Please Note:
*Your solution **must** compile and run on the simula machines.*

Programming Task

Your task is to implement a memory manager that manages page reads and writes from multiple client threads.
In the first programming task the memory manager keeps all the pages in a memory buffer large enough to fit all the pages and the server processes a list of requests and sends them to clients to log.
In the second programming task a user is allowed to specify a memory buffer size limit. The memory manager needs to place some of the pages on disk in order to cope with the buffer size limit. To do this the memory manager uses the least recently used (LRU) page replacement algorithm in order to choose which page to evict from memory when the memory is full.
In the third task you extend the solution for task 1 so that requests are sent from the client to the server instead of all the requests originating in the server.

Please look at the AssignmentDiagram file in the assignment folder to gain a better understanding of the assignment requirements. Sample input and output files are also provided in the Sample Files file.

Please do not use sleep anywhere inside your code. Or something that simulates a sleep system call. Eg. while (1) { }. However you are allowed to use the pause(), pthread_cond_wait() system calls and their Java equivalents to block a thread or process.

Please submit separate source files for each of the three tasks.

Task 1 [35]:

There should be 1 server thread and N client threads, where N is supplied by the user as a **command line argument**. The server opens a file called "all_requests.dat", the file has the following format:

```
<client id><space><read/write><space><page id><space><contents of page>
<client id><space><read/write><space><page id><space><contents of page>
...
…
```

client id – refers to the id of the client thread which is between 0 and N – 1. Note this is not the thread id that is assigned by the system.

read/write – refers to whether the request is a read or write request.

page id – refers to id of the page being requested.

contents of page – this field only exists if the request is a write request. It contains the contents of a page that is to be written into the server which is a sequence of alphabet characters (a-z, A-Z) with no spaces or anything else in between and has a **maximum size** of **4096 characters** (size of page for most machines).

Here is a sample "all_requests.dat" file. In this example there are 2 clients with ids of 0 and 1 respectively. Note the read requests do not have any associated page contents.

All_requests.dat file:

```
0 read 20
1 write 20 helloKKAAA
0 read 1
1 read 21
0 read 20
0 write 8 peter
```

The server thread also reads from the file called "init_buffer_pages.dat" which contains the initial contents of ALL the buffer pages (**you can be sure read and write requests are to pages existing in this file**). It has the following format:

```
<page id><space><contents of the page>
<page id><space><contents of the page>
.....
.....
.....
```

Here is a sample init_buffer_pages.dat file:

```
1 abcdefghigkhhLLK
2 ABjjjjjssksksksskksk
20 hellohowareyou
21 HHBGHH
8 JHjhjjsjjkwje
```

The program should do the following:

1) The server thread starts and reads in the init_buffer_pages.dat file to initialise the memory buffer.

2) Create N client threads, where the number N is taken from the command line.

3) The server thread reads from all_requests.dat file to get the page read or write requests. **The server must process these requests in order from the 1st to the last.** For example the 2nd request on the list must be processed before the 3rd request.

4) Each client thread should keep a log of the pages that it has read inside a file called "client_log_*n*" where *n* is its thread id. **The pages read must be written into the log file in the order that they are received from the server. Please note you only need to log read requests. In the case of a write request the server does not need to contact any of the clients.** This will let us know if server is processing the requests in the correct order. The beginning of each entry should contain the page id followed by the page contents. Here is the format:

<page id><space><contents of page>
<page id><space><contents of page>
.....
.....
Here is an example (note in the example below the page 20 has changed between the first and last reads):

20 hellohowareyou
1 abcdefghigkhhLLK
20 helloKKAAA

5) Please note that the server must wait for the current client thread to have finished writing its log entry before getting the next thread to log its read request.

6) When all the clients finish their requests the server thread must end and cause the entire process to end.

Please note we will be testing your program based on if client log files contain the correct data.

TASK 2: [20 marks]

Extend task 1 so that it now supports a memory buffer of limited size. Provide the same functionality as task 1 except now make the server work with a limited buffer size.

Like task 1, the server is the only thread that can access the buffer directly.

1. Allow user to specify a 2nd command line argument which specifies the buffer size in terms of the number of pages. Let that number be B.

2. The server reads the first B pages from the init_buffer_pages.dat file into the buffer. It then copies **all the pages** into a **store file**. You can organize the store file anyway you want as long as it stores all pages including those not in memory. I recommend that you use a binary file to do this since that allows you to do random seeks on the file. Assume all pages have a fixed size of 4096 characters.

3. When a page request for a **non-buffer** resident page occurs one page that is currently in the buffer must be evicted from memory to make room for the newly requested page to be loaded from the store file. The page replacement policy you need to implement is the least recently used (LRU) policy.

4. Once a page to be evicted has been chosen then the program must decide whether it should be flushed to the store file or discarded.
   a) If the page has been dirtied (modified) since its last load from disk then it should be flush to the store file on disk.
   b) Otherwise it should be discarded without flushing to the store file.

5 The server needs to keep a log of the sequence of page evictions (pages either discarded or flushed to disk) in a log file called "server_log.dat". The entries in the log file are **in the order that they** were evicted. Looking at this file I will be able to tell if you have implemented the LRU page-replacement policy correctly.

The server_log.dat file should be in the following format:

```
<page id><space><contents of page>
<page id><space><contents of page>
...
...
```

For example:

```
20 abscde
10 KJlkljf
29 jjkjkKKH
....
```

Task 3 [25 marks]

Extend task 1 (not task 2) so that the requests are issued by the clients instead of the server. The server now **do not** use the "all_requests.dat" file. Instead every client thread reads from a different input file called "client_requests_*n*.dat", where *n* is the thread id. Note this is **not** the thread id that is assigned by the system. Note thread id goes from 0 to N-1.
The client request files have the following format:

```
<request no.><space><read/write><space><page id><space><contents of the page>
<request no.><space><read/write><space><page id><space><contents of the page>
```

Apart from request no. field the other fields have the same meaning as in the "all_requests.dat" file in task 1.

Note request no. is a number starting from 1 2 3 ... to the total number of requests. **You can assume that when you combine all the client request files that all request**

number starting from 1 to the total number of requests exists. Your job is to process all the requests in **ascending request number order**. Also note the request numbers within each client request file are always sorted in ascending order.
Here is a sample pair of client request files. In this example there are 2 clients.

client_requests_0.dat file:
1 read 20
4 write 21 hellohowareyou
5 read 21

client_requests_1.dat file:
2 read 21
3 write 20 petermaryjohn

The program should do the following:

1) The server thread starts and reads in the init_buffer_pages.dat file to initialise the memory buffer (same as task 1).

2) Create N client threads, where the number N is taken from the command line (same as task 1).

3) Each thread reads from its own client_request_*n*.dat file to get the page read or write requests. The requests are sent to the server in the order that they appear in the file. **The server must process these requests in ascending request number order.**

4) **The requests are blocking requests,** that is if a request from client x is to read a page client x should not send the next request until the server has provided the requested page to the client x. Similarly for a write request the client x should not issue the next request until it has received confirmation from the server that its write has completed. Requests from different clients can be issued concurrently.

5) Each client thread should keep a log of the pages that it has read inside a file called "client_log_*n*" where *n* is its thread id. **The pages read must be written into the log file in the order that they are received from the server.** (same as task 1)

6) a) **The server must process the requests in ascending order according to request number specified in the client_requests_*n*.dat files.** For example request number 2 must be processed before request number 3, although request number 3 arrived at the server before request number 2. This can happen since the thread issuing request number 3 may have executed before the thread issuing request number 2.
   a) **The server is NOT allowed to open the client_requests_*n*.dat file or the all_requests.dat file. The server must get the request number and all other**

request information from the client when the client sends its read or write request.

b) The clients are **NOT allowed** to send **all** their requests at once at the beginning. The client can only send a new request after its previous request has been processed. See point 4 for more details.

c) **Not following the above rules will result in very significant reductions in your mark (you may loose up to 100% of the marks for task 3).** The reason is if you don't follow these rules then this task will be almost the same as task 1.

7) When all the clients finish their requests the server thread must end and cause the entire process to end. (same as task 1)

Bonus Task 1 [10 marks]:

Do questions 3 except have multiple servers and multiple clients. Please read detailed specifications from BonusTask1.zip.

Bonus Task 2 [15 marks]:

Do task 3 but implement each client in a separate process instead of thread. Therefore there should be N client processes and 1 server process. The server process needs to have **N client communication threads** which handle the communication with the N client processes. Therefore each client communication thread handles communication for one particular client process.

Bonus Task 3 [10 marks]:

Do Task 1 using both POSIX threads and Java threads.

Bonus Task 4 [10 marks]:

Do Task 1 using both monitors and semaphores.

Programming Style [20 marks]:

Programming style will be judged on the following:

1. Correct use of synchronization techniques such as mutex, semaphores, monitors, etc. [15 marks]
2. Comments / functional decomposition [5 marks]

Total Marks:

Please note that the total marks for this assignment is capped at 100. If your marks add to higher than 100 then it will be reduced down to 100.

Submission Details

- **Please ensure that your name and student number is on every file that you submit.**
- Zip up everything in your project directory into one file.
- Submit your zip file via LMS

You can submit the same filename as many times as you like before the assignment deadline; the previously submitted copy will be replaced by the latest one.

Return of Assignments

Departmental Policy requires that assignments be returned within three weeks of the submission date. We will endeavour to have your assignment returned before the OSS exam. The time and place of the return will be posted on LMS.