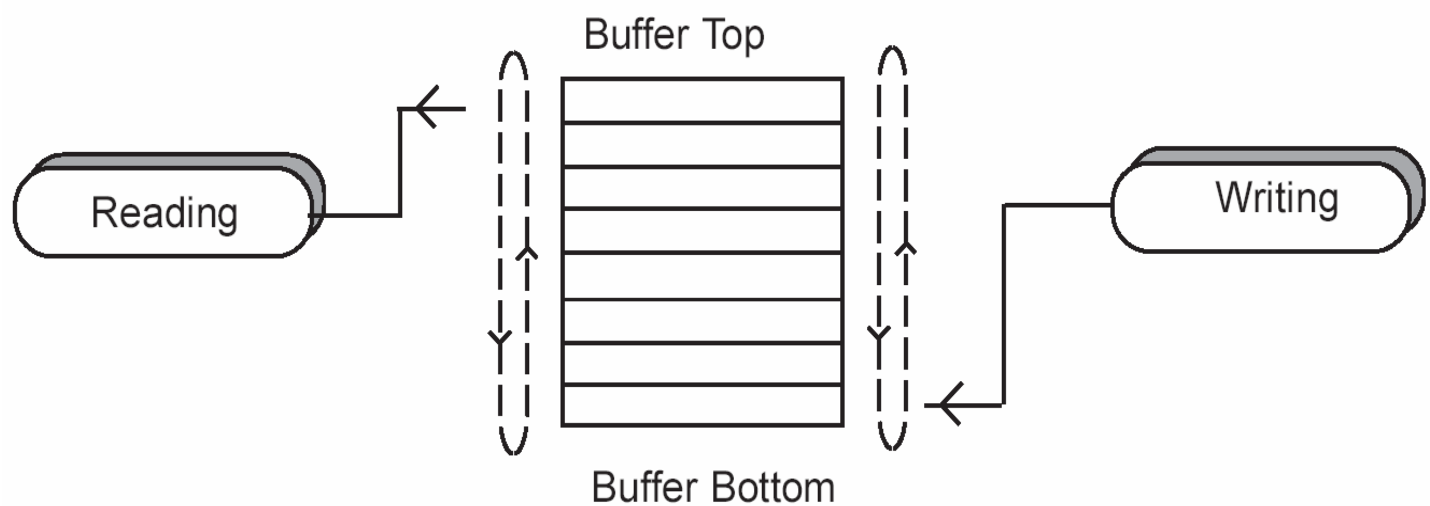
Inter-Process Communication - The Pipeline

- One inter-process communication mechanism that handles synchronisation for itself is the Pipeline. This form of communications is generally characterised by the following properties:-
  - Implemented typically as a wrap around buffer with a finite size, i.e. a circular queue, which is used to store the data being communicated.
  - Accesses to the pipeline are handled via software primitives, i.e. calls to the kernel rather than the have the process directly access the data for itself.
  - This important difference means that access to the pipeline can be managed or controlled by the kernel that can prevent two or more processes accessing it at the same time.
  - Pipelines operate on a FIFO/sequential basis, i.e. data can only be read in the order it was written.
  - That is, random and/or direct access to data is not catered for, which makes it more amenable to implementation in a distributed system using a network. (The kernel accepts the read/write requests from the process and can transmit them serially over a network).
  - Restricted to communicating between two processes only, one reading, the other writing. Multiple pipelines will be required in any situation where the same data has to be transmitted to several reader processes.
  - The read operation is destructive, that is, once data has been read from the pipeline, it is physically deleted and no longer exists in the pipeline.
  - Theoretically data can only be transferred in one direction, from writer to reader, however most operating systems implement bi-directional pipelines by having two separate pipelines hidden within one.

Pipeline Synchronisation Properties

- Any process attempting to write to a full Pipeline will be suspended by the kernel until such time as there is space available to complete the write. This is to prevent data being lost if the circular buffer were to overflow and attempt to wrap around on itself
- Any process attempting to read data from an empty buffer will also be suspended by the kernel until there is sufficient information to complete the transaction. This is to prevent a reading process theoretically reading the same data twice when the buffer wrapped around.
- Finally, any process that has been suspended because it attempted to read from an empty buffer, or because if attempted to write to a full one will be allowed to resume processing automatically when the conditions that lead to its suspension have been removed i.e. the pipeline has data in it, or there is now space in the pipeline respectively.



Using a Pipeline

- Typically a pipeline is accessed using a number of software primitives with hooks to the operating system kernel to handle the suspension and resumption of processing. Such primitives might include the ability to:-
    - Create or Open a data pipeline.
    - Read data from a pipeline.
    - Write to a pipeline.
    - Delete a pipeline.
- The problem with blindly reading from or writing data to a pipeline is that a process runs the risk of getting suspended if the read or write cannot complete due to lack of data or space within the pipeline.
- This could be catastrophic in some systems, because a suspended process would means that all other activities carried out by that process will also get suspended, not just the read or write operation. This could have a ripple effect throughout the system causing other process to suspend themselves and cause lockup in the system.
- For this reason, many operating systems provide a primitive to “test the water” w.r.t. the pipeline and see if a subsequent read or write operation would result in it being suspended. If necessary the process could then defer that read/write until later
- Thus many operating systems will also provide a primitive to :
    - Determine if there is data or space available to be read from or written to the pipeline.

A More Detailed look at the CPipeline Class

- The CPipeline Class encapsulates five member functions to facilitate the creation and use of a pipeline in a program.
- These functions are outlined below with a brief description of what they do. A more detailed description and implementation of them can be found in the rt.h and rt.cpp files.

CPipeline(Name, size)	-	The constructor responsible for creating the pipeline
BOOL Read(void *data, int size)	-	A function to read 'size' bytes of data from the pipeline and store at the address pointed to by 'data'. Returns true/false if the read is successful or if it fails. Note that suspension is not deemed a failure
BOOL Write(void *data, int size)	-	A function to write 'size' bytes of data to the pipeline using data stored in memory at the address pointed to by 'data'. Returns true/false if the read is successful or if it fails. Note that suspension is not deemed a failure
int TestForData()	-	Returns the number of available bytes of data in the pipeline that can be read without the process getting suspended
~CPipeline()	-	A destructor to delete the pipeline at the end of its use

- Tutorial Question 7 demonstrates the use of Pipelines

Example Use of Pipelines: Program 1 – Writing the Data

```
#include     "rt.h"

struct       example {                  // structure template for data to be written to the pipeline
     int     x ;
     float   y ;
} ;

// Some data to be written in to the pipeline.

int   i = 5;                                                        // a simple int
int   array[ 10 ] = { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 0 } ;    // array of 10 integers

char name[15 ] = "Hello World" ;                                    // a string of 15 characters
struct  example mystruct = {2, 5.5 } ;                              // a structure with an int and a float

int   main()
{
      CPipe   p1("MyPipe") ;                           // Create a pipe 'p1' with the name "MyPipe"

      p1.Write(&i, sizeof(i)) ;                        // write the int 'i' to the pipe
      p1.Write(&array[0], sizeof(array)) ;             // write the array of integers' to the pipe
      p1.Write(&name[0], sizeof(name)) ;               // write the string to the pipe
      p1.Write(&mystruct, sizeof(mystruct)) ;          // write the structure to the pipeline

      return 0 ;
}
```

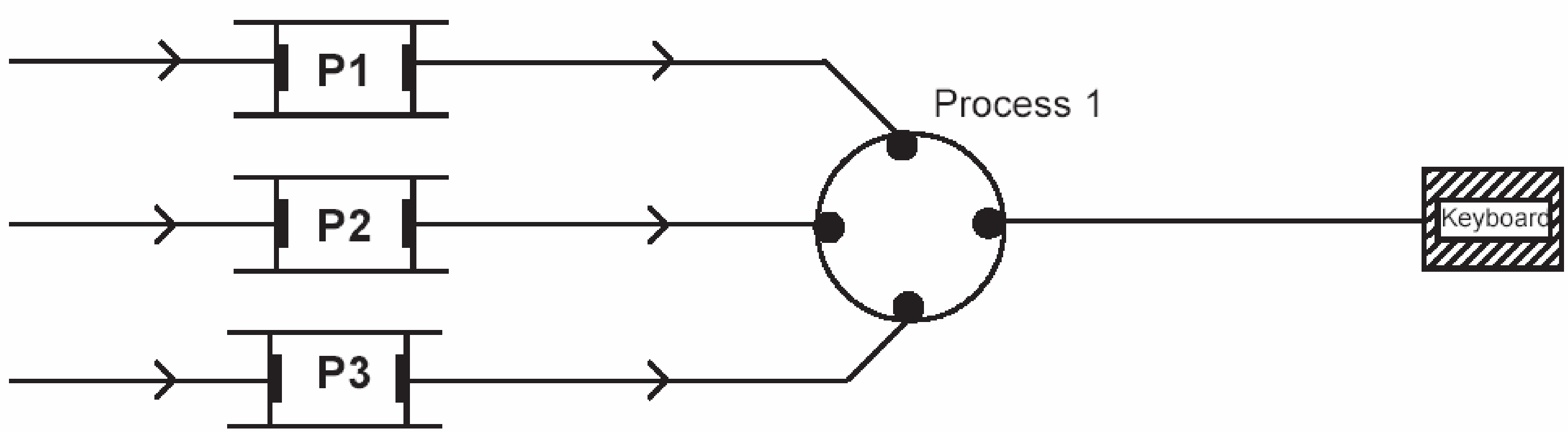
Example Use of Pipelines: Program 2 – Reading the Data

```
#include        "rt.h"
struct          example {                        // structure template for data that we intend to read from pipeline
        int     x ;
        float   y ;
} ;

// Some variables to hold the read from the pipeline.

int     i ;                     // a simple int
int     array[ 10 ]  ;          // array of 10 integers

char   name[15 ] ;              // a string of 15 characters
struct  example mystruct ;      // a structure with an int and a float in it

int     main()
{
        CPipe    p1("MyPipe") ;                  // Create a pipe 'p1' with the name "MyPipe"

        p1.Read(&i, sizeof(i)) ;                 // Read the int 'i' from the pipe
        p1.Read(&array[0], sizeof(array)) ;      // Read the array of integers' from the pipe
        p1.Read(&name[0], sizeof(name)) ;        // Read the string from the pipe
        p1.Read(&mystruct, sizeof(mystruct)) ;   // Read the structure from the pipeline

// Now print out the data read from the pipeline

        printf(" i = %d\n", i) ;
        for( int x = 0; x < 10 ; x ++)
                printf("array[%d] = %d\n", x, array[x]) ;

        printf("%s", name) ;
        printf("mystruct.x = %d, mystruct.y = %f\n", mystruct.x, mystruct.y) ;
        return 0 ;
}
```

Handling Multiple Pipelines

- Suppose a process is actively reading data from say three pipelines and maybe a keyboard (which has properties similar to a pipeline) as shown below. Because data could arrive along any pipeline at any time, the process will probably not be able to predict in advance which pipeline to read first.
- If the process were to make a guess at which pipeline holds data by attempting to read from it, we know from the properties of a pipeline that it will be suspended if there is insufficient data to satisfy the read operation resulting in the process being unable to read data from any other pipeline when that arrives.
- These pipelines would all eventually fill up and lead to their writers also getting suspended. How do we solve this ?

1. Use the TestForData() Primitive
2. Use Multiple Threads, one for each pipeline



Solution using TestForData()

```
#include        "rt.h"

int     pipe1data ;                     // a single integer
float   pipe2data[10] ;                 // an array of 10 floats
char    pipe3data[20] ;                 // a string of up to 20 character
char    KeyData[2]  = {'\0', '\0'} ;    // a 2 character keyboard command initialised to be empty

int     main()
{
        CPipe   p1("Pipe1") ;           // create the three named pipelines
        CPipe   p2("Pipe2") ;
        CPipe   p3("Pipe3") ;

//      Now generate an endless polling loop checking if any data has arrived from any of the four sources. No error checking for the sake of clarity.

        while( 1 )              {
                if ((p1.TestForData()) >= sizeof(pipe1data) )                   // if at least 1 integer in pipeline
                        p1.Read(&pipe1data , sizeof(pipe1data)) ;               // read data from pipe

                if ((p2.TestForData()) >= sizeof(pipe2data) )                   // if at least 10 floats in pipeline
                        p2.Read(&pipe2data , sizeof(pipe2data)) ;               // read data from pipe

                if ((p3.TestForData()) >= sizeof(pipe3data) )                   // if a 20 character string in pipeline
                        p3.Read(&pipe3data , sizeof(pipe3data)) ;               // read data from pipe

//      The primitive TEST_FOR_KEYBOARD() below tests the keyboard to see if any key
//      has been pressed, if so it returns true (i.e. a value other than zero)

                if (TEST_FOR_KEYBOARD() != 0) {         // if a key has been pressed maintain a scrolling array of the last two characters read
                        KeyData[0] = KeyData[1] ;       // move up previous character read
                        KeyData[1] = getch() ;          // read the character from keyboard
                }
        }
}
```

Type Safe Pipelines

- The successful operation of a pipeline depends very much on the reader and writer process reading and writing the same type of data in the same order, e.g. an ‘int’, followed by a ‘float’, then a ‘double’ etc.
- For example, it would be useless and cause chaos in the system if the writer process wrote a ‘float’ into the pipeline, but the reader process read that data as an ‘int’, or perhaps 4 ‘chars’.
- As far as the pipeline is concerned, there is no difference between the data, since they are both (typically) 4 bytes in size and as long as the writer writes a 4 byte item of data into the pipe, and the reader reads the same sized data out, then it’s not the pipelines problem !!
- To avoid these sorts of problems we could introduce an element of type-safety into our programs, in essence we could use a more sophisticated version of the pipeline which only permits data to be read/written according to a template agreed when the pipeline is created.
- For example, the template would state that this pipeline can only be used to hold integers, or float or objects/structures or type ‘X’
- Such a pipeline is represented by a C++ templated class version of the Pipeline, called, not surprisingly CTypedPipe.
- An example of its use is shown overleaf

Example Usage of a Type safe Pipeline

The type of data held by the pipeline

```
#include "rt.h"

CTypedPipe<int>        Pipe1("pipe") ;                        // create an 'int' pipe

int main()
{
      int x, i=5, j=0 ;

      for(x=0; x < 10; x++)              {
             Pipe1.Write(&x) ;                      // note no size argument req'd as size is implied by type of pipeline
             printf("Wrote %d into Pipe. Number of ints in pipe = %d\n", x, Pipe1.TestForData()) ;
      }

      for(x=0; x < 10; x++)              {
             Pipe1.Read(&j) ;                       // again no size argument req'd as size is implied by type of pipeline
             printf("Read %d from Pipe. Number of ints in pipe = %d\n", j, Pipe1.TestForData()) ;
      }

      return 0 ;
}
```