1. A) It's optimal for the prime factor program to spawn 2 threads. Because the user wants to run this program in the background it is best for the game task to utilized two cores. One for game logic and one for rendering as stated. Since the game is fast pace it is best not to share cores with other processes concurrently whereas the prime factor program can be run on a single core. At best utilize two cores, if not, a single core to run the prime factor program.

   B) For this particular circumstance I would optimize throughput over latency. To my intuition, game logic and rendering graphics for games only need to go as fast as the frame rate. Also seeing that the prime factor program is run in the background it is not in the imminent interest but it still requires a significant amount of computation. So we want to focus on optimizing how much work is getting done or how many jobs are getting done at once. In addition since we want gaming computations to run on independent cores we want to optimize for utilization to make sure that core is not being shared by the prime factor program.

2. The ABA problem occurs when dealing with lock free data structures, share memory and comes from the fact that pointers are spatially unique but not temporally unique. Although lock free data structures endeavor to do atomic modifications using compare and swap there is still room for interleaving between reading the expected value and the CAS. Therefore in the process of removing a node from a lock free data structure another thread can modify the structure but return to the CAS with reused pointer values but with different and with a high probability misleading values. One way to solve the ABA problem is by using 128 bit pointers. In situations like the above we want our CAS to fail even though technically it is the correct pointer value. We can distinguish these re-used pointers by having counters in the other 64bits and storing these freed pointers in a list. Obviously enough we want a machine that supports 128 bit pointers but also 128 bit compare and swap operations to go with this.

3. A) When using mutices we want to prevent reading when writing and writing when reading. Since we cannot implement the invariant that multiple threads can read simultaneously the best yet correct performance we can get my using mutices is to have single global mutex. Note that we have to check whether the buffer is full or empty with the lock held.

   push( ) : get the lock, check if full, write and increment pointer then unlock

   pop( ): get the lock, check if empty, read and increment pointer then unlock

   size( ): get the lock, temp = (write pointer – read pointer), unlock then return temp

B) Compared to using mutices, using reader- writer locks allow more parallelism due to the fact that reading can happen simultaneously among threads. However, since we are incrementing the pointer even when reading it's not a pure read operation. Thus we would have to make use of CAS when calling pop from multiple threads simultaneously and when calling size.

push( ) : get write lock, check if full, write and increment pointer then unlock

pop( ): get reader lock, check if read pointer is less than write pointer, CAS on read pointer, then unlock

size( ): ideally you want both reads and writes not happening while we compute for size because reading in this case will affect the size. Therefore, setting a write lock when calling size ( ) seems like the way to go.

C) As explained above CAS is necessary when trying to use reader-writer locks. So by allowing more concurrent reads by using reader-writer locks CAS does make it more performant. However, trying to implement a lock free buffer by only using CAS seems challenging. We have to have a tool that stops any threads trying to read when there is no more items to read but it seems to me that there is no atomic way to check this condition.

4. A) The problem here comes from the fact that instructions of thread 1 are not atomic so there could be an interleaving of checking the if statement, setting proc_info to a null pointer then fput proc_info. One simple fix is to put them inside a critical section with a mutex. That way each thread will happen atomically in what ever order. Another way is to cache thr->proc_info before the if statement and fput that cached value. There could still be an interleaving but since we are not dereferencing the shared variable directly it solves the issue.