2. **Producer/Consumer Problem**
    a. **Pre-requisite**
        i. You must use local git repository to do this assignment.
            1. Create a folder in which you want to store all the code for this assignment.
            2. Run 'git init'
                a. Also run:
                    i. git config –local user.name <your name>
                    ii. git config –local user.email <your ddn email>
            3. Now you can start committing your code by:
                a. git add <your file name>
                b. git commit
                    i. Please enter a useful commit message.
                c. Please commit all your work at the end of the day or when a good milestone is reached.
                    i. Doesn't matter if your code is working yet or not.
                    ii. Here is an example of a good commit message: https://lwn.net/Articles/163807/
            4. I will be relying upon your git commit history to understand/investigate how well you did.
                a. You will have to make continuous progress and can't just copy code last day, and declare you are done.
            5. There are many tutorials on the net for basics of using git.
            6. In addition to creating a local git repo, you may also upload your assignment to github like Malkeet did.
                a. Please get in touch with Malkeet for git related queries.
    b. **Develop a solution to this problem.**
        i. A producer process creates data which is to be consumed by a consumer process.
        ii. A shared buffer of a given size is used to hold the data items.
            1. Producer puts the data into this buffer, and consumer takes data from this buffer. Sounds simple enough. But not so much. There are many questions?
                a. How will consumer know where in the buffer index to read the data from?
                b. How will producer know consumer has read the data in a buffer index and it can now store next data there?
                c. How will you stop producer from overwriting the data to a buffer index that consumer is not done reading yet? Or What if consumer reads from a buffer index that is not filled by producer yet?
                d. In other words, there are many problems that arise out of two processes using a shared buffer, this problem is about solving

them using various process synchronization or thread synchronization facilities available in a specific development environment.

2. Note that consumer must consume the data in first in first out manner (sequentially) i.e., the entry that the producer put in first will be consumed first by the consumer.

c. Requirements:
    i. Create a parent process that will create various children
        1. Producer – this is a child process.
        2. Consumer – this is a child process.
        3. Data generator – this could be a thread in parent or a child process.
            a. This thread will generate data (unique KEY) used by the producer thread.
            b. An optional command line argument is accepted which will specify how many KEYs to generate.
                i. Let's just call this number key_count for now.
        4. Verifier – this could be a thread in parent or a child process.
            a. This thread will verify that the producer consumer threads are working correctly i.e., all the data generated by producer is consumed by the consumer sequentially and there is no data corruption.
        5. Both the child processes (producer and consumer) must print statistics when they receive SIGUSR1 e.g.
            a. PID/Start-end address and size of the buffer/Start time of the process/For how long it has been running.
            b. How many entries produced if producer or consumed if consumer? (Since the process started).
            c. Current value of any counters/variables etc.... used (optional).
        6. Parent should accept separate optional command line arguments for producer and consumer (even floating-point value) specifying a delay in milliseconds to introduce after an item is produced or consumed – this helps us simulate cases where producer and consumer are operating at different speeds.
            a. Delay becomes active just one time when process receives SIGUSR2, so it is delayed as many times as SIGUSR2 is received.
    ii. A buffer of fixed size is shared b/w the two producer / consumer.
        1. Process must accept an optional cmd line argument specifying the size of this buffer in terms of number of entries.
            a. We use number of entries instead of buffer size as entry size could change in future, specifying number of entries allows the process to allocate a big enough buffer to store those entries.
            b. Note that this is different than the key_count mentioned earlier. Buffer will usually be much smaller than key_count.

            i. E.g., Size of this buffer could just be 1024 entries while overall data entries that needs to be processed could be say 1 Lakh (key_count).

iii. You will need to figure out a way to co-ordinate access to the shared buffer i.e., handle many problems that arise out of shared buffer access b/w two process.
    1. What if the producer tries to overwrite an already populated entry waiting to be consumed?
    2. What if the producer is writing the same entry that the consumer is reading?
    3. And so on…. There are many more synchronization issues.

iv. Your implementation must be efficient e.g., it should use locking instead of busy waiting.

v. The format of data created by the producer should be meaningful. This is just an idea – you can be more creative. Let's call this format data_entry_t for now.
    1. KEY: A randomly generated key from data generator.
    2. PID of the producer that generated it.
    3. Number of this items since process started running i.e., if this is the 11,000th item generated since process started it should contain 11000.
    4. Time when this item was generated (must be accurate up to nanoseconds).
    5. There should be a way to show these details (print them) to either stdout or a logfile.

vi. A bit more about how these threads are supposed to work together:
    1. There are 4 entities (thread/process) as I mentioned earlier.
    2. In the beginning, data generator will generate 'key_count' number of unique KEYs (see Section c.i.3.b) and store them to non-volatile storage.
        a. This storage could be a disk file or a database or any other method you can come up with as long as it maintains the sequence.
    3. After data generator has generated all the data, it will inform producer and consumer to start processing data.
        a. How will it inform? You people decide.
    4. Producer
        a. Producer will produce the data in same sequence as the data generator thread generated, you can't read random items and process them.
            i. Format of data produced is mentioned in c.i.v. Basically it will take KEY and add other details to it to create data_entry_t.
        b. Producer is free to read one key at a time or a group of keys at a time or all keys at a time and process them or if all the keys from data generator are in a disk file – it might just use mmap().
    5. Consumer
        a. Consumer will start processing data from producer.

   b. Also note that there needs to be a way for consumer to store or commit all the data (data_entry_t) it has consumed. This needs to be something persistent i.e., non-volatile storage.
      i. Consumer might decide whether to commit the data in one go at the end i.e., cache it or commit it after x number of entries are processed or commit each entry consumed.
         1. In IO terms, we refer to these concepts as write back and write through – just read about them.

6. After producer/consumer are done processing all the data in this session, they will stop, and inform the data verifier to verify all the data for this session.
   a. Data verify thread will read the data committed by consumer as well as data generated by data generator which producer used.
      i. It will verify KEYs in data_entry_t matches the corresponding KEY in data that was generated by data generator.
      ii. It will also verify other fields like PID of the producer.
      iii. Also make sure time field is always incrementing and so on.
      iv. Basically, come up with a way to verify producer / consumer are working correctly.
7. After this, we go back to data generator thread which will generate more data and cycle will continue.
   a. If parent received SIGRTMIN+1 signal any time before this point, then the cycle will end, and all the threads/processes should safely exit.
      i. Basically, sending this signal is a way of saying, whole program should quit after Step 6.
8. You must also implement something called as a watchdog. This watchdog process or thread will monitor producer/consumer (how?) and send a signal (which one?) that will cause core to be generated for all the processes involved if it detects that producer and consumer are stuck. By stuck we mean to say there is more work to be done but processes are deadlocked due to one reason or another and are not able to continue.
   a. You must setup OS environment in such a way that the corefiles are generated.
9. Logging
   a. There should only be informational logging on stdout, e.g.
      i. Starting with command line arguments…
      ii. Created child producer process with pid: %d
      iii. Created child consumer process with pid: %d
      iv. Data generator thread: Generating %d keys

        v. Data generator thread: Done. Notifying consumer to start processing data.
        vi. And so on.
    b. You should also implement a -d or –debug or –verbose cmd line option which will log to a given log file much more verbose logs that will help us debug.
        i. E.g., this would be pointers/buffer index/addresses/details of which entry process is working on/or logging that it's trying to acquire this or that lock before calling lock function etc.… and so on.
        ii. The contents are up to you to decide.

vii. A bit more about testing.
1. You will also need to do QA on your program, run it with various incorrect parameter etc.… and make sure it works or quits with invalid input error.
    a. This is just basic QA.
2. Writing test cases for your program
    a. These are just example; I know you folks are much more creative.
    b. Test Case 1: Slow producer
        i. Write a wrapper script that will start your program with the delay parameter mentioned in Section c.i.6.
        ii. Now send SIGUSR2 to the producer to slow it down.
            1. This will test the slow producer case.
            2. Expected result: We should not see any crash due to watchdog timer or the verify stage must not fail.
    c. Just like this, you can write various test cases to test out your code.

d. Questions you should be able to answer.
    i. Questions related to multi-process execution model/time-sharing etc....
    ii. Details related to parent/child processes.
        1. fork/exec/wait and related system calls.
    iii. What are signals?
        1. Several types of signals.
            a. What are the most used signals in Linux and their details e.g., SIGTERM/SIGKILL etc.....
        2. How a process can register handlers for these?
            a. sigaction () and related functions.
        3. How to send a signal to a process using command line or from inside a process.
        4. Which signal generate corefiles?
            a. Which OS settings needs to be changed for corefiles to be generated?

        b. Location of core files?

iv. Process synchronization facilities provided by Linux OS.
    1. What is the critical section?
    2. Mutex/Semaphore etc..... Why do we need them?
    3. You should know which functions to use as well as details of their implementation in user space and kernel space.

v. Atomic instructions
    1. What are they? Why do we need them?
    2. Can they be implemented just in software? Or do we need HW support?
        a. What HW instructions are used to implement atomic instructions?
    3. How are they used in the critical section?

vi. Shared memory
    1. What is it?
    2. Various Functions that help us create shared memory b/w two processes in Linux.
    3. How is this implemented by the OS (as by default two processes are not able to share their address space)?
    4. Do the pointers to shared memory in both processes have the same address? Or are they different?
        a. And why?