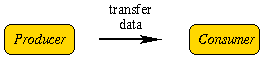
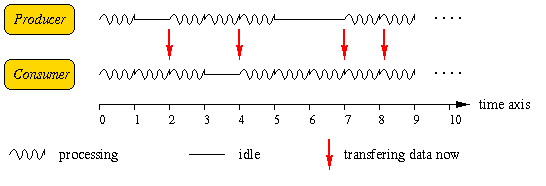
**Producer – Consumer Example**

**The Producer-Consumer Problem**

Let's suppose we have two computers labeled *Producer* and *Consumer*. Producer spends its time grinding away processing its data, but every now and again it wishes to send the fruits of its labor to Consumer. Meanwhile Consumer grinds away processing data received from Producer.



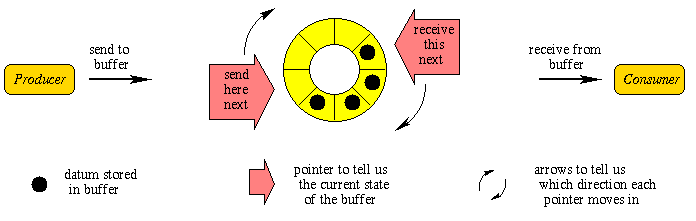
So, what is the problem here ? Producer sends data to Consumer, so what ? Well, Producer can only send each datum when Consumer is ready to receive it, and so the former has to wait doing nothing until the latter is ready to receive. Similarly, Consumer may have to wait for Producer to send a datum before it can continue with its own processing. All in all, as the following diagram shows, both Producer and Consumer may waste as much time do nothing as they actually spend processing.



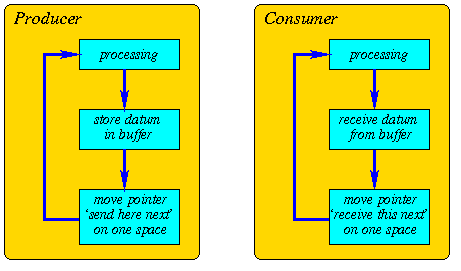
And so our problem is how to transfer data such that Producer and Consumer are never idle unnecessarily.

**First pass at solving the problem**

Any solution to this problem must first remove the obstacle of one computer having to wait for the other before it can send/receive. We do this by introducing a temporary storage area termed a *buffer* where Producer can send its data until such time as Consumer is ready to receive it.



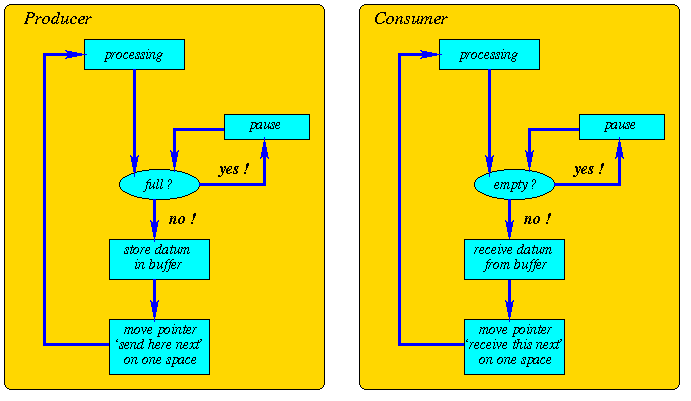
Now Producer and Consumer can each follow their own algorithm (i.e. sequence of steps) as follows, and quite independently of each other.



Note how we make good use of the buffer's spaces by recycling each one once its contents have been received by Consumer. Well, I guess that might seem like the end of the story. Unfortunately the fact that Producer and Consumer can now operate independently of each other raises a difficulty often found in distributed systems. If two computers operate independently could they accidentally step on each other's toes ?

**Second pass at solving the problem**

One potential problem with our solution so far is that Producer might try to send a datum when the buffer is already full of data still waiting to be received by Consumer. Similarly, Consumer might be trying to receive data from an empty buffer, that is, when all the data sent so far by Producer has already been received. We have to find a way to prevent Producer sending to a full buffer, and of Consumer receiving from an empty buffer. We do this by insisting that Producer (resp. Consumer) pause for a moment whenever the buffer is full (resp. empty), which will hopefully give time for Consumer (resp. Producer) a chance to make space (resp. make a deposit) in the buffer. By adding this extra refinement to the algorithms for Producer and Consumer we get the following.



**Final pass at solving the problem**

But, how do we know when the buffer is full, and when it is empty. This we really do have to know in order to prevent the buffer from becoming corrupted. A simple answer would be to keep a tally of the number of unread data items currently in the buffer. Starting at zero, the tally is incremented by one every time a send is made, and decremented by one every time a receive is made. Whenever this number is zero the buffer must be empty, and so receiving is not allowed. When the number equals the overall number of spaces in the buffer then the latter must be full, and so sending is not allowed.

**And finally, ...**

This completes the design of our solution, and so it just remains to code it up in an appropriate programming language.