Everything You Need to Know

Ensure you read: README file and comments in header files.

Each library in the codebase has a corresponding test (in the test directory). These tests are instructive, you can often work out what a library has to do and even some of how it should be implemented from its test.

The main codebase comprises over 20 (.c and .h) source code files (of which you are expected to complete the implementation in 7 files). In addition, there are over 20 source code files in the test directory. In a project of this size, it is possible that there are bugs. It is possible therefore that bug fixes will be released when you are working on the assignment. **This is normal.** In addition, advice, clarification, documentation and README files may be released after release of the initial codebase. Again, this is normal. In part it is done to allow you to concentrate on the parts of the project that are initially important. It may also be done to give you experience of how in a real project, things can change during development. The specification of the functions you have to implement will not change and no changes to the codebase will be released in the week before the assignment is due. **None of this is a reason to delay starting work on the assignment.**You are expected to start work straight away.

**You must not change function signature in header (.h) files**. Header files will only change if the module leader releases an update that changes them.

Contact the module leader if you need clarification of the project requirements or need help interpreting documentation or need help to solve programming problems. Please make sure you read the module g[uidance on asking effective questions](safari-reader://ncl.instructure.com/courses/24657/pages/how-to-ask-for-help) before requesting help.

If you think you have found a bug in the codebase, please use the form you can access from [the page about asking effective questions](safari-reader://ncl.instructure.com/courses/24657/pages/how-to-ask-for-help).

**The application/simulation**

As shown in the following figure, the project simulates an application in which a set of one or more producer processes creates jobs to submit (enqueue) to a FIFO (first-in, first-out) job queue for one or more consumer processes to dequeue and process. The job queue is in shared memory and in addition each process (producer or consumer) has a private persistent log of jobs that they have produced (if they are a producer) or consumed (if they are a consumer). Many common client/server applications can be modelled in this way as some combination of producers (clients) who submit work for consumers (servers).

The overall application proceeds as follows:

* Until they have produced a specified number of jobs, if the queue is not full, each producer process produces a job, enqueues the job on the job queue and logs the job in their private log.
* Until they have consumed a specified number of jobs, if the queue is not empty, each consumer process dequeues a job from the job queue, processes it and logs the job in their private log.

A job is represented by a tuple: (id, pid), where id is an id for a job generated locally by the producer that produced the job and pid is a process id for the producer. This means, assuming integrity constraints are maintained, a job should be globally unique (by the combination of id and pid). For simplicity, in the simulation, pids are not real process ids but ids allocated to each process at application start up. A startup script is provided that manages this allocation. In a real-world version of the application, both the job id and pid could be replaced by URLs of job specifications and for producers and consumers.

The C application code for the producer and consumers processes that drive the simulation is provided for you. Your task is to complete libraries that support the simulation (see Section 7). You are **not** writing the whole application (or all of the library code).

The overall integrity constraint is that producers should not produce duplicate jobs and each job should only be consumed once by one consumer. That is, no two consumers should consume the same job. If you complete the libraries specified in Section 7 then you will have a project that demonstrates whether this constraint holds for various different approaches to concurrency control.

Whether integrity is maintained is verified by checking that the state of log files after a simulation run satisfies the following conditions:

1. **Jobs are well-formed**: entries in logs are a five digit job id (including leading 0s, if necessary), followed by the character "p" (producer), followed by a two digit process id (including a leading 0, if necessary). For example: 00025p01
2. **Logs are sets**: no producer log contains duplicate entries and no consumer log contains duplicate entries
3. **No jobs are lost**: every job that appears in a producer log must have a corresponding entry in one of the consumer logs
4. **Consumers do not create jobs**: every job that appears in a consumer log must have a corresponding entry in one of the producer logs
   * Conditions 2, 3 and 4 mean that the union of producer logs equals the union of consumer logs. That is, for producer logs P0 to Pn and consumer logs C0 to Cm:  
     P0 ∪ P1 ∪ ... ∪ Pn = C0 ∪ C1 ... ∪ Cm
5. **There is no duplication of jobs between consumers.** Each job is consumed by one and only one consumer. That is, each pair-wise intersection of the set of consumer logs is the empty set:  
   * for each and every pair of consumer logs Ci and Cj, Ci ∩ Cj = ∅
6. **For busy waiting solutions, there is no duplication of job ids**: the intersection of job ids in producer logs is the empty set (the reason for this condition will be apparent when you look at the busy waiting applications)
   * for the set of job ids in each producer log (Pidi), the pairwise intersection with job ids in other logs is the empty set: Pjob\_idi ∩ Pjob\_idj= ∅

Further information about the overall simulation, how to run it and how to analyse the process logs, as well as how to run tests of libraries, will be provided in README files. This will include a script to do the above analysis of logs after each run of a simulation.

After completion of the assignment, and as part of your feedback, a working solution will be released. This will allow you to access all the learning outcomes even if you were unable to complete the  assignment correctly.

**Functions To complete**

You have to complete the implementation of the functions with a TODO comment in the following .c files:

* ipc\_jobqueue.c - wrapper functions for a queue of jobs that is stored in shared memory for IPC
* joblog.c - the implementation of functions to set up a log and log jobs to file
* jobqueue.c - the implementation of a queue of jobs based on a fixed-sized circular buffer/array
* mutex\_lockvar.c - the implementation of mutex functions that are based on busy-waiting on a "lock" variable
* mutex\_peterson.c - the implementation of functions for Peterson's busy-waiting solution to mutual exclusion for two processes
* sem\_ipc\_jobqueue.c - wrapper functions for a queue of jobs that is stored in shared memory and uses semaphores to maintain integrity constraints
* shobject\_name.c - the implementation of a utility function to generate names of shared objects

joblog.c, jobqueue.c and shobject\_name.c only require single process, standard C programming. **You should start work on these and run the tests for them straight away. Do not wait to start work on these files.**

ipc\_jobqueue.c, mutex\_lockvar.c, mutex\_peterson.c and sem\_ipc\_jobqueue.c assume some understanding of the IPC theory introduced later in the operating systems part of the module. However, **you may find it straightforward to implement the functions in ipc\_jobqueue without that background** (it uses IPC but you do not have to do any IPC programming to implement the required functions).

If you think it will help to reduce code duplication or simplify logic, you may define your own private helper functions in the above files.

Do **not** modify any files other than those listed or functions other than those specified by a TODO comment in the files listed. Comments in files clearly state which files you should edit and which functions in those files you should implement. A TODO comment means that you have to implement. A DO NOT EDIT comment means you do not edit the file, declaration or function with which it is associated.

The specification of the functions you have to implement is given in corresponding header (.h) files, i.e. see ipc\_jobqueue.h for the specification of the functions in ipc\_jobqueue.c.

README files, this assignment specification, comments and hints in the .c files etc. provide additional guidance and hints for completing the assignment.

**How to work and programming to test**

1. Compile the project and run the tests as instructed in the README file provided. You will see that tests fail. Your job is to complete the project until no tests fail.
2. Start with joblog.c, jobqueue.c and shobject\_name.c (in any order you wish) then do ipc\_jobqueue.c followed by the remaining 3 files
3. For each .c library file, after making changes, compile the project again and run the tests.
4. Keep working until all tests succeed for a given file library. You can run tests for all code or for an individual library file.