50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Processes, Synchronisations, and Deadlock

Natalie Agus

1 Inter-processes communication

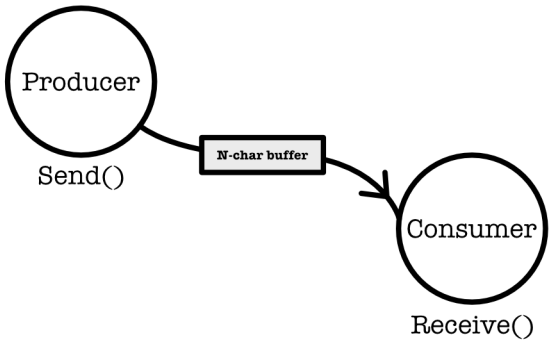

Figure 1

There are many processes that can run in the computer and inter-processes communication is a common task. Challenges that arise from inter-processes communication are concurrency, asynchrony, and signalling exercise. As shown in Figure 1, we can see processes as either the producer or consumer, depending on its data flow, and both processes have access to a **shared memory buffer** to pass information to one another.

2 The Precedence Constraint

In order for processes to communicate successfully, there is a need to establish a **precedence constraint**:

1. Consumer cannot consume data **before** it is produced: $send_i \leq rcv_i$
2. Producer cannot overwrite data in the buffer **before** the consumer read it: $rcv_i \leq send_{i+N}$, given a buffer of size N.

The precedence constraint can be fulfilled by **FIFO buffering** and **semaphores**, therefore allowing a smooth communication between producer process and consumer process.

3 FIFO buffering

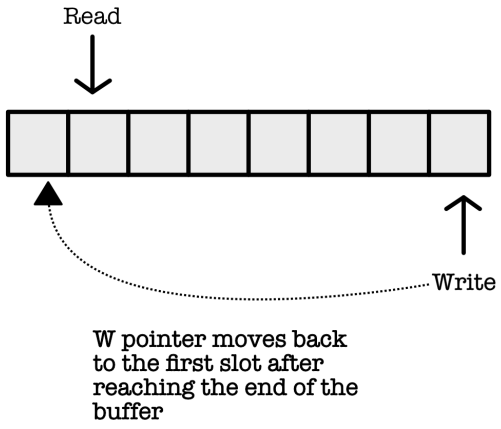

Figure 2

Producer can pass information to the consumer using FIFO buffer, that is implemented as a ring buffer, shown in Figure 2:

- The consumer has Read (R) pointer, that moves in one particular direction (in this case, left to right).
- The producer also has a Write (W) pointer, that also moves in the same direction as the read pointer (left to right), and loops back to the first slot on the left once it has finished writing on the rightmost cell.
- **To fulfil the precedence constraint, especially constraint number 2**, one has to hope that the W pointer does not overlap (catch up) with the R pointer as they move to the right.
- We can fix this using **semaphores**.

4 The Semaphores

In its simplest essence, Semaphore is a shared integer variable that takes on positive value that represents the number of available 'resources'. It tells the programs (producer and consumer both) how many resources are left in the (shared) pool. In this case of producer consumer program, the number of initial semaphores is the number of empty slots in the buffer. The programs can call two functions to "increase" or "decrease" the semaphore if they need to use the resources:

1. Given a Semaphore s of a certain positive initial value,
2. `Wait(s)`
   This function decreases the semaphore by 1, and can be called by a program that want to **use** the resource. **If the current semaphore is 0, wait until there's available semaphore**.
3. `Signal(s)`
   This function increases the semaphore by 1, and can be called by a program that has finished using the resource and want to **return** the resource back to the pool

4.1 Semaphore implementation in the kernel

The code for Semaphore is implemented in the system kernel, meaning that the OS is the only one that can change the semaphores. When a program calls Wait(s) or Signal(s), **it invokes SCV call**, and gives control to the kernel to increase or decrease the semaphore accordingly. What the kernel does is as follows:

1. When a program A calls `Wait(s)`, it got interrupted and switch to kernel mode. The kernel will check if there's available semaphore, if yes, decrement it, and resume the program A. Otherwise, stall the program A (put to sleep) until there's available semaphore (meaning some other program calls `Signal(s)`.
2. When a program A calls `Signal(s)`, it got interrupted and also switch to kernel mode. The kernel will increment the semaphore, and it will wake-up (change the sleep status) on other programs B or C that are waiting on the semaphore, then return to resume the program A.
3. Note: *The scheduler will rotate execution between programs that are awake*, so do not worry about which program will be executed next. The semaphore part of the kernel code only takes care of this semaphore resources that are shared between programs.

The code implementation of semaphore methods as supervisor call is illustrated below[1]:

[1] "Lock" in Figure 3 simply means the semaphore s (the particular resource you'd want to keep track)

Semaphores as Supervisor Call

```
wait_h( )
{
    int *addr;
    addr = User.Regs[R0];   /* get arg */
    if (*addr <= 0) {
      User.Regs[XP] = User.Regs[XP] – 4;
      sleep(addr);
    } else
      *addr = *addr - 1;
}

signal_h( )
{
    int *addr;
    addr = User.Regs[R0];   /* get arg */
    *addr = *addr + 1;
    wakeup(addr);
}
```

Calling sequence:

```
...
|| put address of lock
|| into R0
CMOVE(lock, R0)
SVC(WAIT)
```

SVC call is not interruptible since it is executed in supervisory mode.

Figure 3

4.2 Semantic guarantee of semaphores : Single producer, single consumer

The semaphore guarantees both precedence constraint with the following,

- Create two semaphores: `chars` and `space`,
  - `chars` is a resource consumed by the consumer, and produced by the producer
  - `space` is a resource consumed by the producer, and consumed by the consumer
- The first constraint, $send_i \leq rcv_i$ is fulfilled by letting the producer code to `signal(chars)` and the consumer code to `wait(chars)`
- The second constraint, $rcv_i \leq send_{i+N}$ is fulfilled by letting the producer code to `wait(space)` and the consumer code to `signal(space)`

The basic implementation is as follows:

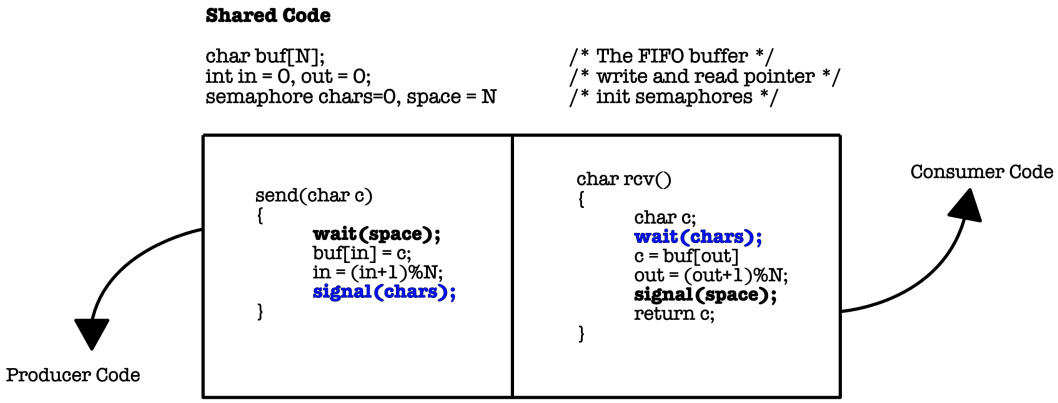

Figure 4

4.3 Semantic guarantee of semaphores : Multiple producer, multiple consumer

Consider the multiple producer scenario in Figure 5:

- We do not want producers to interfere with one another (messing up the write pointer)
- In the case that we have multiple consumer, we do not want it to interfere with each other as well (messing up with the read pointer)
- Add another semaphore that guarantees **mutual exclusion** (mutex), meaning that only one producer can write at a time, and only one consumer can read at a time.

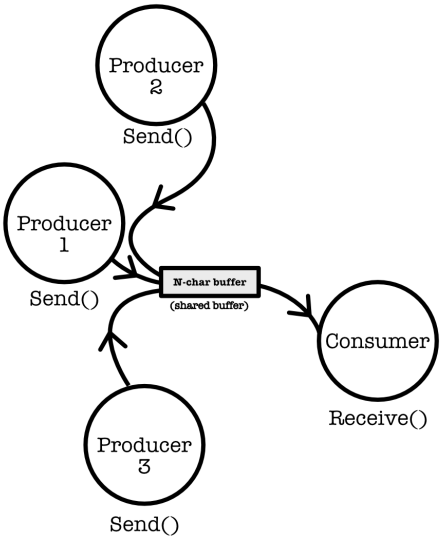

Figure 5

The basic code implementation is as follows:

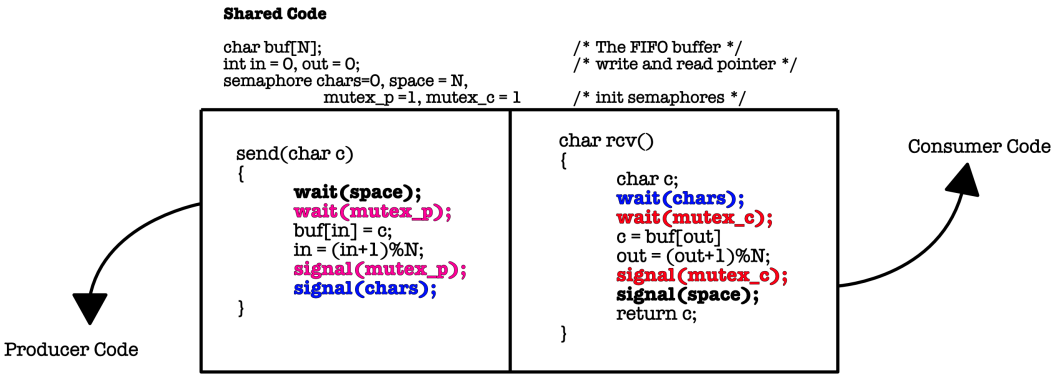

Figure 6