I. Description

In this assignment you will experience the design and implementation of asynchronous I/O operations and I/O scheduling for a disk device driver. You are required to program some I/O functions starting from an already provided code basis.

II. Provided code

You are provided the following code:

1. **A device driver for a disk.**
   The disk is simulated as a file. The following functions are provided:

```
/*  Returns total number of device sectors or a -1 for errors. */
int getNumSectors(int dd);

/* Reads into buffer one disk sector starting from a device offset
 * The device offset is expressed in number of blocks
 * Returns the number of read sectors or a negative number in case of errors*/
int dev_read(int dd, char *buffer, int offset );

/* Writes from buffer one sector starting from a device offset
 * The device offset is expressed in number of blocks
 * Returns the number of written sectors or a negative number in case of errors.*/
int dev_write(int dd, char *buffer,  int offset );

/* Opens a device: name is the file simulating the block device
 * Returns a device descriptor or -1 in case of errors. */
int dev_open(char *name);

/* Releases a device. It waits that all operations on device are finished.  */
int dev_rls(int dd);
```

2. **A request declaration**

```
struct aio_rq{
  int dd; /* device descriptor */
  int tid; /* thread id */
  int offset; /* disk offset */
  int type;  /* type =   READ_RQ/WRIYE_RQ */
  char *buffer; /* buffer to be read or written */
};
```

3. **A queue library.**

```
/* enqueues an element */
struct queue* enqueue( struct queue*, void * data);
/* dequeues an element */
void* dequeue( struct queue*);
/* returns 1 if the queue is empty and 0 otherwise*/
int queue_empty ( struct queue* s );
/* If it finds the data in the queue it removes it and returns it. Otherwise it
returns NULL */
void* queue_find_remove(struct queue* s, void * data );
```

4. Several main programs

- **main1.c :** Synchronously writes a sector to a device, reads it back and checks for correctness.
- **main2.c** : Demonstrates and tests enqueuing requests to a queue and removing them.
- **main3.c** : Asynchronously writes a sector to a device. Overlaps computation with I/O. Synchronously reads it back and checks for correctness.
- **main4.c**: Asynchronously writes a sector to a device. Overlaps computation with I/O. Asynchronously reads it back and blocks waiting for the operation to finish. Checks for correctness.
- **main5.c**: Asynchronously writes a sector to a device. Cancels the operation.
- **main6.c**: Asynchronously writes several sectors to a device. Overlaps computation with I/O. Waits for all operations to finish.

5. A utility `create_disk_image` for creating disk images:

- **It requires two parameters: a file name which will store the disk image and the number of blocks of the disk**
- **Example which creates an disk image with 1000 blocks:**
  `$ create_disk_image` disk1.img 1000

III. Requirements

In this assignment you have to fulfill the following requirements:

1. Implement in dev.c the following functions, whose signature is already defined in dev.h:

```
 /* Starts a read operation. Returns 0 if successful and -1 for errors */
int async_read(struct aio_rq *r);

/* Starts a write operation.  Returns 0 if successful and -1 for errors */
int async_write(struct aio_rq *r);

/*  Checks the status of the r request.  Returns 1 if operation has finished, 0
otherwise  */
int async_status(struct aio_rq *r);

/*  Blocks and waits until the current operation finishes. */
int async_wait(struct aio_rq *r);

/* Cancel an already started operations. If the operation has been already
scheduled it waits to finish. */
int async_cancel(struct aio_rq *r);
```

2. The functions must allow to overlap I/O with computation. In order to do so the functions `async_read` and `async_write` must immediately return and the I/O must be done in background by an disk scheduler (Hint: examples of overlaping I/O and computation are given in main3, main4 and main6).
3. The disk scheduler must be implemented as POSIX thread, which is to be initialized once when the first driver function is to be used.
4. The disk scheduler has to implement the FCFS policy (Hint: The scheduler can use two queues: one for the requested operations and one for finished operations.)
5. When no request is available the scheduler must be blocked (***no*** busy waiting).
6. Use mutex and condition variables for synchronization.

7. When the assignment is finished the programs main3 to main6 must be correctly running. You need as well to write additional test programs in order to insure correct operation.
8. You have to describe in a report:
    - your solution
    - compare your solution with the provided code
    - contrast your solution with disk scheduling and asynchronous I/O theory (Silberschatz Chapters 12 and 13)
    - correctness testing including the tests you have performed in order to assure that your solution is bug-free
    - problems encountered
    - personal conclusion