Ex. 7B Reader-Writer problem

Dr S. RAJARAJAN/SoC/SASTRA

- The **readers-writers** problem is another example for the common computing problem in concurrency.
- There are at least three variations of the problems, which deal with situations in which many processes/threads try to access the same shared resource at a time.
- Some threads may read and some may write, with the constraint that no process may access the shared resource for either reading or writing, while another process is in the act of writing to it.
- It *is* allowed for two or more readers to access the shared resource at the same time but writing has to happen exclusively.
- Two versions:
  - Readers have priority over writers
  - Writers have priority over readers
- In most solutions, the non-priority group can starve.

Reader-writer Vs Producer-Consumer

RW	**PC**
A writer is simply modifying or overwriting the existing content. It does not increase the quantity of the available data.	A producer generates new data and appends to the existing pool of data. The quantity of data is increased for each addition.
A reader simply reads the data without removing it from the storage.	A consumer removes the data for each consumption leading to emptying of the buffer.
There is no issue like full or empty buffer.	Buffer may become full or empty.
Multiple readers may simultaneously access the data for reading without any conflict. Only writers require exclusive access.	Neither the producer nor the consumer may share the buffer with others. Only one of the entities may access at a time.
Ex. File read, write	Ex. Stack, Queue

Sample Program1

```
#include<stdio.h>
#include<stdlib.h>
#include<semaphore.h>
#include<pthread.h>
#include<unistd.h>
#include<sys/types.h>

// 10 readers & 10 writers
```

```
sem_t mutex;
sem_t rwmutex;

int ind=0;
int readcount=0;
int rcount=0;
void *reader()
{
          readcount++;
          sem_wait(&mutex);
          rcount++;
          if(rcount==1)
          {
               sem_wait(&rwmutex);
          }
          sem_post(&mutex);
          printf("\n %d readier is inside ",(pthread_self()%10));
          sem_wait(&mutex);
          rcount--;
         if(rcount==0)
         {
             sem_post(&rwmutex);
         }
         sem_post(&mutex);
         printf("\n reader is leaving");
}
void *writer()
{
         printf("\n%d writer is about to eneter", (pthread_self()%10));
         sem_wait(&rwmutex);
         printf("\n %d writer is updating ",(pthread_self()%10));
         sem_post(&rwmutex);
}

int main()
{
         int n2,i;
         pthread_t r_tid[10];
         pthread_t w_tid[10];
         sem_init(&mutex,0,1);
         sem_init(&rwmutex,0,1);
         for(i=0;i<n2;i++)
         {
                   pthread_create(&r_tid[i],NULL,reader,NULL);
                   ind++;
                   pthread_create(&w_tid[i],NULL,writer,NULL);
```

```
        }
        for(i=0;i<n2;i++)
        {
                pthread_join(r_tid[i],NULL);
                pthread_join(w_tid[i],NULL);

        }
       return 0;
}
```

Sample Program2

// Display the id of reader and writer

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define MAXTHREAD 10  /* define no of readers */
int data=0;

/* Function prototypes */
void read_data();
void write_data();

void* reader(void*);
void* writer(void*);

sem_t q;              /* establish que */
int rc = 0;           /* number of processes reading or wanting to */
int wc = 0;
int write_request = 0;

int main()
{
   pthread_t readers[MAXTHREAD], writerTh;
   int index;
   int ids[MAXTHREAD]; /* readers and initialize mutex, q and db-set them to 1
*/
   sem_init (&q,0,1);
   for(index = 0; index < MAXTHREAD; index ++)
   {
      ids[index]=index+1;
      if(pthread_create(&readers[index],0,reader,&ids[index])!=0){
         perror("Cannot create reader!");
         exit(1);
      }
```

```
    }
    if(pthread_create(&writerTh,0,writer,0)!=0){
        perror("Cannot create writer");
        exit(1);
    }

    pthread_join(writerTh,0);
    sem_destroy (&q);
    return 0;
}

void* reader(void*arg)                  /* readers function */
{
    int index = *(int*)arg;
    int can_read;
    while(1){
        can_read = 1;
        sem_wait(&q);
        if(wc == 0 && write_request == 0)  rc++;
        else                               can_read = 0;
        sem_post(&q);

        if(can_read) {
            read_data(); //Commence reading
            printf("Thread %d reading\n", index);
            sleep(index);
            sem_wait(&q);
            rc--;     // Current reader leaving after decrementing readers count
            sem_post(&q);
        }
    }
    return 0;
}
;
void* writer(void *arg)          /* writers function */
{
    int can_write;
    while(1){
        can_write = 1;
        sem_wait (&q);
        if(rc == 0)   wc++;
        else          { can_write = 0; write_request = 1; }
        sem_post(&q);

        if(can_write) {
            write_data();
            sleep(3);

            sem_wait(&q);
```

```
            wc--;  // Writer completes writing and decrements writer count before leaving
            write_request = 0;
            sem_post(&q);
        }

    }
    return 0;
}

void read_data()
{
        printf("%d", data);
}
void write_data()
{
        data++;
}
```