Lab 2.1 (Synchronization with semaphores)

Implement a concurrent program in C language, using Pthreads, which creates two client threads, then it acts as a server.
A client thread loops reading the next number from the binary file (**`fv1.b`** and **`fv2.b`**, respectively), and storing it in a global variable **`g`**. Then, it performs a signals on a semaphore to indicate to the server that the variable is ready, and it waits on a semaphore a signal from the server indicating that the number has been processed (simply multiplied by **`3`**), finally, it **prints the result and its identifier**.
The server loops waiting the signal of the clients, doing the multiplication, storing the results on the same global variable **`g`**, and signalling to the client that the string is ready to be printed.

The main thread waits the end on the threads, and prints the total number of served requests.

Lab 2.2 (Semaphores with timed wait and signals)

Implement a concurrent program in C language, using Pthreads, which generates two threads, and then wait for their completion. The first thread **th1** must:

- Sleep a random number of milliseconds t in range **1** to **5**
- Print "**waiting on semaphore after t milliseconds**"
- Wait on a semaphore **s**, initialized to **0**, no more than **tmax** milliseconds (**tmax** is passed as an argument of the command line),
- Print "**wait returned normally**" if a **sem_post(s)** was performed by the second thread **th2** within **tmax** milliseconds from the wait call (or if the **sem_post** call is performed by **th2** before the **sem_wait** call performed by **th1**,
    - otherwise, it must print "**wait on semaphore s returned for timeout**".
- Terminate

The second thread **th2** must:

- Sleep a random number of milliseconds **t** in range **1000** to **10000**
- Print "performing signal on semaphore **s** after **t** milliseconds"
- Terminate

For the first thread, you must implement and use a function with prototype
int wait_with_timeout(sem_t *S, int tmax),
which, using the appropriate system calls for the management of semaphores and **SIGALARM** signals, allows you to define the maximum time that a process can be blocked on the semaphore **s** queue before it is unblocked, and can proceed regardless of a call to **sem_post(s)**.
Function **wait_with_timeout** returns a flag set to **1** if a timeout occurred.

For sleeping less than a second use **nanosleep** system call (**man nanosleep**)

```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);
```

Lab 2.3 (Semaphores with sem_timedwait and signals)

Implement **Lab 2.2** exercise using **sem_timedwait** (see example in course page)