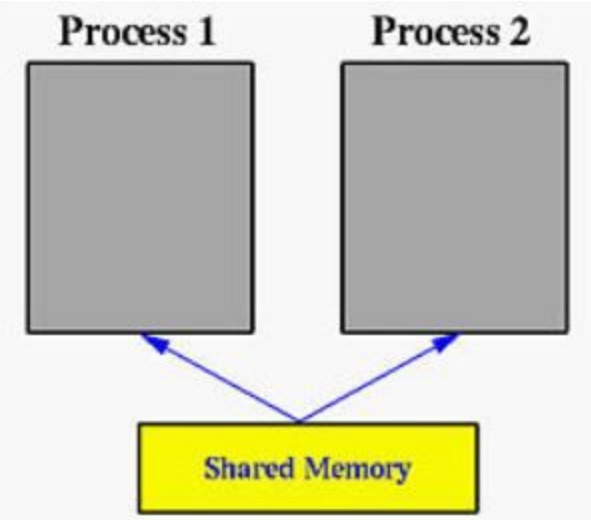
Lab 4 - Shared Memory and Synchronization

- Shared memory is one of the techniques employed by two or more process to communicate. A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces.

- The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address spaces is "extended" by attaching this shared memory.



- In the discussion of the fork() system call, we mentioned that a parent and its children have separate address spaces. While this would provide a more secured way of executing parent and children processes (because they will not interfere with each other), they shared nothing and have no way to communicate with each other.

- The operating system keeps track of the set of shared memory segments. In order to acquire shared memory, we must first request the shared memory from the OS using the shmget() system call. The second parameter specifies the number of bytes of memory requested. shmget() returns a shared memory identifier (SHMID) which is an integer. Refer to the online man pages for details on the other two parameters of shmget().

- The processes must "attach" the shared memory to its local data segment. This is done by the shmat() system call. shmat() takes the SHMID of the shared memory segment as input parameter and returns the address at which the segment has been attached. Thus shmat() returns a char pointer. And once execution is completed each process should "detach" itself from the shared memory after it is used. shmdt() is used for the same.

Consider the sample c code below:

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
      int shmid, *a, *b, i;

      shmid = shmget(IPC_PRIVATE, 2*sizeof(int), 0777|IPC_CREAT);

      if (fork() == 0) {

            /* Child Process */
            b = (int *) shmat(shmid, 0, 0);

            for( i=0; i< 10; i++) {
                  sleep(1);
                  printf("\t\t\t Child reads: %d,%d\n",b[0],b[1]);
            }
            /* each process should "detach" itself from the
               shared memory after it is used */

            shmdt(b);
      }
      else {
            a = (int *) shmat(shmid, 0, 0);

            a[0] = 0; a[1] = 1;
            for( i=0; i< 10; i++) {
                  sleep(1);
                  a[0] = a[0] + a[1];
                  a[1] = a[0] + a[1];
                  printf("Parent writes: %d,%d\n",a[0],a[1]);
            }
            wait(0);

            /* each process should detach itself from the
               shared memory after it is used */

            shmdt(a);
            shmctl(shmid, IPC_RMID, 0);

      }
return 0;
}
```

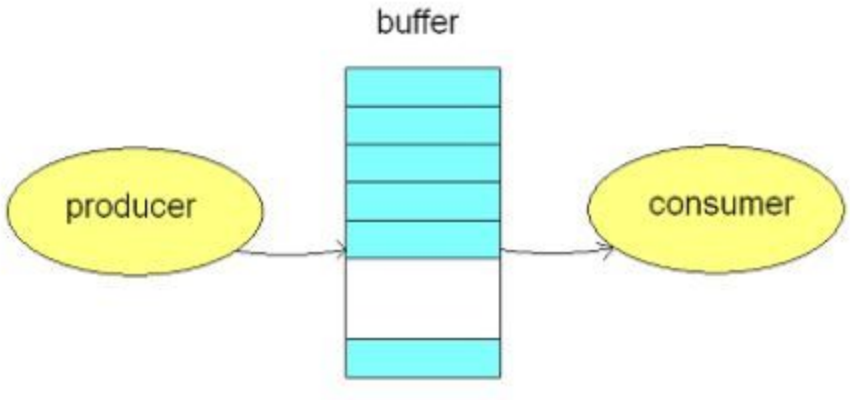
Instructions:

1. Compiler and run the above code and notice the result.
2. Modify the sleep in the child process to sleep(2). What happens now?
3. Modify the sleep in the parent process to sleep(2). What happens now?

Classic Synchronization Problems:

1) Producer - Consumer Problem:

We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section. Implement the problem using given pseudocode.



Pseudocode:

1. Producer

```
item nextProduced;
while (1)
{
while (((in + 1) % BUFFER_SIZE) == out)
; /* do nothing */
buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
}
```

2. Consumer

```
item nextConsumed;
while (1)
{
while (in == out)
; /* do nothing */
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;}
```