1. Using CADE Labs,
    a. `g++ -c main.cpp`
    b. `nasm -felf64 myAsm.s`
    c. `g++ -o myProg main.o myAsm.o`
    d. `./myProg`

main.cpp

```cpp
#include <iostream>

extern "C" {
    long fib(long num);
}

using namespace std;

int main() {
    cout << fib(6) << endl;
    return 0;
}
```

myAsm.s

```
section .text ; code section
global fib ; linker needs to know about the symbol "fib", nm -a main.o

fib: ; fib logic section
    cmp rdi, 1 ; if(n == 1)
    je returnZero ; return 0
    cmp rdi, 2 ; if(n == 2)
    je returnOne ; return 1
    mov rsi, rdi ; copy n to scratch register 1
    mov rdx, rdi ; copy n to scratch register 2
    sub rsi, 2 ; scratch register 1 = n - 2
    sub rdx, 1 ; scratch register 2 = n - 1
    mov rdi, rdx ; pass in n - 1 to call fib
    call fib ; recurse n - 1
    mov rdx, rax ; copy return value to register 1
    mov rdi, rsi ; pass in n - 2 to call fib
    call fib ; recurse n - 2
    add rax, rdx ; add n - 1 + n - 2
    ret

returnOne:
    mov rax, 1
    ret

returnZero:
    mov rax, 0
    ret
```

2. A process can be defined as the isolated execution of a program which has independent resources such as virtual memory space, process identifier etc. A process can have child processes. Processes are managed by the PCB (Process Control Block). The process doesn't share data across other processes unless explicitly forced to do so.

While threads are a part of processes that exist for concurrent programming that allows the programmers to use parallelism to optimize their code execution. A process is started with a single thread but can be populated with multiple threads which run in parallel. A thread shares the resources of the process and therefore isn't isolated or has independent resources. A thread can share data across other threads as they belong to the same process.

Process (thread) scheduling is a policy which is followed by the process manager to dequeue and queue active processes. In other words, when the process manager removes a running process and selects another process to be run on the CPU is called process scheduling.

A system call is a way in which a program communicates with the kernel of OS to request a service such as process control, file management etc. It is important because it allows the programmer to access a resource from the OS. It affects scheduling because there are system calls that govern process control. For example, fork(), wait() and exit() tells the kernel to create a process, wait for a process and end the process respectively.

Context switching is the process of storing the state of a process so that it can be restored at a later stage. System calls allow to save the state of the process and the process scheduler then follows a given strategy to keep the programs running efficiently.

3. Spatial locality means that all the instructions that are a close to a recently executed instruction spatially (storage/space), are likely to be executed sooner than the process that are stored far away. In other words, if a storage location was accessed recently, then it is likely that nearby storage location will be accessed soon. It helps in faster access of subsequent memory.

Temporal (time) locality means that a recently executed instruction is more likely to execute very soon. In other words, if a memory location is accessed recently, it is likely that the same memory location will be accessed again soon. For example, if a memory location is used often, it can be moved to the cache memory on the CPU to optimize latency and performance.

It is important to understand because it allows the programmer to optimize latency and throughput of their software. It allows the programmer to optimize the communication between the CPU and the memory. If the CPU will want to use some data from the memory, with the help of locality of reference, it can help knowing which data may be needed again and therefore should be kept in the cache memory. When a CPU wants to access data, it first checks if it's present in the cache memory and if it will be reused again. If it finds it, it will be very fast, but if not, the cache memory will request the data from the RAM and then store it in the cache memory if it is closer to the location that was used since it can mean that it is likely to be used again. In parallel processing, the latency to access a memory location may depend on the location of that address regarding its distance from the memory location that is being currently accessed. It is important to exploit locality of reference in parallel processing because the application is limited by the latency of the memory. It can help a programmer to understand how locality works, and how they can access data across different threads to get the best performance out of it by predicting the data that is in cache memory based on their data structures. For example, arrays are good for locality, while hash tables aren't. Running multiple threads that process the same

data will induce temporal locality therefore the data will be transferred to the cache memory to reduce latency. But mutexes and locks will cause the parallelism to slow down due to time dilation that is caused to avoid corrupting the data.

4. Physical memory can be defined as a reference to the actual memory (RAM) modules that are present on the motherboard. On the other hand, virtual memory can be defined as a memory management technique used by the computer to allow users to access memory larger than the physical capacity, therefore increasing the potential of a system. Physical memory is way faster than the virtual memory because virtual memory is stored on the hard drive and its performance differs across different types of hard drives such as NVMe SSD, PCIe SSD, SATA SSD, SATA HDD etc. The physical memory is limited by the size of the RAM modules, whereas virtual memory is limited by the hard disk space allocation. Physical memory can directly access the CPU while virtual memory cannot.

Paging is a memory management scheme that allows the physical address space of a process to be non-contiguous. The memory management unit (MMU) on the CPU handles paging. The physical address space is divided into fixed sized blocks called frames, while the logical address space is split into fixed sized blocks called pages. The page size is always equals to the frame size. This allows easy remapping from one memory to the other, otherwise this could be huge hit on performance. A 32-bit address can allow access to 2^{32} different memory addresses (4GB of RAM) at a time. But a 64-bit address can give access to 2^{64} memory addresses which is exponentially more than 4 GB of RAM. It basically means that more than 4 GB of physical memory can be easily handled and accessed with full performance.

A virtual address space is a range of virtual memory addresses made available to a process. It is a contiguous block of memory made available to a process. A typical virtual address will start with a low address and will go up to the highest memory address provided by the VAS. The addresses can be divided into different section. From lowest to highest we have, Reserved Space (space reserved for the OS), Text Segment (code, read/execute, stores binary state of the process), Data Segment (read/write, global or static local variables that have been initialized by the programmer resides here), BSS Segment (read/write, global or static local variables that have not been initialized resides here), Heap (free floating coarsely managed memory section for the programmers to store global variables which supports dynamic memory allocation, by default global variables are stored here, this space grows up, handled at execution), Memory Mapping Segment (file mapping, shared libraries and other anonymous mapping are stored here, grows downwards), Stack (temporary memory space that stores temporary variables usually generated by functions, handled during runtime), Command line arguments (arguments passed into the process), Environment variables (such as %TEMP%, %APPDATA% etc. are stored here) and lastly the Kernel Space (user cannot access this space, it will cause segmentation fault). The structure may be more detailed than what I have mentioned but I believe this should be enough for this purpose. This is the basic structure of VAS.

I believe that idealized picture of VAS is not completely accurate in the age of 64-bit systems because the address space can be extremely large, i.e., 16 exabytes. A small portion of it is only used. I think locality of reference also comes into play in this matter. Having an unnecessarily large VAS can slow things down. Therefore, there could be chunks of memory aligned systematically by an entity, may it be the programmer or the CPU itself, to maintain optimum locality.

5. A thread is basically a thread or a path of execution of a process. Threading is a way for a program to split itself into multiple parallel threads of execution. This can greatly increase performance of a process if used correctly. Threading is used to enable parallel processing of data and logic in an application. It allows demanding applications to run smoothly. For example, there are two different lists that need to be sorted. If a single threaded process tries to do that, it will take twice the time it takes to sort one list. But if it is a multithreaded process, it will take about the same time to sort both lists as it takes to sort one list. In other words, it is a way to divide and conquer workload of a process by optimizing time between independent tasks that can be run in parallel.

Critical section can be defined as a section of concurrent code that is accessing shared data. This can cause data corruption and unexpected behavior. For example, there is a variable x, and there are two different methods one of which changes the x value to 1 and the other changes it to 0. If the process is split into 2 threads for running both methods concurrently, it can lead to unexpected results. Sometimes the process will return 1 and sometimes 0. This is not at all a good thing. To solve this, there are several ways, but they must follow the conditions mentioned below:

1. Mutual Exclusion: No other thread is allowed to execute in the critical section until the thread that got there first finishes.
2. Progress: When no thread is executing in the critical section and there is another thread that wants to execute in that section, they should be able to do so and not wait indefinitely.

3. Bounded Waiting: There must a limit to the number of times a thread is allowed to execute in a critical section after a different thread has requested to execute in the same section.

Now, the solutions. There are several ways to fix this problem, some of which are,

1. Mutex Locks: A software library that consists of methods that can lock and unlock a section for atomic execution. Atomic operations basically look like indivisible entities to the other threads and is therefore impossible for them to execute it in that instant. They can only do so before or after an operation has gone atomic.
2. Condition Variables: A queue of threads waiting to enter the critical section in a respective manner based on a condition for example, if the other thread has finished executing.
3. Barriers: It is a synchronization method that can allow a program to makes all its thread wait at a point until every other thread reaches the same point.

```
1.  //
2.  // Created by Anirudh Lath on 4/29/22.
3.  //
4.
5.  #include <iostream>
6.  #include <mutex>
7.  #include <vector>
8.  #include <thread>
9.  #include <condition_variable>
10.
11. using namespace std;
12.
13. class Barrier {
14. private:
15.     int count_; // Total number of threads
16.     mutex m_; // Mutex to lock
17.     condition_variable cv_; // Condition variable to make the threads wait for the
    others
18.
19. public:
20.     Barrier(int count) { // Constructor
21.         count_ = count;
22.     }
23.
24.     void wait() {
```

```
25.         unique_lock<mutex> lock(m_); // Lock because changing a member variable that
    may be accessed at the same time
26.         // by the other threads
27.         if(--count_ == 0) { // If all threads have finished running, notify all the
    other threads.
28.             cv_.notify_all();
29.         }
30.         else {
31.             cv_.wait(lock, [this] {return count_ == 0;}); // If all the threads have
    finished
32.             // running,
33.             // return true
34.             // and unlock all threads to let them continue onto the next task. This
    will also prevent spurious wakes.
35.         }
36.     }
37. };
38.
39. int main() {
40.     Barrier bar(10); // Create barrier that will stop 10 threads
41.     vector<thread> threads; // List of threads
42.     threads.reserve(10); // Reserve space for all threads
43.     mutex m;
44.     mutex n;
45.     for (int i = 0; i < 10; ++i) {
46.         threads.emplace_back([i, &m, &bar, &n](){
47.             for (int j = 0; j < 5; j++) {
48.                 lock_guard<mutex> lock(m); // lock for cout
49.                 cout << "hello thread #" << i << " times " << j << endl;
50.             }
51.             bar.wait(); // wait for all threads to finish
52.             for (int j = 0; j < 5; j++) {
53.                 lock_guard<mutex> lock(n);
54.                 cout << "goodbye thread #" << i << " times " << j << endl;
55.             }
56.         });
57.     }
58.
59.     for (int i = 0; i < 10; ++i) {
60.         threads[i].join(); // join threads
61.     }
62.
63.     cout << "FINISHED!" << endl;
64.     return 0;
65. }
```