More General Non-Sharable Resources

- The problem of solving mutual exclusion is only one application for a semaphore. Let's consider another problem. Suppose we have a resource which is not entirely mutually exclusive but which nevertheless has to be protected from too many processes/threads accessing it at the same time.

Examples in Computer Systems

- In a computing environment, such a resource might be a set of networked printers that can be used by a spooler to route documents to the first available free printer. It does not matter which printer ultimately prints, as long as each printer only deals with one document at a time.
- Another example might be a windows environment where the window manager can support the creation of say 4 simultaneous windows at a time without running out of memory resources. Processes may wish to create a window to interact with a user.
- Finally consider a networked system where the IT manager has purchased a licence for a software product that covers up to say 100 simultaneous users.

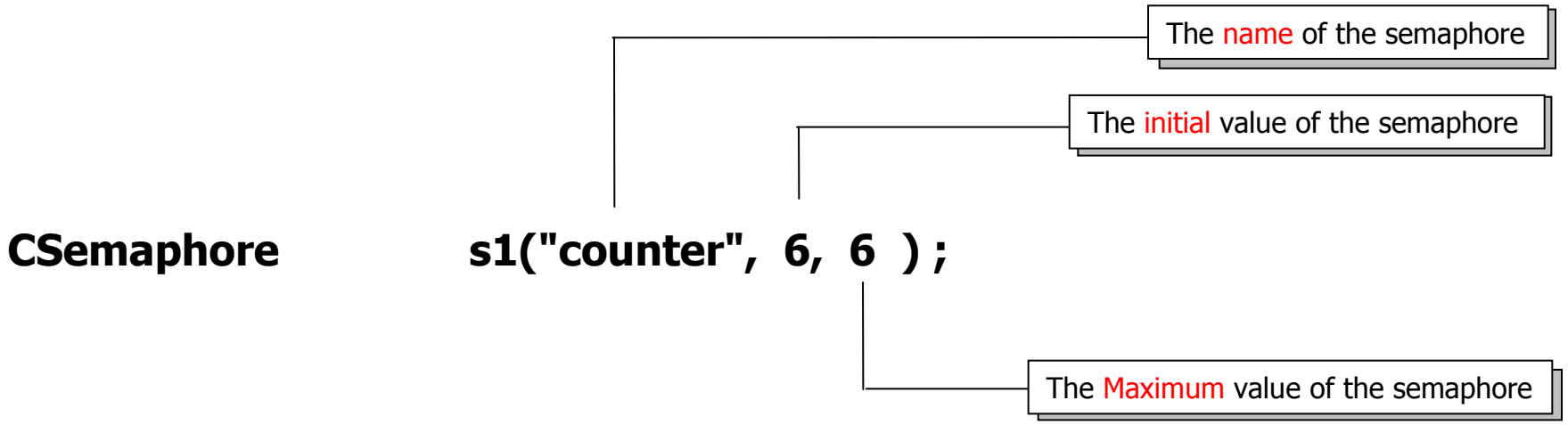
Examples in the Real World.

- In the real world a good example of this is say a post-office or bank with say 6 counters.
- Here, customers wishing to use the post office have to line up and wait for a free counter to become available. Any counter will do since they are all equivalent to each other.
- Now imagine a simulation where people are represented by processes/threads and that the post-office counters represent a finite set of resources which our processes are attempting to gain access to.

Solution using a Counting Semaphore

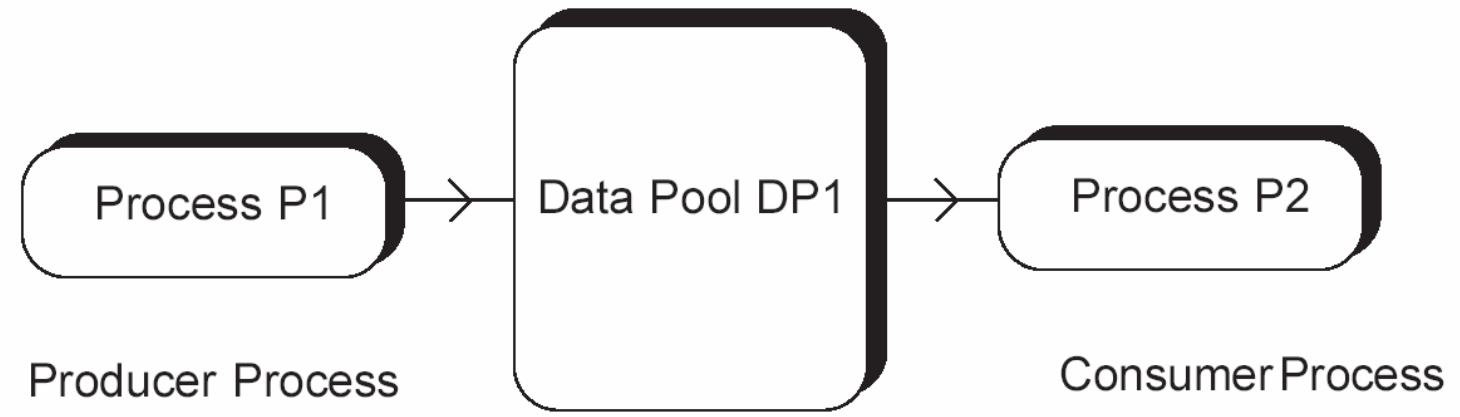
- One solution to this type of problem would be to create a 'general' or counting semaphore to keep track of how many resources are currently free or in use.
- For example, in our post-office, where there are 6 free counters available at opening time (the start of our simulation), we could introduce a counting semaphore with the initial value 6, that is, equal to the number of free resources.
- Now, processes/threads (i.e. people or customers in our post-office simulation) that wish to perform a transaction using one of these counters, will have to wait for a free counter. This could be simulated by getting the process/thread to perform a WAIT() on a single semaphore created to control access to all 6 counters.
- If a counter is free, (indicated by a semaphore value > 0), then the process/thread can use the WAIT() primitive to decrement the semaphore and proceed to use a counter, thus the first 6 customers waiting outside the post-office at opening times will be allowed to gain individual access to a counter.
- The 7^{th} customer waiting outside will get suspended when it performs the WAIT()
- Of course once a customer has performed a transaction at the post office counter, they can release their counter for use by another customer by executing the SIGNAL() primitive.
- This will wake up the next waiting customer who can then proceed to use the vacated counter.

- In summary, the value of the semaphore indicates how many resources are free. If it gets to 0, processes (customers) will have to wait. To create a counting semaphore, only requires that we create a CSemaphore object with the maximum/count value specified, as shown below.
- The one below is created with an initial and maximum value of 6



Classical Concurrent Systems Problems: The Producer - Consumer Problem

- Consider the arrangement of processes P1 and P2 shown below communicating via the data pool DP1.
- P1 is a producer process that generates or produces data stored in the data pool DP1.
- P2 is a consumer process that reads or consumes data from the pool DP1 and acts upon it when ever such data is produced.
- Such an arrangement occurs frequently in many concurrent systems and within operating systems themselves. For example
  - P1 could be a process submitting data to a print spooler (a process represented by p2). The communicating resource in this case could be a file where the document is written, to be read out later by p2 when it can find a free printer to use
  - From a different perspective, p1 could be that print spooler communicating with p2 a device driver using a shared buffer.

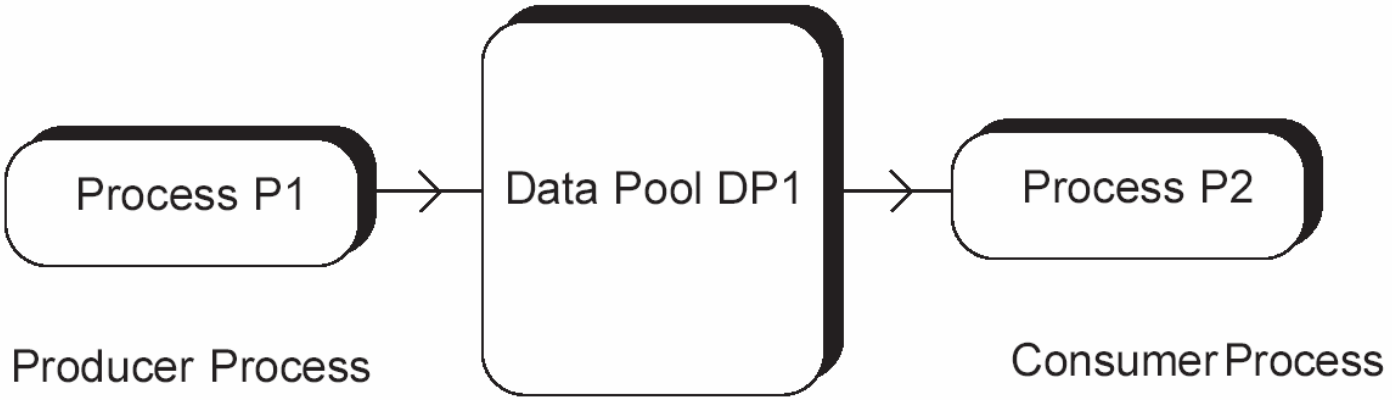


Design Problems

- Because P1 and P2 are independent processes/devices who cannot communicate with each other, other than via the resource represented here by DP1, or synchronise their activities or speeds, there are two obvious problems associated with simply letting them carry on updating and reading the pool in their own time and at their own speed.

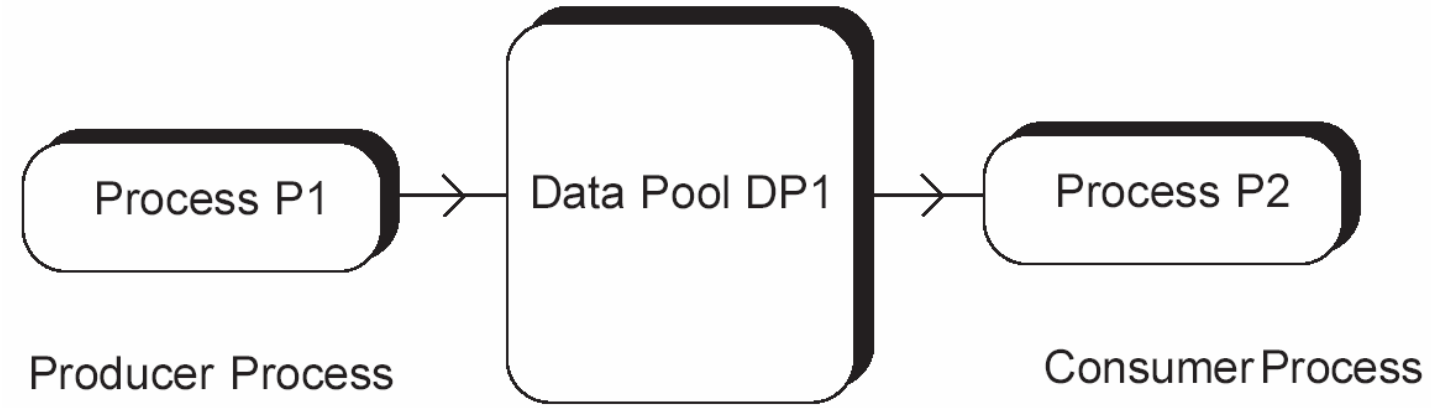
Problem 1 - Data Corruption.

- Process P1 could decide to update the information in the pool DP1 at the same time as Process P2 is reading it out.
- Because the pool is randomly accessible by both processes, there is a risk that P2 could read data out the pool that is made up in part of previous data, and the new data that P1 is attempting to write to the pool. P2 then ends up working with a mixture of out-of-date and up-to-date information leading to incorrect results.
- In essence we have a Mutual Exclusion problem as we have seen before where only the producer or the consumer should be accessing the shared resource at any one time.



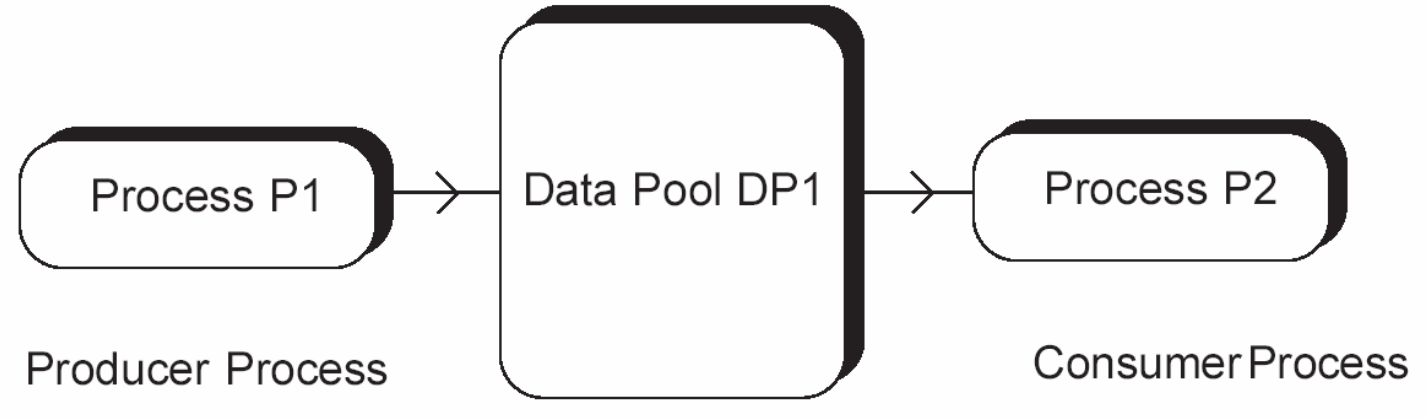
Problem 2 - Process Synchronisation.

- Consider for a moment the functionality performed by processes P1 and P2.
- They are in effect processes whose activities are independent of each other, other than the occasional need to communicate data. In effect they carry out the production or consumption of data in their own time at their own speed.
- For example P1 the Producer, could be executing its program very rapidly, gathering data from say temperature sensors etc. It might wish to store this information inside the data pool DP1 for later consumption by process P2.
- Process P2 on the other hand might be performing complex mathematical operations on the data that it reads from DP1 before attempting to read another set of data.
- Herein lies the problem. Processes P1 and P2 are not synchronised at all, P2 does not know when P1 has updated the pool.
- Similarly, P1 does not know when P2 has read the last set of data from the Pool.



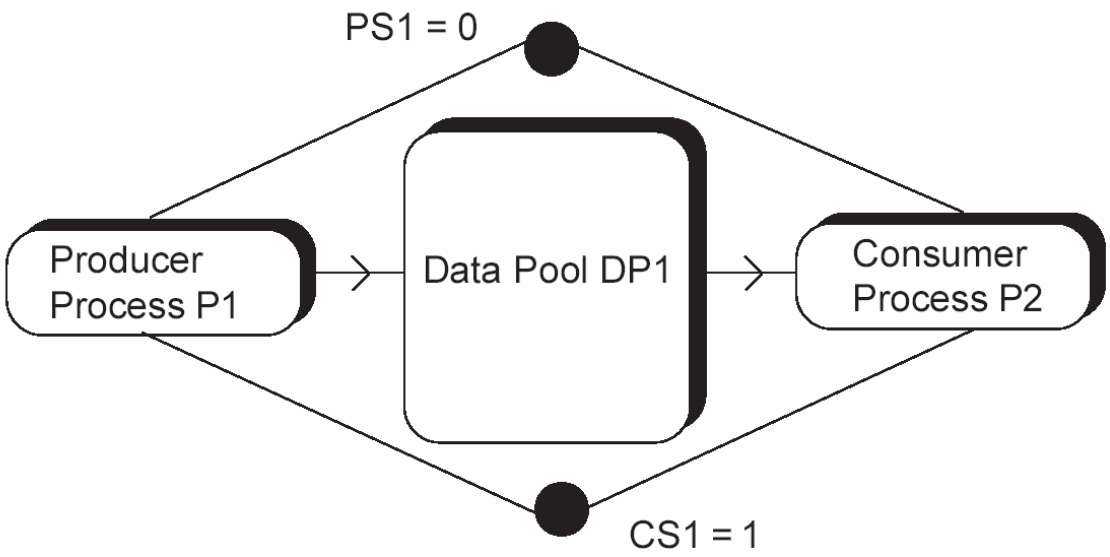
- Because of this lack of synchronisation
  - P1 might generate data faster than P2 can consume it and therefore attempt to update the pool, before P2 has had a chance to read the last update, resulting in the loss of that previous data.
  - P2 might attempt to read the pool more frequently that P1 is generating new data, thus P2 will not know if it is reading the same data again, or if it is new data that just so happens to be the same as the last.

- In effect neither process knows what the other one is doing. To overcome this problem, we need to introduce a mechanism that will allow each process to :-
  - Implement Mutual Exclusion on the non-sharable resource DP1 and
  - Allow both processes to Synchronise their actions with another one.



Solving the Producer Consumer Problem

- To solve both problems, we introduce **two** semaphores, PS1 (referred to as the producer semaphore) and CS1 (the consumer semaphore).
- The semaphore PS1 is signalled by the producer process to indicate to the consumer that it has generated data and placed it in the pool.
- The semaphore CS1 is signalled by the consumer process to indicate to the producer that it has read the data in the pool.
- The consumer semaphore is initially created with the values (CS1 = 1), while the producer semaphore is given the initial value (PS1 = 0).



The Producer and Consumer Process's Operation

- First let's start off by examining the Pseudo-code for each process to show how each interacts with these semaphores

Producer Process P1

```
while( need to produce ) {
        CS1.Wait( )
        Update DataPool( )
        PS1.Signal( )
}
```

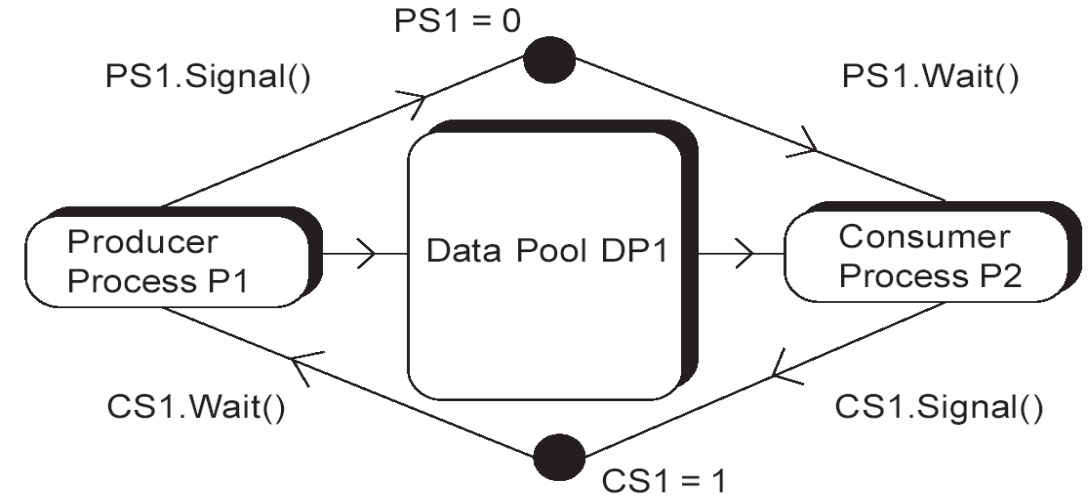
Consumer Process P2

```
while( need to consume ) {
        PS1.Wait( )
        Read DataPool( )
        CS1.Signal( )
}
```

- Notice how each process performs Wait() and Signal() operations on opposite semaphores. For example, the producer P1 performs a Wait() on CS1, while performing a Signal() on PS1 and vice versa.
- Note also how each process performs a Wait() on the opposite semaphore to the other process, e.g. P1 performs a Wait() on CS1, while the consumer, P2 performs a Wait() on PS1.

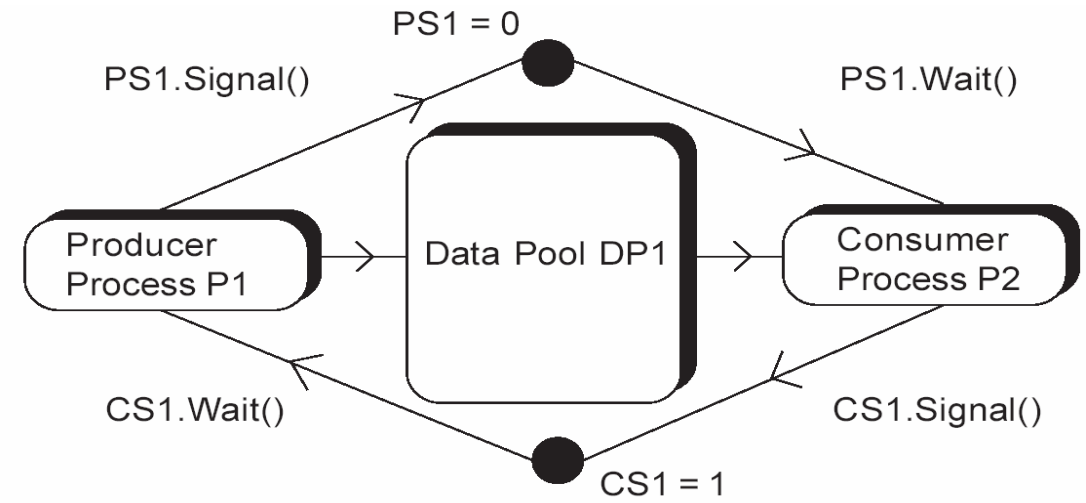
Operation – The Producer goes first

- Imagine the producer gets to the point in its execution where it wishes to generate data first before P2 wants to consume it. It performs a Wait() on CS1 and initially at least is allowed to decrement this semaphore to 0 and continue to generate data for the 1st time.
- If P1 then attempts to go around its '*while*' loop again and generate more data before P2 has consumed it, it will get blocked when it performs a Wait() for the 2nd time on the semaphore CS1 thus preventing the loss of data in DP1.
- In other words it has worked because P1 can update the pool only once before it will get blocked by the Wait() on CS1 .



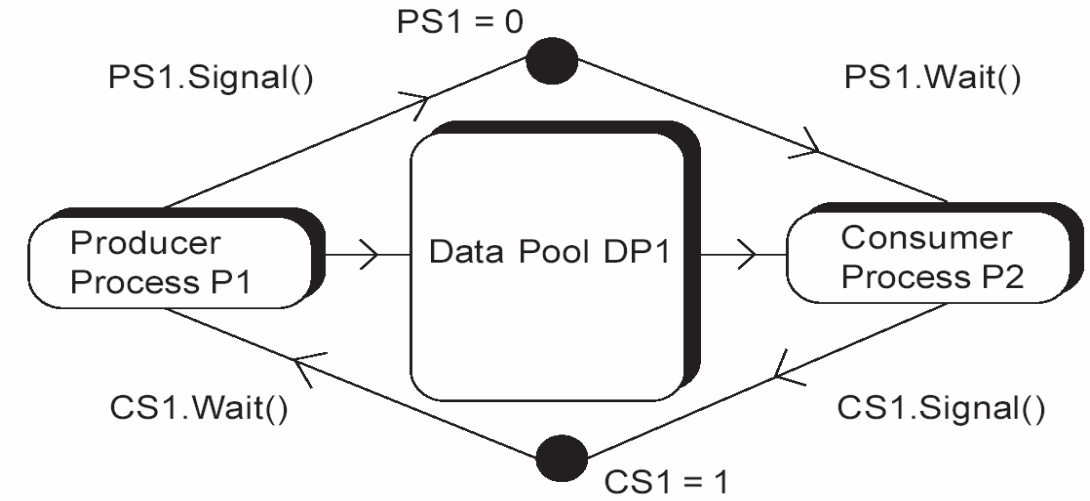
Operation – The Consumer goes first

- Imagine instead that the consumer process P2 gets to the point where it wants to consume data before P1 has produced it
- When P2 performs a Wait() on PS1, initially at least it will get suspended because this semaphore has the value 0.
- This is what we want, because there is nothing in the data pool to read (P1 has not generated anything yet) so P2 should not be allowed to consume data that is not there.



Operation – Both Producer and Consumer working together

- OK, so imagine our Consumer process is currently suspended on PS1 waiting for data as per the previous sheet.
- Along comes the producer process P1 which arrives a bit late (i.e. P2 is waiting for it).
- When P1 performs its Wait() it can decrement CS1 to 0 as before and generate data.
- When P1 Signals() PS1, it will wake up the Consumer Process P2 which is blocked on that semaphore.
- P2 then proceeds to consume the data in the data pool.
- P1 on the other hand goes back around its '**while**' loop and gets suspended on CS1 (which is now at 0). When the consumer has consumed the data in the data pool, it performs a Signal() on PS1 waking up the Producer



Operation (cont…)

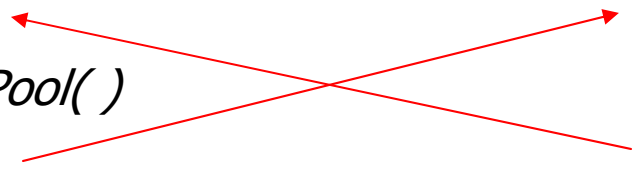
- Thus the Signal() and Wait() operation performs on PS1 and CS1 by Producer and Consumer process act like a handshake as shown below, preventing one process from getting ahead of the other in terms of the production or consumption of data

Producer Process P1

```
while( need to produce ) {
        CS1.Wait( )
        Update DataPool( )
        PS1.Signal( )
}
```

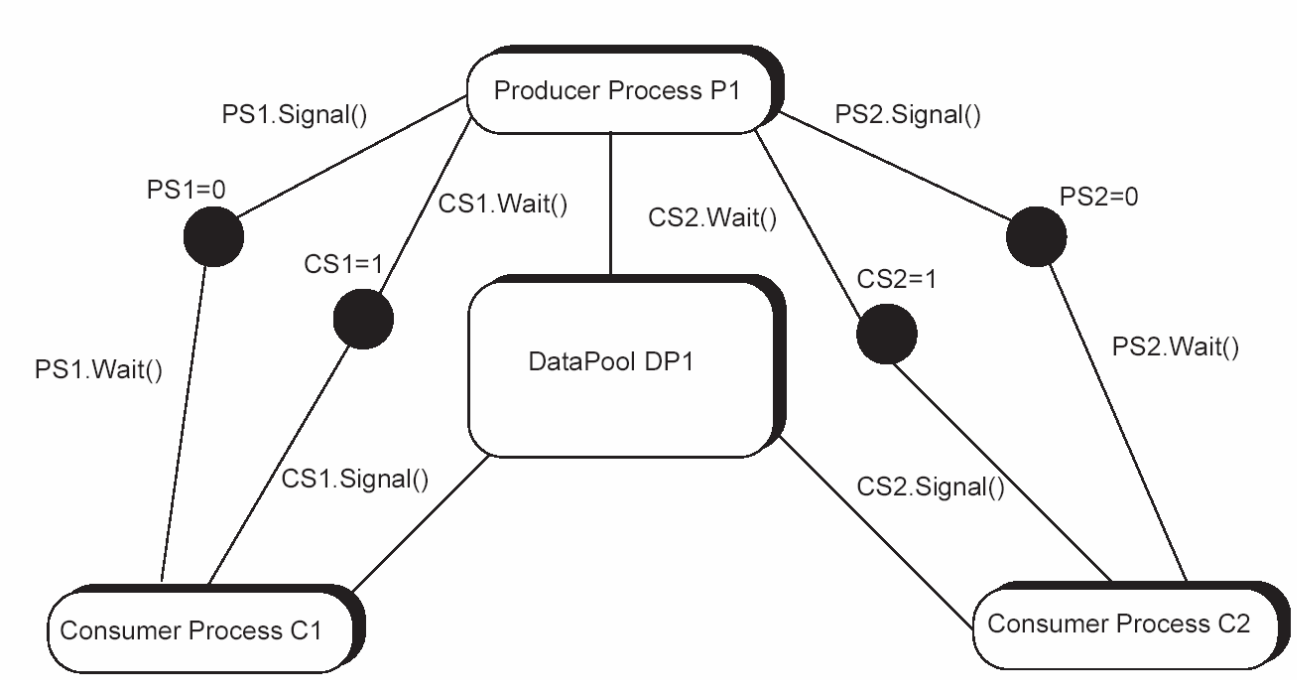
Consumer Process P2

```
while( need to consume ) {
        PS1.Wait( )
        Read DataPool( )
        PS1.Signal( )
}
```



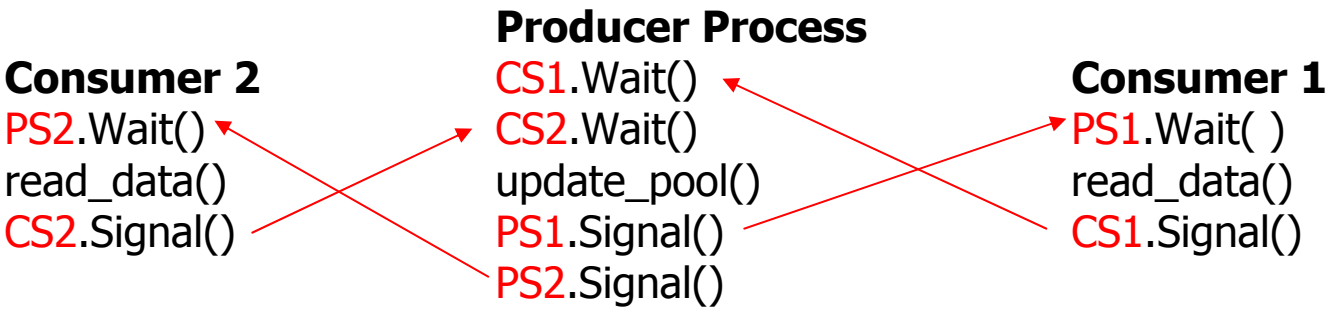
More Producer Consumer Problems - One Producer, Several Consumers

- Consider the arrangement below, where we have one producer, P1 and several (in this case 2) consumers of data, C1 and C2, communicating via the data pool DP1.
- Process P1 must wait until C1 and C2 have both read the data before it attempts to update the pool again. The solution involves the use of two pairs of producer and consumer semaphores as shown below



Solution to One Producer, Several Consumers

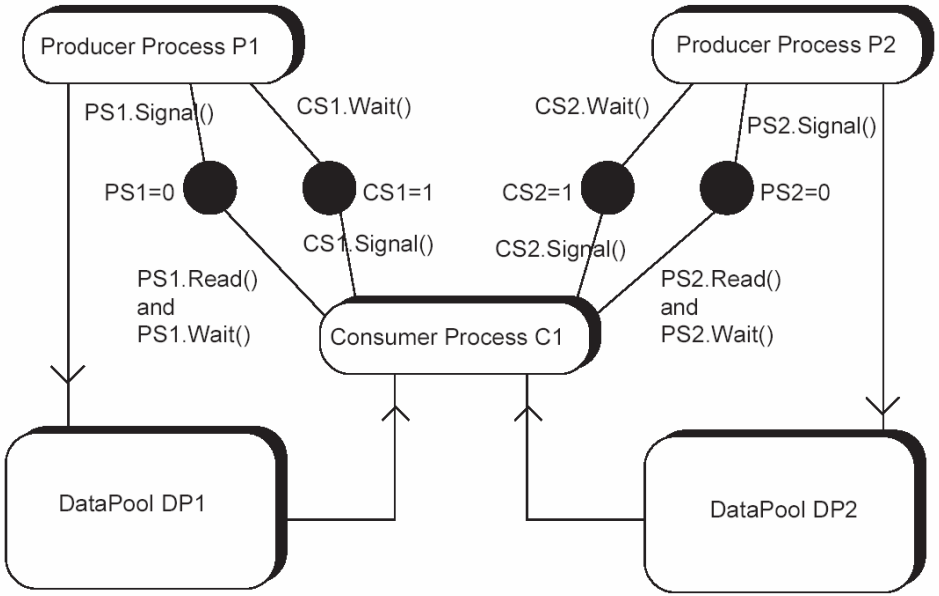
- The code for each consumer and the single producer is shown below, which can be logically and simply extended to include any number of consumer processes.



- From the point of view of each consumer, it appears to be synchronising itself to just a single producer as before, i.e. the other consumers do not affect it.
- In the case of the producer, it has to wait for both consumers to consume their data (the order in which it waits for each does not matter). Then it can update the pool as before signalling both consumer semaphores (again the order or signalling does not matter).

More Producer Consumer Problems - Multiple Producers, One Consumers

- Here we have two producers generating data which they place into two separate data pools for consumption by the single consumer.
- The problem for the consumer is that it does not know which producer will generate data first. If it guesses and waits on PS1 and that guess was incorrect, it will get suspended.
- If in the meantime P2 produced data then the consumer process C1 would not be in a position to deal with the data produced by P2.
- The solution in this case is either multiple threads within the consumer (with each thread responsible for interacting with just one single producer), or introduce a Read() primitive to allow the consumer to POLL a semaphore and hence determine if a producer process has generated data (by signalling PS1 or PS2)



Software Solution

Producer Process

```
CSn.Wait()
    Update Pool n()
PSn.Signal()
```

Consumer Process

```
while( 1 )    {
    if( PS1.Read() > 0)
            read_pool 1 ()
    . . .
    . . .
    if( PSn.Read() > 0)
            read_pool 2 ()        /* etc */
}
```

- **Note:** In the functions read_pool1() and read_pool2() given for the consumer process above, it will be necessary for the consumer to execute a WAIT() and SIGNAL() with the appropriate semaphores to ensure correct handshaking between producer and consumer) e.g.

```
read_pool1()          {
    PS1.Wait()
    read_data_in_pool1
    CS1.Signal()
}
```

```
read_pool2()      {
    PS2.Wait()
    read_data_in_pool2
    CS2.Signal()
}
```