Vidhya Lakshmi Venkatarama
CS533: Concepts of Operating Systems
Assignment 3: write-up

a) The read_wrap method has been implemented with the help of aio_read() method that has been described in the linux man pages. The method first creates a structure for an asynchronous IO block and initializes its value with zeros. It then fills in all the required value in this block (file descriptor, buffer, number of bytes to read, the offset and the method to be notified on completion of IO (i.e no notification because we are using polling). The offset is set using the *lseek()* function described. The asynchronous block structure thus created is passed as an argument to the function aio_read() and the value returned is saved in the variable *readRet* . A 'zero' value of readRet would imply that the read request was successfully queued and a '-1' implies that there was an error in queuing the read request. Since the approach of notification we are using is polling, on a successful read enqueuing, we would check if the read is in progress (when aio_error returns EINPROGRESS) and yield every time this is true. The process is checked for errors while reading and on successful completion (when aio_error returns '0') the asynchronous block is returned with aio_return.

The implementation is tested by calls to both synchronous and asynchronous read functions and in different order. The output files have been attached.

- Output1: The order of forking thread was

```
 i.  thread_fork(read_async,std_input);
 ii.  thread_fork(read_sync,std_input);
 iii.  thread_fork(read_async,file_input);
 iv.  thread_fork(read_sync, file_input);
 v.  thread_fork(read_async, file_input);
 vi.  thread_fork(read_sync, file_input);
```

In the output observed, considering a buffer size of 5000, the file input read v. was forked after the synchronous reading iv. Here the read v. seeked through the file till 5000$^{th}$ character and then started reading.

- Output2:

```
 i.   thread_fork(read_async,std_input);
 ii.   thread_fork(read_async,file_input);
 iii.   thread_fork(read_async, file_input);
 iv.   thread_fork(read_sync,std_input);
 v.   thread_fork(read_sync, file_input);
 vi.   thread_fork(read_sync, file_input);
```

In this order of thread forking, both the asynchronous read were placed before the start of synchronous read and the initial offset was 0. Hence, the read did not seek through for asynchronous read. However, the second synchronous read of the same file (vi.), still started from 5000$^{th}$ character.

b)

1. In a cooperative multithreaded environment, not yielding in the critical section would ensure synchronization if it is running on a uniprocessor (which is true in our case). In our implementation of the Asynchronous I/O, we are performing only read operations and no writes. Considering the readers-writers problem, this could mean that yielding while reading (because it is asynchronous) would still protect shared data (i.e. the data file) from being modified asynchronously. When, in our implementation, we include seeking into a file, such that the current position in a file is shared by all threads accessing it (because we are not reading different file streams but the same file by using the same file descriptor). This means that the asynchronous file read with seeking allowed is a critical section. Hence yielding here would lead to synchronization problems. However, we have implemented the asynchronous IO using different instances of same file descriptor, so different file streams and hence the critical section is still protected when we yield, unless the concept of interrupt comes in.
2. A scheduler as described in the question could maintain a separate data structure to handle threads blocked for IO. The state of these threads would then be BLOCKED. On blocking such a thread, the scheduler could store in an instance of the data structure, the information related to the thread and a unique ID (may be the threadID). The scheduler can then pass the information needed to perform the IO (i.e. the file descriptor, buffer size, buffer, etc) and the unique identifier to the handler. The scheduler could then schedule other runnable threads and whenever the scheduler runs, could poll to see if the IO is complete. The handler on completion of IO would manipulate the condition on which the thread is blocked (completion of IO). As this happens, the scheduler would associate the completed IO with the corresponding thread, remove that thread from the queue of condition variable, change its status to READY, enqueue it on the ready list. Since these signals are asynchronous, they could occur at any time during program execution and potentially preempt the scheduler. This implies concurrency. The critical sections for race conditions could be when the scheduler is accessing/modifying any of the data structures (ready list, blocked list, etc.). These data structures then need to be protected by synchronization primitives. There could be cases where the same file is accessed by different threads. This also is concurrent and needs synchronization. A deadlock can happen in such a situation through many ways, (two threads accessing files (or resources) and leading to circular dependency, etc.). In the scheduler described above, there is a potential for race conditions and dead lock when 2 signals occur at the same time and they try to modify the same data structure in different orders.