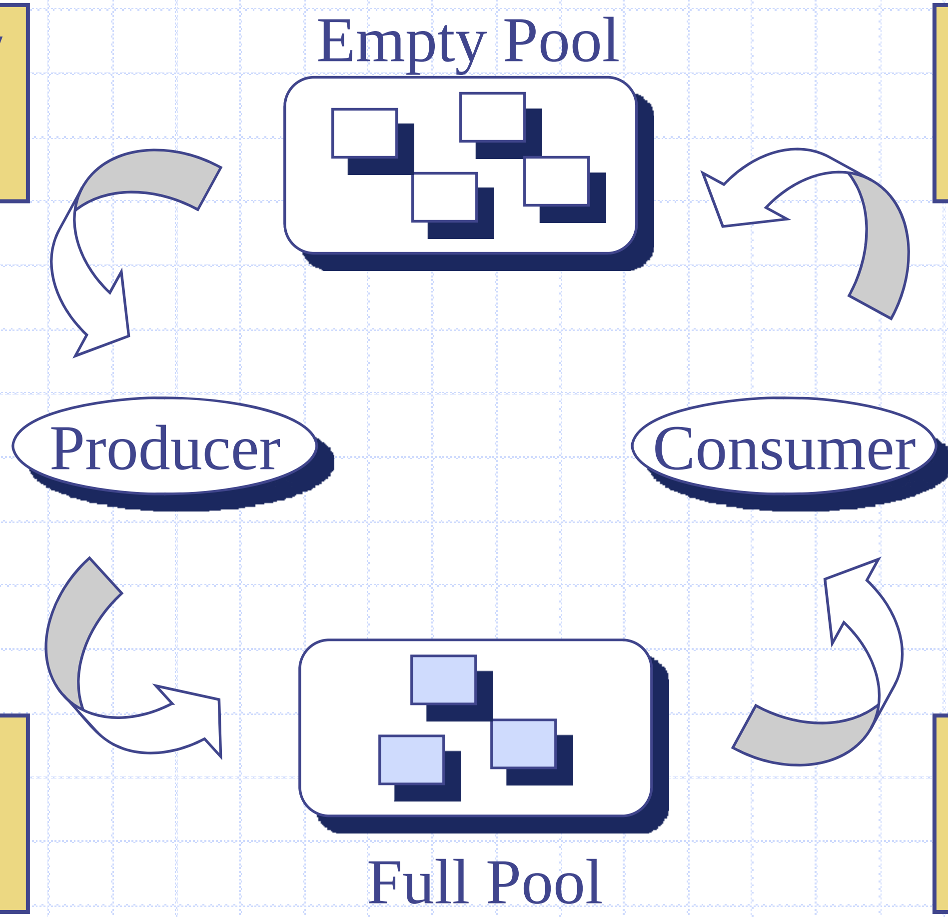
   Note: the consumer must block if there are no full buffers to process

Bounded Buffer Scenario

get an empty buffer from the pool

put the empty buffer back in the pool

fill the buffer with data



process the data in the buffer

put the full buffer in the ***full*** pool

get the next full buffer to process

Solving the Bounded Buffer Problem

To solve the problem, we need to consider all operations that need to be synchronized.

- ie: where will we use semaphores?

1. producer needs to block if there are no empty buffers to fill
   - we will use a counting semaphore, initialized to the number of empty buffers N
   - each time the producer takes a buffer, the semaphore will be decremented
   - producer blocks if semaphore is 0 when it tries to take a buffer
2. consumer needs to block if there are no full buffers to process
   - we will use a counting semaphore, initialized to the number of full buffers 0
   - each time the consumer gets a full buffer, the semaphore will be decremented
   - consumer blocks if semaphore is 0 when it tries to get a buffer to process
3. both threads need to enter critical sections when they update the contents of the buffer pools
   - we need to prevent race conditions, as the buffer pools are shared variables
   - we will use a binary semaphore to ensure that only one thread updates the contents of the buffer pool at a time

Bounded Buffer Problem - Producer

```
producer() {
  buf_type *next, *here;          // next holds the data to write
  while(TRUE) {
    produce_item(next);           // create some data for consumer
    P(empty);                     // get empty buff; block if none
      P(s);                          enter critical section
        here = obtain(empty);          remove buff from empty pool
      V(s);                          exit the critical section
      copy_buffer(next, here);    // copy 'next' into the new buffer
      P(s);                          enter critical section
        release(here, fullPool);       add buffer to full pool
      V(s);                          exit critical section
    V(full);                      // signal consumer that there is...
  }                               // ... data to process
}
semaphore s = 1;                  // binary semaphore for mutual excl
semaphore full = 0;               // counting semaphore for full pool
semaphore empty = N;              // counting semaphore for empty pool
buf_type buffer[N];
fork(producer, 0);
fork(consumer, 0);
```

Bounded Buffer Problem - Consumer

```
consumer() {
  buf_type *next, *here;
  while(TRUE) {
    P(full);                       // get full buff; block if none
      P(s);                           enter critical section
        here = obtain(full);            get buff from full pool
      V(s);                           exit the critical section
      copy_buffer(here, next);     // empty the buffer (to local buff)
      P(s);                           enter critical section
        release(here, emptyPool);       put buffer back in empty pool
      V(s);                           exit critical section
    V(empty);                      // signal producer (buffs available)
    consumeItem(next);             // use the data for whatever ...
  }                                // ... purpose it was intended
}
semaphore s = 1;
semaphore full = 0;
semaphore empty = N;
buf_type buffer[N];
fork(producer, 0);
fork(consumer, 0);
```

Bounded Buffer Problem (final soln)

```
producer() {
  buf_type *next, *here;
  while(TRUE) {
    produce_item(next);
    P(empty);
      P(s);
        here = obtain(empty);
      V(s);
      copy_buffer(next, here);
      P(s);
        release(here, fullPool);
      V(s);
    V(full);
  }
}
semaphore s = 1;
semaphore full = 0;
semaphore empty = N;
buf_type buffer[N];
fork(producer, 0);
fork(consumer, 0);
```

```
consumer() {
  buf_type *next, *here;
  while(TRUE) {
    P(full);
      P(s);
        here = obtain(full);
      V(s);
      copy_buffer(here, next);
      P(s);
        release(here, emptyPool);
      V(s);
    V(empty);
    consumeItem(next);
  }
}
```

green lines are not protected; just general processing

blue lines are counting/synchronising buffer use

red lines are protected critical sections

The End