```c
// PRODUCER-CONSUMER PROBLEM
// PRODUCER AND CONSUMER ARE THREADS OF A SINGLE PROCESS
// Illustrates the use of POSIX semaphores for synchronization
// Note: buffer capacity is 1 !!!
// JAS

// QUESTION: why is the mutex,
// usually used in the classical producer-consumer problem
// not needed, in this case?

// NOTE: error return codes are not checked ...
// You must add them.

// prod_cons_1.c
// compilation: gcc prod_cons_1.c -lpthread -lrt -Wall -o prod_cons_1

//========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

//========================================================================
#define NOT_SHARED 0  // sem. is not shared w/other processes

//========================================================================
sem_t empty, full;       // the global semaphores
int data;                // shared buffer; capacity=1 !!!
int numItems;            // number of items to be produced/consumed

//========================================================================
// Put items (1, ..., numItems) into the data buffer and sum them
void *producer(void *arg) {
  int total=0, produced;

  printf("Producer running\n");

  for (produced = 1; produced <= numItems; produced++)
  {
    sem_wait(&empty);
    data = produced;
    total = total+data;
    sem_post(&full);
  }

  printf("Producer: total produced is %d\n",total);

  return NULL;
}
```

```c
//=========================================================================
// Get values from the data buffer and sum them
void *consumer(void *arg) {
  int total = 0, consumed;

  printf("Consumer running\n");

  for (consumed = 1; consumed <= numItems; consumed++)
  {
    sem_wait(&full);
    total = total+data;
    sem_post(&empty);
  }

  printf("Consumer: total consumed is %d\n",total);

  return NULL;
}

//=========================================================================
int main(int argc, char *argv[]) {
  pthread_t pid, cid;

  if (argc != 2)
  {
    fprintf(stderr,"USAGE: %s numItems\n",argv[0]);
    exit(1);
  }

  numItems = atoi(argv[1]); // num. of items to be produced/consumed

  sem_init(&empty, NOT_SHARED, 1);  // sem. empty = 1
  sem_init(&full, NOT_SHARED, 0);   // sem.  full = 0

  printf("Main started.\n");

  pthread_create(&pid, NULL, producer, NULL);
  pthread_create(&cid, NULL, consumer, NULL);

  pthread_join(pid, NULL);
  pthread_join(cid, NULL);

  sem_destroy(&empty);
  sem_destroy(&full);

  printf("Main done.\n");
  return 0;
}
```

```c
// POSIX shared memory & semaphore - usage example
// Program that writes a digit sequence in shared memory and
// waits for a reader (reader.c) to read it
// The reader must write an '*'
// at the beginning of the shared memory region
// for signaling the writer that the shared memory region can be removed
// (another semaphore could have been used instead - TO DO BY STUDENTS)
// JAS

// writer.c
// gcc writer.c -lrt -Wall -o writer (don't forget '-lrt')

//=========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <sys/types.h>

#define SHM_SIZE 10

//=========================================================================
//names should begin with '/'
char SEM_NAME[] = "/sem1";
char SHM_NAME[] = "/shm1";

//=========================================================================
int main()
{
  int shmfd;
  char *shm, *s;
  sem_t *sem;
  int i, n;
  int sum = 0;

  //create the shared memory region
  shmfd = shm_open(SHM_NAME,O_CREAT|O_RDWR,0600);

  //TO DO BY STUDENTS:
  //try the following alternative for shm_open() call - note the use of
O_EXCL
  // shmfd = shm_open(SHM_NAME,O_CREAT|O_EXCL|O_RDWR,0600);
  //and comment the
  //  if (shm_unlink(SHM_NAME) < 0) ... call at the end of this program
  //Then run this program twice and explain what happens

  if(shmfd<0)
  {
    perror("WRITER failure in shm_open()");
    exit(1);
  }
  if (ftruncate(shmfd,SHM_SIZE) < 0)
  {
    perror("WRITER failure in ftruncate()");
    exit(2);
  }
```

```c
  //attach this region to virtual memory
  shm = (char *) mmap(0,SHM_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
  if(shm == MAP_FAILED)
  {
    perror("WRITER failure in mmap()");
    exit(3);
  }

  //create & initialize semaphore
  sem = sem_open(SEM_NAME,O_CREAT,0600,0);
  if(sem == SEM_FAILED)
  {
    perror("WRITER failure in sem_open()");
    exit(4);
  }

  //write into shared memory region
  s = shm;
  for(i=0; i<SHM_SIZE-1; i++)
  {
    n = i % 10; sum = sum + n;
    *s++ = (char) ('0' + n);
  }
  *s = (char) 0;

  printf("sum = %d\n", sum);

  sem_post(sem);

  //this loop could be replaced by semaphore use
  //TO DO by students
  printf("Busy waiting for 'reader' to read shared memory ...\n");
  while(*shm != '*')
  {
    sleep(1);
  }

  //close and remove shared memory region and semaphore
  sem_close(sem);
  sem_unlink(SEM_NAME);

  if (munmap(shm,SHM_SIZE) < 0)
  {
    perror("WRITER failure in munmap()");
    exit(5);
  }
  if (shm_unlink(SHM_NAME) < 0)
  {
    perror("WRITER failure in shm_unlink()");
    exit(6);
  }

  exit(0);
}
```

```c
// POSIX shared memory & semaphore - usage example
// Program that reads a digit sequence
// written in shared memory by a writer (writer.c)
// After reading, this program writes an '*'
// at the beginning of the shared memory region
// signaling to the writer that  the region can be removed
// (another semaphore could have been used instead - TO DO BY STUDENTS)
// JAS

// reader.c
// gcc reader.c -lrt -Wall -o reader (don't forget '-lrt')

//=========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <sys/types.h>

#define SHM_SIZE 10

//=========================================================================
//names should begin with '/'
char SEM_NAME[] = "/sem1";
char SHM_NAME[] = "/shm1";

//=========================================================================
int main()
{
  int shmfd;
  char *shm, *s, ch;
  sem_t *sem;
  int sum = 0;

  //open the shared memory region
  shmfd = shm_open(SHM_NAME,O_RDWR,0600);
  if(shmfd<0)
  {
    perror("READER failure in shm_open()");
    exit(1);
  }

  //attach this region to virtual memory
  shm = (char *) mmap(0,SHM_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
  if(shm == MAP_FAILED)
  {
    perror("READER failure in mmap()");
    exit(2);
  }

  //open existing semaphore
  sem = sem_open(SEM_NAME,0,0600,0);
  if(sem == SEM_FAILED)
  {
    perror("READER failure in sem_open()");
    exit(3);
  }
```

```
  //wait for writer to stop writing
  sem_wait(sem);

  //read the message
  s = shm;
  for (s=shm; *s!=0; s++)
  {
    ch = *s;
    putchar(ch);
    sum = sum + (ch - '0');
  }
  printf("\nsum = %d\n", sum);

  //once done signal exiting of reader
  //could be replaced by semaphore use (TO DO by students)
  *shm = '*';

  //close semaphore and unmap shared memory region
  sem_close(sem);

  if (munmap(shm,SHM_SIZE) < 0)
  {
    perror("READER failure in munmap()");
    exit(4);
  }

  exit(0);
}
```

```c
// PRODUCER-CONSUMER PROBLEM
// PRODUCERS AND CONSUMERS ARE THREADS OF A SINGLE PROCESS
// SYNCHRONIZATION USING CONDITION VARIABLES
// JAS

//========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

//========================================================================
#define BUFSIZE 8
#define NUMITEMS 100

//========================================================================
int buffer[BUFSIZE];
int bufin = 0;
int bufout = 0;
int items = 0;
int slots = 0;
int sum = 0;

//========================================================================
pthread_mutex_t buffer_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  slots_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t  items_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slots_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t items_lock = PTHREAD_MUTEX_INITIALIZER;

//========================================================================
void put_item(int item)
{
  pthread_mutex_lock(&buffer_lock);
  buffer[bufin] = item;
  bufin = (bufin + 1) % BUFSIZE;
  pthread_mutex_unlock(&buffer_lock);
  return;
}

//========================================================================
void get_item(int *itemp)
{
  pthread_mutex_lock(&buffer_lock);
  *itemp = buffer[bufout];
  bufout = (bufout + 1) % BUFSIZE;
  pthread_mutex_unlock(&buffer_lock);
  return;
}

//========================================================================
void *producer(void * arg)
{
  int i;
  for (i = 1; i <= NUMITEMS; i++)
  {
    /* acquire right to a slot */
    pthread_mutex_lock(&slots_lock);
    printf("Producer: available slots = %d\n",slots);
    while (!(slots > 0))
      pthread_cond_wait (&slots_cond, &slots_lock);
```

```
    slots--;
    pthread_mutex_unlock(&slots_lock);
    put_item(i);
    printf("Producer: produced item %3d\n",i);
    /* release right to an item */
    pthread_mutex_lock(&items_lock);
    items++;
    pthread_cond_signal(&items_cond);
    pthread_mutex_unlock(&items_lock);
  }
  pthread_exit(NULL);
}

//========================================================================
void *consumer(void *arg)
{
  int myitem;
  int i;
  for (i = 1; i <= NUMITEMS; i++)
  {
    pthread_mutex_lock(&items_lock);
    printf("Consumer: available items = %d\n",items);
    while(!(items > 0))
      pthread_cond_wait(&items_cond, &items_lock);
    items--;
    pthread_mutex_unlock(&items_lock);
    get_item(&myitem);
    printf("Consumer: consumed item %3d\n",myitem);
    sum += myitem;
    pthread_mutex_lock(&slots_lock);
    slots++;
    pthread_cond_signal(&slots_cond);
    pthread_mutex_unlock(&slots_lock);
  }
  pthread_exit(NULL);
}

//========================================================================
int main(void)
{
  pthread_t prodtid, constid;
  int i, total;
  slots = BUFSIZE;
  total = 0;

  for (i = 1; i <=NUMITEMS; i++)
    total += i;
  printf("The checksum is %d\n", total);

  if (pthread_create(&constid, NULL, consumer, NULL))
  {
    perror("Could not create consumer");
    exit(EXIT_FAILURE);
  }

  if (pthread_create(&prodtid, NULL, producer, NULL))
  {
    perror("Could not create producer");
    exit(EXIT_FAILURE);
  }
```

```
  pthread_join(prodtid, NULL);
  pthread_join(constid, NULL);

  printf("The threads produced the sum %d\n", sum);
  exit(EXIT_SUCCESS);//EXIT_SUCCESS e EXIT_FAILURE <- stdlib.h
}
```

```c
// PRODUCER-CONSUMER PROBLEM
// PRODUCERS AND CONSUMERS ARE INDEPENDENT PROCESSES
// BUFFER IS IN SHARED MEMORY (EXTERNAL TO THE PROCESSES)
// SYNCHRONIZATION USING CONDITION VARIABLES (IN SHARED MEMORY)
// JAS

// PRODUCER program
// prod_01.c (to be run together with cons_01.c)

//========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <sys/types.h>

//========================================================================
#define SHM_NAME "/shm1"
#define BUFSIZE 5
#define NUMITEMS 50

//========================================================================
typedef struct {
  pthread_mutex_t buffer_lock;
  pthread_cond_t  slots_cond;
  pthread_cond_t  items_cond;
  pthread_mutex_t slots_lock;
  pthread_mutex_t items_lock;
  int buffer[BUFSIZE];
  int bufin;
  int bufout;
  int items;
  int slots;
  int sum;
} Shared_memory;

//========================================================================
Shared_memory * create_shared_memory(char* shm_name, int shm_size)
{
  int shmfd;
  Shared_memory *shm;

  //create the shared memory region
  shmfd = shm_open(SHM_NAME,O_CREAT|O_RDWR,0660); // try with O_EXCL

  if(shmfd<0)
  {
    perror("Failure in shm_open()");
    return NULL;
  }

  //specify the size of the shared memory region
  if (ftruncate(shmfd,shm_size) < 0)
  {
    perror("Failure in ftruncate()");
    return NULL;
  }
```

```
  //attach this region to virtual memory
  shm = mmap(0,shm_size,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
  if(shm == MAP_FAILED)
  {
    perror("Failure in mmap()");
    return NULL;
  }

  //initialize data in shared memory
  shm->bufin = 0;
  shm->bufout = 0;
  shm->items = 0;
  shm->slots = BUFSIZE;
  shm->sum = 0;

  return (Shared_memory *) shm;
}

//========================================================================
void destroy_shared_memory(Shared_memory *shm, int shm_size)
{
  if (munmap(shm,shm_size) < 0)
  {
    perror("Failure in munmap()");
    exit(EXIT_FAILURE);
  }
  if (shm_unlink(SHM_NAME) < 0)
  {
    perror("Failure in shm_unlink()");
    exit(EXIT_FAILURE);
  }
}

//========================================================================
void init_sync_objects_in_shared_memory(Shared_memory *shm)
{
  pthread_mutexattr_t mattr;
  pthread_mutexattr_init(&mattr);
  pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);

  pthread_mutex_init(&shm->buffer_lock, &mattr);
  pthread_mutex_init(&shm->slots_lock, &mattr);
  pthread_mutex_init(&shm->items_lock, &mattr);

  pthread_condattr_t cattr;
  pthread_condattr_init(&cattr);
  pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

  pthread_cond_init(&shm->slots_cond, &cattr);
  pthread_cond_init(&shm->items_cond, &cattr);
}

//========================================================================
void put_item(int item, Shared_memory *shm)
{
  pthread_mutex_lock(&shm->buffer_lock);
  shm->buffer[shm->bufin] = item;
  shm->bufin = (shm->bufin + 1) % BUFSIZE;
  pthread_mutex_unlock(&shm->buffer_lock);
  return;
}
```

```
//=========================================================================
void *producer(void * arg)
{
  int i;
  Shared_memory *shm = arg;

  printf("In producer thread\n");

  for (i = 1; i <= NUMITEMS; i++)
  {
    // wait for a slot to be available
    pthread_mutex_lock(&shm->slots_lock);
    printf("Producer: available slots = %d\n",shm->slots);
    while (!(shm->slots > 0))
      pthread_cond_wait (&shm->slots_cond, &shm->slots_lock);
    shm->slots--;
    pthread_mutex_unlock(&shm->slots_lock);

    // produce item
    put_item(i,shm);
    printf("Producer: produced item %3d\n",i);

    // update num. produced items and notify consumer
    pthread_mutex_lock(&shm->items_lock);
    shm->items++;
    pthread_cond_signal(&shm->items_cond);
    pthread_mutex_unlock(&shm->items_lock);
  }
  pthread_exit(NULL);
}

//=========================================================================
int main(void){
  pthread_t prodtid;
  int i, total;
  Shared_memory *shmem;

  printf("\nPRODUCER: starting after 5 seconds ...\n");
  sleep(5);

  if ((shmem = create_shared_memory(SHM_NAME, sizeof(Shared_memory))) ==
NULL)
  {
    perror("PRODUCER: could not create shared memory");
    exit(EXIT_FAILURE);
  }

  init_sync_objects_in_shared_memory(shmem);

  // NOT NECESSARILY A THREAD ... COULD BE JUST A CALL TO producer()
FUNCTION
  if (pthread_create(&prodtid, NULL, producer, shmem))
  {
    exit(EXIT_FAILURE);
  }

  pthread_join(prodtid, NULL);

  destroy_shared_memory(shmem, sizeof(Shared_memory));
```

```
  total = 0;
  for (i = 1; i <=NUMITEMS; i++)
    total += i;
  printf("PRODUCER: the checksum is %d\n", total);

  exit(EXIT_SUCCESS);
}
```

```c
// PRODUCER-CONSUMER PROBLEM
// PRODUCERS AND CONSUMERS ARE INDEPENDENT PROCESSES
// BUFFER IS IN SHARED MEMORY (EXTERNAL TO THE PROCESSES)
// SYNCHRONIZATION USING CONDITION VARIABLES (IN SHARED MEMORY)
// JAS

// CONSUMER program
// cons_01.c (to be run together with prod_01.c)

//=========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <sys/types.h>

//=========================================================================
#define SHM_NAME "/shm1"
#define BUFSIZE 5
#define NUMITEMS 50

//=========================================================================
typedef struct {
  pthread_mutex_t buffer_lock;
  pthread_cond_t  slots_cond;
  pthread_cond_t  items_cond;
  pthread_mutex_t slots_lock;
  pthread_mutex_t items_lock;
  int buffer[BUFSIZE];
  int bufin;
  int bufout;
  int items;
  int slots;
  int sum;
} Shared_memory;

//=========================================================================
Shared_memory * attach_shared_memory(char* shm_name, int shm_size)
{
  // PÔR PROD. E CONSUM. A TENTAR CRIAR E SE NÃO CONSEGUIR FAZ ATTACH
...???
  int shmfd;
  Shared_memory *shm;

  shmfd = shm_open(SHM_NAME,O_RDWR,0660);

  if(shmfd<0)
  {
    perror("Failure in shm_open()");
    return NULL;
  }

  //attach this region to virtual memory
  shm = mmap(0,shm_size,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
  if(shm == MAP_FAILED)
  {
    perror("Failure in mmap()");
```

```
    return NULL;
  }

  return (Shared_memory *) shm;

}

//========================================================================
void get_item(int *itemp, Shared_memory *shm)
{
  pthread_mutex_lock(&shm->buffer_lock);
  *itemp = shm->buffer[shm->bufout];
  shm->bufout = (shm->bufout + 1) % BUFSIZE;
  pthread_mutex_unlock(&shm->buffer_lock);
  return;
}

//========================================================================
void *consumer(void *arg)
{
  int myitem;
  int i;
  Shared_memory *shm = arg;

  printf("In consumer thread\n");

  for (i = 1; i <= NUMITEMS; i++)
  {
    // wait for an item to be available
    pthread_mutex_lock(&shm->items_lock);
    printf("Consumer: available items = %d\n",shm->items);
    while(!(shm->items > 0))
      pthread_cond_wait(&shm->items_cond, &shm->items_lock);
    shm->items--;
    pthread_mutex_unlock(&shm->items_lock);

    // consume an item
    get_item(&myitem,shm);
    printf("Consumer: consumed item %3d\n",myitem);

    shm->sum += myitem;

    //update num. available slots and notify producer
    pthread_mutex_lock(&shm->slots_lock);
    shm->slots++;
    pthread_cond_signal(&shm->slots_cond);
    pthread_mutex_unlock(&shm->slots_lock);
  }
  pthread_exit(NULL);
}

//========================================================================
int main(void)
{
  pthread_t constid;
  Shared_memory *shmem;

  printf("\nCONSUMER: starting after 10 seconds ...\n");
  sleep(10);
```

```
  if ((shmem = attach_shared_memory(SHM_NAME, sizeof(Shared_memory))) ==
NULL)
  {
    perror("CONSUMER: could not attach shared memory");
    exit(EXIT_FAILURE);
  }

  // NOT NECESSARILY A THREAD ... COULD BE JUST A CALL TO consumer()
FUNCTION
  if (pthread_create(&constid, NULL, consumer, shmem))
  {
    perror("CONSUMER: could not create consumer");
    exit(EXIT_FAILURE);
  }

  pthread_join(constid, NULL);
  printf("CONSUMER: the threads produced the sum %d\n", shmem->sum);

  if (munmap(shmem,sizeof(Shared_memory)) < 0)
  {
    perror("Failure in munmap()");
    exit(EXIT_FAILURE);
  }

  exit(EXIT_SUCCESS);
}
```

```c
// POSIX CONDITION VARIABLES
// Illustration of pthread_cond_broadcast()
// cond_broadc_01.c
// JAS

//========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>

//========================================================================
#define NTHREADS    5

//========================================================================
int conditionMet = 0;
pthread_cond_t cond  = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

//========================================================================
// Function to check the return code
// and exit the program if the function call failed
void checkResult(char *string, int err)
{
  if (err != 0)
  {
    printf("Error %d on %s\n", err, string);
    exit(EXIT_FAILURE);
  }
  return;
}

//========================================================================
void *threadFunc(void *arg)
{
  int res;
  int threadNum = *(int *)arg;

  res = pthread_mutex_lock(&mutex);
  checkResult("pthread_mutex_lock()\n", res);

  while (!conditionMet)
  {
    printf("Thread %d blocked because condition is not met\n", threadNum);
    res = pthread_cond_wait(&cond, &mutex);
    checkResult("pthread_cond_wait()\n", res);
  }

  printf("Thread %d executing critical section for 5 seconds ...\n",
threadNum);
  sleep(5);

  res = pthread_mutex_unlock(&mutex);
  checkResult("pthread_mutex_lock()\n", res);
  return NULL;
}
```

```
//============================================================================
int main(int argc, char *argv[])
{
  int res=0;
  int i;
  int threadnum[NTHREADS];
  pthread_t threadId[NTHREADS];

  printf("Main thread: creating %d threads\n", NTHREADS);
  for(i=0; i<NTHREADS; ++i)
  {
    threadnum[i]=i+1;
    res = pthread_create(&threadId[i], NULL, threadFunc, (void*)
&threadnum[i]);
    checkResult("pthread_create()\n", res);
  }
  printf("Main thread: doing some work until condition is met ...\n");
  sleep(10);
  //The condition has occured ...! Don't ask me what condition or why ...

  //Set the flag and wake up any waiting threads
  res = pthread_mutex_lock(&mutex);
  checkResult("pthread_mutex_lock()\n", res);
  conditionMet = 1;
  printf("Main thread: the condition was met;\n waking up all waiting
threads, using pthread_cond_broadcast()...\n");
  res = pthread_cond_broadcast(&cond);
  checkResult("pthread_cond_broadcast()\n", res);

  res = pthread_mutex_unlock(&mutex);
  checkResult("pthread_mutex_unlock()\n", res);

  printf("Main thread: waiting for threads and cleanup\n");
  for (i=0; i<NTHREADS; ++i)
  {
    res = pthread_join(threadId[i], NULL);
    checkResult("pthread_join()\n", res);
  }

  res = pthread_cond_destroy(&cond);
  checkResult("pthread_cond_destroy()\n", res);
  res = pthread_mutex_destroy(&mutex);
  checkResult("pthread_mutex_destroy()\n", res);

  printf("Main thread: completed.\n");
  return 0;
}
```