CS 460 – Programming Assignment 3
Implement Asynchronous I/O
(Due midnight Friday 3/12/2015)

## 1. Overview

In assignment 2, you have implemented a scheduler that allows multiple user-level threads to run within a single kernel thread, the main thread. The threads that we tested, however, do not support blocking I/O efficiently yet. Blocking I/O means when a thread does an I/O operation, it will be blocked, gets swapped out for the next thread to run. Since all our user-level threads run on a single kernel thread, if a user-level thread does blocking I/O, the kernel thread becomes blocked (and gets swapped out for other kernel threads to run), thus all the user-level threads within our kernel thread will be blocked. What we really want is to have the blocked user-level thread swapped out for another user-level thread (within our current kernel thread) to run.

In this assignment, you will implement a wrapper that emulates the **read** system call. **read** is used to perform blocking I/O on a file descriptor. Our wrapper should mirror the semantics of **read** as closely as possible, ***while only blocking one user-level thread***.

The assignment must be done on lab machines (lx.encs.vancouver.wsu.edu) using your encs login account. This assignment can be done individually or with a team of two.

Example binaries are also included so you know what the output programs should be.

## 2. Instructions

### 2.1. Background research

It is crucial for you to read, understand, and interact with a potentially unfamiliar interface. Before continuing, thoroughly read the following Linux man pages:

aio(7)
aio_read(3)
aio_error(3)
aio_return(3)

You may find it helpful to brush up on the syntax and semantics of the synchronous read call:

read(2)
lseek(2)

### 2.2. Implementation and testing

Login to lx.encs.vancouver.wsu.edu and copy /encs_share/cs/class/cs460/proj3 to your home directory. The directory includes

- Makefile
- Thread_switch.s and thread_start.s: assembly files for context switching
- queue.h & queue.c : ADT for thread library
- scheduler.h: header file specifying scheduler functions and thread states
- scheduler.o: a correct object file for the scheduler. If you are confident about your scheduler in assignment 2, you can replace this one with yours.

- ***async.c: You are to complete this file***
- **main*.c**: **You are to complete these files**. Main test files for testing correct blocking behavior and correct read semantics.

**You are to complete the async.c file and write programs (main*.c) to test your code.**

### 2.2.1. Implementation

Write a function, read_wrap. It should have exactly the same signature as read:

```
ssize_t read_wrap(int fd, void * buf, size_t count) {
  // your code here!
}
```

Use the AIO interface to create an AIO control block and initiate an appropriate asynchronous read. Yield until the request is complete.
**read_wrap** should have as close to the semantics of read as possible. That is, it should return the same value that read would have, and put the same result into *buf* that read would have.
Furthermore, if the file is *seekable*, **read_wrap** should start reading from the current position in the file, and then seek to the appropriate position in the file, just as read would have. This is arguably the most difficult part of this assignment, since **aio_read** does not seek automatically.

Your **scheduler.h** file and **async.c** should *#include* the following libraries (there are some strong hints here about which functions you might need to use):

| | |
|---|---|
| `<aio.h>` | For `struct aiocb`, `aio_read`, `aio_error`, `aio_return`. |
| `<errno.h>` | For `EINPROGRESS`. |
| `<unistd.h>` | For `lseek`, `SEEK_CUR`, `SEEK_END`, `SEEK_SET`. |
| `<string.h>` | For `memset`. |

### 2.2.2. Testing

Testing is a significant portion of this assignment. You will need to verify two things: correct scheduling behavior and correct reading semantics.

**Correct Scheduling Behavior (main_test_scheduling_behavior.c)**
You must write a test that proves that read_wrap correctly only suspends the current thread, allowing other threads to continue during the I/O.
The easiest way to do this is to re-use your test code from assignment 2, but add a thread that reads from standard input (file descriptor 0), which should cause the thread to wait until the user inputs something at the terminal and presses enter. Ensure that your computations proceed while the reading thread is busy-waiting, and that the thread properly resumes when input arrives.

**Correct Reading Semantics (main_test_reading_semantics.c)**
Furthermore, it is your responsibility to ensure that your implementation of **read_wrap** mirrors the semantics of read as closely as possible. This means that you must test that **read_wrap** correctly returns the same number of bytes as read, and that **read_wrap** returns an error whenever read returns an error.

You must also test that **read_wrap** has the same seeking behavior as read. Standard input is not *seekable*, so you will have to run tests on a real file. See open(2) for more on how to create a new file descriptor from a path name. Test code that uses open must *#include <fcntl.h>*.

## 3. Submission guidelines and grading criteria

### a. Submission

Please provide sufficient comments for your code. Clearly indicate if a step is incomplete.

Please zip all files

- A report file that includes tests and test result analysis
- The code, and
- A README file that describes how to run your code.

Submit the zip file on prog3 dropbox on Angel (Those having difficulties accessing Angel can send the file to the TA's email address: solongo.munkhjargal@email.wsu.edu)

The code is expected to run on the lab machines (lx.encs.vancouver.wsu.edu).

### b. Grading criteria

| **Criteria** | **Points (/100)** |
|---|---|
| **read_wrap.c** | 40 |
| Tests for Correct Scheduling Behavior | 30 |
| Tests for Correct Reading Semantics | 30 |