Exercise 1

An ASCII file, stored in the UNIX system, includes an undefined number of records. Each record includes 4 fields: an integer value, two strings, and a real number. All 4 fields have variable size and are separated by an unknown number of spaces. The two strings have a maximum size equal to 100 characters. The following is a possible correct format of such a file:

```
123 string1 STRING2 45.67
68799 ABC VWXYZ -12345.879
. . .
```

Show how it is possible to use the UNIX system call **read** to store the entire file in an array of structures of type **record_t**. The number of records stored in the file is indicated on the first line of the file. The array of structures must be dynamically allocated by the function.

The structure **record_t** is defined as follow:

```
#define N 101

struct record_s {
  int  i;
  char s1[N], s2[N];
  float f;
} record_t;
```

Solution

The number of records is on the first line

```
#define REC_SIZE sizeof(record_t)
#define MAX_C 256 // I suppose each record line is no longer than MAX_C=256 ASCII characters

void my_read(const char *fileName) {
  int fd, i, j, nR, n;
  record_t *record;
  char buffer[MAX_C], c;

  fd = open(fileName, O_RDONLY);
  i = 0;
  do {
    read(fd, &c, sizeof(char));
    buffer[i++] = c;
  } while (c != '\n');
  buffer[i] = '\0';

  n = atoi(buffer);
  record = (record_t*)malloc(n * sizeof(record_t));

  for (i = 0; i < n; i++) {
    j = 0;
    do {
       read(fd, &c, sizeof(char));
```

```
        buffer[j++] = c;
      } while (c != '\n');
      buffer[j] = '\0';
      fprintf(buffer, "%d %s %s %f", &record[i].i, record[i].s1, record[i].s2, &record[i].f);
    }
}
```

Exercise 2

First, illustrate the use of condition variables in the UNIX system.

Then, clarify how they can be used in a Producer and Consumer scheme (with P producers and C consumers, where P and C are generally larger than 1).

Please, remind the following system calls.

```
int pthread_cond_init (pthread_cond_t *restrict cond,const
pthread_condattr_t *restrict attr);
int pthread_cond_wait (pthread_cond_t *restrict cond,pthread_mutex_t
*restrict mutex);
int pthread_cond_signal (pthread_cond_t *cond);int pthread_cond_broadcast
(pthread_cond_t *cond);
int pthread_cond_destroy (pthread_cond_t *cond);
```

Solution

Condition variables provide a place for threads to rendezvous
Condition variables allow threads to wait in a race-free way for arbitrary conditions to occur
The condition itself is protected by a mutex
A thread must first lock the mutex to change the condition state
Other threads will not notice the change until they acquire the mutex, because the mutex must be locked to be able to evaluate the condition.

Condition variables are a synchronization mechanism among threads that let them to wait a certain condition to happen.

```
typedef struct cond_s {
     pthread_cond_t full, empty;
     pthread_mutex_t m;
     int count;
} cond_t;

void Producer() {
     pthread_mutex_lock(&cond.m);
     while (cond.count == MAX)
          pthread_cond_wait(&cond.empty, &cond.m);
     enqueue();
     cond.count++;
     pthread_mutex_unlock(&cond.m);

     pthread_cond_signal(&cond.full);
}

void Consumer() {
     pthread_mutex_lock(&cond.m);
     while (cond.count == 0)
         pthread_cond_wait(&cond.full, &cond.m);
     dequeue();
     cond.count++;
     pthread_mutex_unlock(&cond.m);
     pthread_cond_signal(&cond.empty);
}

void CondInit(cond_t *cond, int count)
{
     pthread_cond_init(&cond->full, NULL);
     pthread_cond_init(&cond->empty, NULL);
     pthread_mutex_init(&cond->m, NULL);
     cond->count = 0;
}
```

Laface

```
static void init (struct prodcons *b) {
  pthread_mutex_init (&b->lock, NULL);
  pthread_cond_init (&b->notempty, NULL);
  pthread_cond_init (&b->notfull, NULL);
  b->readpos = 0;
  b->writepos = 0;
  b->count = 0;
}

/* Store an integer in the buffer */
static void put (struct prodcons *b, int data) {
  pthread_mutex_lock (&b->lock);
  /* Wait until buffer is not full */
  while (b->count == BUFFER_SIZE) {
      pthread_cond_wait (&b->notfull, &b->lock);
      /* pthread_cond_wait reacquired b->lock before returning */
    }
  /* Write the data and advance write pointer */
  b->buffer[b->writepos] = data;
  b->writepos++;
  if (b->writepos >= BUFFER_SIZE)
    b->writepos = 0;
  b->count++;
  /* Signal that the buffer is now not empty */
  pthread_cond_signal (&b->notempty);
  pthread_mutex_unlock (&b->lock);
}

/* Read and remove an integer from the buffer */
static int get (struct prodcons *b) {
  int data;
  pthread_mutex_lock (&b->lock);
  /* Wait until buffer is not empty */
  while (b->count == 0) {
      pthread_cond_wait (&b->notempty, &b->lock);
    }
  /* Read the data and advance read pointer */
  data = b->buffer[b->readpos];
  b->readpos++;
  if (b->readpos >= BUFFER_SIZE)
    b->readpos = 0;
  b->count--;
  /* Signal that the buffer is now not full */
  pthread_cond_signal (&b->notfull);
  pthread_mutex_unlock (&b->lock);
  return data;
}
```

Lucidi

```
#include <pthread.h>
...
pthread_cond_t cond_var = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
producer
produce(data);
pthread_mutex_lock (&mutex);
pthread_cond_signal (&cond_var);
pthread_mutex_unlock (&mutex);
consumer
while (1) {
pthread_mutex_lock (&mutex);
while (condition == FALSE) {
pthread_cond_wait (&cond_var, &mutex);
}
... CS ...
pthread_mutex_unlock (&mutex);
}
```

Exercise 3

A Windows program initializes a semaphore named **sem** to N, and then it allow **at most N** thread entering the critial section **CS** using in all threads the following prologue and epilogue:

```
WaitForSingleObject (sem,  INFINITE);
CS
ReleaseSemaphore (sem, 1, &pc);
```

Re-implement the same prologue and epilogue using only mutexes as synchronization strategies.

Please, remind and use only the following synchronization system calls.

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpsa,BOOL fInitialOwner,LPCTSTR
lpszMutexName);
BOOL ReleaseMutex (HANDLE hMutex);
```

Solution

```
n = N;
me = CreateMutex (NULL, ..., ...);

myWait (me) {
  WFSO (me, INFINITE);
  n--;
  if (n >= 0)
    releaseMutex (me);
  else {
    releaseMutex (me);
    WFSO (w);
  }
}

myRelease (me) {
  WFSO (me, INFINITE);
  if (n == 0)
    releaseMutex (w);
  n++;
  releaseMutex (me);
}
```

Exercise 4

A program must manipulate an extremely large file (e.g., several GBytes), thus it decides to do that using several threads each one manipulating a section of the file over and over again until the file has been completely manipulated.

Following this scheme, write a program able to run N threads (where N is an integer value, e.g., 8, 12, 16) to manipulate the file. Each one of the threads will manipulate 1GBytes of the file, **mapping** the correct section of it into its local memory. For the sake of simplicity, file manipulation consists in reading the file byte-by-byte and rewriting each byte by increasing its value of a constant value C (e.g., C=5) module 256.

For example, if the file is large 10 GBytes and there are 4 threads, thread number 3 may manipulate the first GByte of the file, thread number 2 the second GByte, thread number 4 the third GByte, etc., until the entire file has been read and written and all threads terminate.

Please remind the following system calls.

```
LPVOID MapViewOfFile (HANDLE hMapObject,DWORD dwAccess,DWORD
dwOffsetHigh,DWORD dwOffsetLow,SIZE_T dwNumberOfByteToMap);
BOOL UnmapViewOfFile (LPVOID lpBaseAdress);
```

If you do not remember the exact syntax of other Windows API system calls, write down a version that likely resembles what you remember together some a C comment, briefly summarizing what you were willing to use.

Solution

See other exam

```
/*
Pseudo function prototypes
CreateThread(function pointer, pointer to par, pointer to thread id...)
CreateFileMapping(file handle, mode, how many bytes low, how many bytes high)
CreateMutex(security attributes, initial owner, name)
*/
#define N 4

int section = 0;
BOOL done = false;
HANDLE fHandle;
HANDLE mHandle;
HANDLE mut;
LARGE_INTEGER file_size;
```

```
DWORD WINAPI thread_func(LPVOID args) {
  HANDLE mtHandle = mHandle;
  LARGE_INTEGER start_offset = {.QuadPart = 0};
  LARGE_INTEGER size_section = {.QuadPart = 0};

  while(true) {
   WaitForSingleObject(mut, INFINITE);
   if(!done) {
    if(file_size.QuadPart - section*(1<<30) > 0) // 1 << 30 = 2^30
        start_offset.QuadPart = section * (1<<30);
        size_section.QuadPart = file_size.QuadPart - section*(1<<30) >= (1<<30) ?
                                1<<30 : file_size.QuadPart - section*(1<<30);
        if(start_offset.QuadPart*(1<<30) + size_section.QuadPart >= file_size.QuadPart)
           done = true;
        section++;

    MutexRelease(mut);

    char *base_p = MapViewOfFile(mHandle, PAGE_RDWR, start_section.lowPart, start_section.highPart,
size_section)
    int counter = 0;
    while(counter < size_section.QuadPart) {
      char *moving_p = base_p;
      char tmp_value = *moving_p;
      tmp_value = (tmp_value + C) % 256;
      *moving_p = tmp_value;
      counter++;
      moving_p++;
     }
    UnmapViewOfFile(base_p);
   } // end if
   else { // we are done
      MutexRelease(mut)
      break;
  }

}

int main() {
  fHandle = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL)
  mut = CreateMutex(NULL, false, NULL)
  GetFileSizeEx(fHandle, &file_size)
  HANDLE mHandle = CreateFileMapping(fHandle, READ | WRITE, lg.LowPart, lg.HighPart);
  HANDLE thread_handles[N];
  for(int i = 0; i < N; i++) {
   thread_handles[i] = CreateThread(thread_func, NULL, NULL...)
  }
```

Exercise 5

Write a C++ multi-thread piece of code that is able to:

1. Run N threads, each one creating a random number and storing it into a stack, until one of them generate the value 0. When the value 0 is generated all threads stop.
2. Run 2 threads, one for even and one for odd numbers. They all pick one element from the stack and, if the value is of their assigned type (even or odd), they save it into a shared vector.

3. Run 1 thread that, when the stack is empty and the vector full, prints the entire vector.

Let the synchronization be coherent with the task each thread has to carry.

Note: if you do not remember the exact syntax of a C++ class, write down a version that likely resembles what you remember together some C++ comment, briefly summarizing what you were willing to use.

Solution

`You missed any notification to avoid the while(true)`

```
#include...
using namespace std;

void random_creation(StackClass<int> &v, mutex &m, condition_variable &cv) {
  static bool end = false;

  while(!end) {
    m.lock();
    if(!end) {
      int n = randomgenerator();
      if( n != 0) {
        v.push(n);
      } else {
        end = true;
      }
    }
    m.unlock();
  }
}

void even(StackClass<int> &v, mutex &m, vector<int> even_odd) {
  int val;
  while(tre) {
    m.lock();
    val = v.pop();
    if(val % 2 == 0) { // Even
      even_odd.emplace_back(val);
    }
    m.unlock();
  }
}

void odd() {
DUAL HERE.
}

void empty_full(StackClass<int> v) {
  while(true) {
    if(v.empty() || v.full()) { //if it is empty or full
      for(int i = 0; i < v.size(); i++) {
        cout << v[i] << " ";
      }
    }
  }
}
```

```
int main() {
  int value = 0;
  mutex m; // I used the same mutex because I have to be sure that the operation on the stack are performed in
mutual exclusion
  vector<thread> threadPool;
  StackClass<int> v;

  for(int i = 0; i < N; i++) {
    threadPool.emplace_back(thread(ref(random_creation), ref(v), ref(m)));
  }

  threadPool.emplace_back(thread(ref(even), ref(v), ref(mEvenOdd), ref(m)));
  threadPool.emplace_back(thread(ref(odd), ref(v), ref(mEvenOdd), ref(m)));

  threadPool.emplace_back(thread(ref(empty_full), ref(v)));

  for(int i = 0; i < N+3; i++) {
    threadPool.join();
// There is no termination condition because I didn't understand if 0 shoul terminate only the generation
threads
// or all the others so these join will not be performed all (while(true in even, odd and empty_full))
// to avoid that the static bool end in thread generation should have done global not local, so the same
termination status (a random 0 generated) terminate all threads and
// not only the N generations thread.
  }

  return 0;
}
```

Exercise 6

Write a small C++ program to accumulate a matrix of integers by rows (so that the final results will be a vector where, at each index, is stored the sum of all elements on the columns for the indexed row).

The main program must print the vector only when ready (you can omit the print but not any synchronization primitive).

Make the task able to start the computation only when its row is fulfilled by the user (after their creation).

Note: if you do not remember the exact syntax of a C++ class, write down a version that likely resembles what you remember together some C++ comment, briefly summarizing what you were willing to use.

Solution

```
#define N 3
#define M 4

int matrix[N][M];
int v[N];

void accF(future<int>& f){

  int i = f.get();
  v[i] = 0;
  for(int j=0; j<M; j++){
    v[i] += matrix[i][j];
  }

}

int main(){

  promise<int> p[N];
  future<int> f[N];
  future<int> fu[N];
  for(int i=0; i<N; i++){
    f[i] = p.get_future();
    fu[i] = async(launch::async, accF, ref(f[i]));
    for(int j=0; j<M; j++){
      cout << "Insert number: ";
      cin >> matrix[i][j];
    }
    p.set_value(i);
  }

  for(int i=0; i<N; i++){
    cout << f[i].get() << '\t';
  }
  cout << endl;


}
```