My notes to take while studying:

Parallelism involves tasks happening at the same time. Because of this it requires multiple processing units adica cores! Whereas Concurrency involves multiple tasks making progress in overlapping time periods. If I were the apply the same image filter to different blocks of a large image, simultaneously this is data parallelism because it is manipulating data parallel. Fine grained parallelism means there is frequent communication and synchronization between tasks. They are small not as big adica mai aproape de unul pe altul. When we talk about speed up in parallel performance it is a ration comparing the sequential execution (adica fara threads just one) to the parallel execution. Sequential/parallel. Amahl’s law says the you can speed up programs quite a bit using threads however, any sequential sections of a program limits the maximum speedup of a program. There is something called “The Free Lunch is Over” which basically means clock speed are not increasing significantly, so that means softwareis required to be parallelized for performance gains on multi-core processors. The GIL in python simplifies memory management but prevents true parallelization in python for CPU-bound tasks with the threading module. The multiprocessing module in python however doesn’t have those issues wit the GIL and you would really want to use that module when doing CPU-bound tasks where true parallelism is required for improved performance. CPU-bound tasks are often complex mathematical calculations for things like phyisics. When the CPU has gott compute something that is cpu bound. When threads are created in the same process. They share the same memory like global variables it is quite an advantage insome cases. The critical section in context of concurrent programming is part of the code where a shared resource is actions among many threads which require exclusive access to prevent some race conditions. The threading.lock in python is a way to prevent those race conditions. It makes it so that only one thread at a time can execute a specific block of code (aka critical section) at a time which prevents race conditions. Python classes are an important thing to know in this class to understand its OOP. When you create a threaded class, it has a bhevaior called run and it is ran when you call the start on the thread object you created. It internally calls run() when start() is called. If you want to schedule the threading module adica to delay it before it runs, in python the threading.Timer(delay, target\_function) allows that. In the life cycle of a thread after you call start(), the thread is then in the runnable (or ready) state since the system scheduler hasn’t actually allocated CPU time for it to execute yet. So it will be in a runnable state until the cpu schedules and executes it.if two or more threads are blocked forever, each waiting for a lokc or resource that is held by another thread in the same waiting group, that is called a deadlock. I is bet to use threads for i/o bound tasks (network requests or disk access). The gil prevents true parallel execution on multiple core which limits speedup. When there are bugs in a multithreaded program that seme to disappear and reappear when you try to observe throught multiple runs or through print statements and debugger, we call this heisenburgs. Code is considered thread safe if it functions correctly firstly and then produces the predicitable results even when executed concurrently by multiple threads. The most common source of thread safety is shared mutable state. (which is data that can be changed and accessed by multiple threads). A context switch can be triggered when a blocking system call is made (like waiting for i/o). the information that is saved as a threads context is the first program counter (PC), cpu registers, and stack pointer. Is there is too much context switching going on we call it threashing. Which this degrades the performance as the system ends up spending more time switching than ctually executing code. A really good benefit for using a thread pool is that it reduces overhead of thread creation and destruction by reusing existing worker threads. Multiple threads can download tasks submitted to the pool to run concurrently by using the map function where you have the target function that is mapped to a given list and the threads in the pool takes those items from the list and apply the function. Overhead of thread management and context switching and with a combo of the gil can outweigh the benenfits of using a thread pool on cpu bound tasks. Would be better to either keep it sequential or use the multiprocessing module. The main danger of using shred memory directly for interthread communication without proper synchronization is that there can be lots of race conditions which can corrupt data and have unpredictable results. The fundamental principle of the queue.Queue data structure is the FIFO (first-in-first-out). It is advantageous to use a queue for interthread communication because it is inherently thread safe handling internal locking automatically. When a producer puts None onto the queue that is a sentinel value, which signals to the consumers that noe more valid items will be produced. (sometimes use have to pass in the number of None values for the number of threads or you could implement it in a way that where the thread puts it back onto the queue before breaking. The internal counter of a counting semaphore is metaphorically speaking the number of available permits or slots for accessing a shared resource. It’s a way to limit threads accessing shared resource to prevent bugs. It can also be used to have a maxsize of a queue adica how many threads can pull from a queue. A counting semaphore can be non-negative integer whereas a binary semaphore is just 1 or 0 (which is more like a mutex or lock rather than as something many threads can have at once in a limited number). Using the syntax with semaphore or even with lock is a shorthand way that automatically calls .acquire() and .release() before entering and upon exiting the block. Ensures a release even when error occurs and it helps with readability too. The threading.Barrier synchronization primitive makes a group of threads wait until all participating threads have reached a specific synchronization point before any are allowed to proceed. When the .wait() method is called the thread blocks until a specified number of parties (threads) required by the barrier have all called wait(). A process has its own private isolated memory space. Memory usage differs between threads and processes primarily from that processes have independent memory spaces while threads share the same memory space. The multiprocessing module is the primary interface for creating and managing processes. Os.getpid() returns the unique process ID (PID) of the currently executing process. Inter-Process Communication (IPC) is necessary because processes need a way to communicate amongst themselves. Lol they got their own memory therefore they cnt directly share variables and need a way to communicate. There is a limitation to the threading module but can be overcome by using multiple processes it the GIL – that is overcome through the mp module which allows true parallelism for cpu bound tasks. The challenge with multiprocessing however is that there is higher overhead and complexity compared to interthread communication via shared memory. The multiprocessing.Queue is the class that is designed to be process safe for communication between different processes. The mp.Pipe() returns a pair of connection objects representing the two ends of a pipe. The mp.Pipe() is typically best suited for direct communication between two closely related processes (e.g. parent/child) mp.Manager facilitates safe sharing of objects like lists and dictionaries between processes by using a separate server process to manage the actual objects and providing proxies to worker processes. The advantage of using multiprocessing.shared\_memory allows direct memory access without pickling this offers lower overhead especially for large numerical data. Shared\_int = mp.Value(“I”,10) this creates a shared integer and you would access it by sytax of .value attribute (shared\_int.value += 1) the map() in the multiprocessing.Pool manages to apply a given function to each item in an iterble, distributing the work among the pool’s worker processes and collecting results. The key difference in execution behavior between pool.map() and pool.apply\_async is that the mapis synchronous (blocks until all results are ready) while the apply\_async() is asynchronous (meaning it returns immediately with a result object) Iterating through the list of result objects (results) and calling .get(0 on each object ([p.get() for p in results] so the .get() basically is used on each object in the iterable. When you use the pool.apply\_async with a callback function the purpose of pool.close with a pool.join() ensures to signal that there are no more tasks will be submitted and then wait for all submitted tasks in the pool to complete their execution. The biggest reason you would choose processes over threads especially for those performance critical applications is to overcome the GIL limitation for CPU-bound tasks and achieve true parallelism across multiple cores. The drawback with processes however is the higher resource overhead. (memory, context switching,) and more complex inter-process communication. (IPC) to import the mp.Lock, mp.Semaphore, etc.. it is imported from the multiprocessing module or a multiprocessing.Manager object. The throughput in the context of OS scheduling goals is completing as many process/threads as possible per unit of time. The convoy effect is something that comes form the first come first serve scheduling. As a result short processes get stuck waiting behind long processes that arrived earlier. The major drawback of the shortestjob first scheduling algorithmis that it requires knowing the length of the next cpu burst, which is usually difficult to predict accurately. The round robin scheduling algorithm assigns each process a small, fixed unit of cpu time known as a time slice or quantum. The potential problem you get from priority scheduling queues is that there can be starvation of lowpriority processes if there’s a continual supply of high-priority processes. The aging technique used in priority scheduling is to prevent starvation by gradually increasing the priority of processes that have been waiting for a long time. The multilevel feedback queue scheduling from basic multilevel queue scheduling is that the process can migrate between different queues based on their execution behavior (CPU usage, waiting time) context switching in an operating system is the mechanism of saving the state (context) of the currently running process/thread and restoring the state of another to allow the CPU to switch between them. What you save in the context is the program counter, cpu registers, stack pointer, and process state. The key difference between premptive and nonpreemptive (cooperative) multitasking is that the the prememptive multitasking the OS can forcibly interrupt a running process to switch to another; in non-preemptive processes must voluntarily yield control. Virtual memory abstraction allows for process isolation, preventing processes from interfering with each others memory spaces. While threads within a single process share resources like the heap and code segment. Each thread possesses its own private stack use for local variables function parameters and return addresses. The main purpose of cache coherence in modern multicore processors is ot ensure that ll cores maintain a consistent and correct view of shared memory data despite having their own private caches. False shring is when different therads modify logically distinct vriables that happen to reside on the same physical cche line, causing unnecessary cache coherence traffic. Hyperthreading (aka simultaneous multithreading -SMT) enhances cpu performance by a single physical core duplicates some internal execution resources like registers to appear as multiple logical cores to the OS allowing faster switching between threads to hide latency. Non-uniform Memory Access computer architecture defining characteristic is the time required for a processor to access a particular piece of memory depends on the physical location of that memory relative to the procesor’s node (local vs remote access times differ). The architectural feature that makes graphics processing units (GPUs) particularly effective for certain parallel computing tasks is a massively parallel design with thousands of smaller cores optimized for SIMD (Single Instruction, Multiple Data) operations. The physical movement time of the read/write head (seek time) required to access different data location making concurrent random access ineeficient is a primary limit to the parallel I/O performance of traditional spinning hard disk drives. Advisory file locking and madantory file locking difference is the the advisory locking relies on processes copperating to explicitly check for and respect lock, while mandatory locking is enforeced by the operating system kernel itself. A key beneit of file system journaling provides for modern operating systems is improved data integrity and faster file system recovery after unexpected crashes or power failures by logging metadata changes before applying them. The Boss Worker pattern: the boss’s job is to distribute tasks to worker threads and potentially managing them. The main role of the worker threads in this pattern is to receive tasks from the boss and execute them independently. It could be that the boss puts tasks onto a shared queue for the workers to execute. Producer thread in a producer consumer pattern generate data or tasks and place them into a shared buffer or queue. The role of the consumer threads in a the producer consumer pattern is to retrieve data or tasks from a shared buffer of queue and process them. The primary benefit and purpose of the producer consumer pattern is the decoupling the rate of data generation (production) from the rate of data processing (consumption) Something that distinguished the bounded buffer from the basic producer consumer pattern is that the shared buffer has a fixed maximum size. A producer will block or wait in the bounded buffer pattenr when the shared buffer is full. The consumer block will wait in the bounded buffer pattern when the shared buffer is empty. The typica lrole of the server in the client server pattern is to listen for incoing client connections process client requests and senf back responses. The clients typically initiate connections ot the server send requests for services or data and receive responses. The main synchronization challenge in the reader writer problem is allowing multiple reader threads to access a shared resource concurrently while ensuring that writer threads have exclusive access. In reader priority solution ot the reader writer problem wirters might potentially face starvation if there is a continuous stream of incoming readers. The dining philosophers problem, each philosopher must acquire two chopsticks simultaneously (specifically the ones to their left and right) the classic ocncurrent issue in the dining philosophers problem is a dealock where each philosopher acquires on resource (their left chopstick) and waits indefinitely for the other resource held by their neighbor if not implemented correctly. The sleeping barber problem, the barber goes to sleep when there are no customers waiting in the shop chairs and none currently being served, until someone arrives and wakes him. When a potential customer arrives at the barber shop when all waiting room chairs are already occupied, the customer gives up and leaves the shop immediately without getting a haricut.in the cigarette smokers problem the agent entity provided in each cycle two out of three distinct ingredients (tobacco and paper, but lacking matches) required to make a cigarette. The core concurrency challenge that is highlighted by the synchronization requirements in the cigarette smokers problem is coordinating complex conditions to ensure the correct smoker (the one with the third ingredient) is signaled and avoids deadlock or starvation. The fundamental technique emploted in the monte carlo simulations is using repeated random sampling of inputs or processes to botain numerical results or staitistical estimates. The biggest difference regarding the GIL when comparing Cpython threading ot C#/.NET threading is that C# does not have a GIL enabling threads to execute truly in parallel on multiple CPU cores for CPU-bound work. When compared to python’s threading and multiprocessing modules, what library is considered the cornerstone of modern c# concurrency introducing abstractions like Task and Taks<Result>? 🡪 that is the Task Parallel Library (TPL) which is found primarily in system.threading.tasks. The Thread class in the system.Threading namespce is used to represent and directly create a managed operating system thread in C#. the parameterize threading works by passing an object argument for example workerThread.Start(messageToSend) method call. The Join method must be clled on a thread object to make the current thread pause and wait until that specific thread object finishes its execution. The primary difference in application lifetime behavior etween foreground and backgrounf threads in .NET is that a .NET application will exit even if backgrounf threads arestill running but it will wait for all foreground threads to complete. The primary purpose of using the lock keyword (which uses system.threading.monitor implicitly) in C# is to acquire a mutial-exclusion lock on a object, ensuing only one thred can execute the code block (critical section) at any given time. Beyond mutual exclusion provided by the lock keyword there are other additional dvanced synchronization capabilities that the system.threading.monitor class offers which is condition variable support via methods like wait pulse and pulseall for complex coordination scenarios. The system.threading.mutex ffers a system-wide synchronization between threads in different processes if given a unique name and this is something that is generally not available with the lock keyword or monitor class in c#. the main use for system.threading.semaphoreslim is to limit the number of threds that can concurrently access a specific resource or execute a particular section of code. Event Wait Handles (autoresetevent, manualresetevent,manualresetslim) are primitive like manualreseteventslim is primarily used for signaling between threads where one or morethreads wait (Wait()) until another thread signals (Set()). System.threading.readerwriterlockslim is particularly advantageous compared to a simple lock when a shared resource is read very frequently by multiple threads but modified (written to) only occasionally. Simple low level atomic operations (increment, decrement, add, compare-and-swap\_ on shared primitive types often avoiding the need for heavier locking mechanisms are the type of operation that are provided by the static methods in the system.threading.interlocked class. The central abstraction provided by the task parallel libry tpl in c# that represents an asynchronous operation is the Task (for operations without a return value) or Task<TRESULT> for operations with a return value. The most common way to execute a cpu bound operation asynchronously using tpl typically utilizing the threadpool is to Task.Run(()=>{/\*CPU-bound work\*/}); the sttic method of the task class that allows thread to block until all tasks within a provided collection have finished their execution is the task.waitall(). The primary goal of using async and await keyword when performing potentially long-running operations (especially i/o bound ones) is to enable asynchronous operations without blocking the calling thread making application more responsive and scalable. When the await operator is applied to a task that has not yet been completed, its crucial behvaior is that it typically registers a continuation for the rest of the method and returns control immediately to the caller of the async method. Collections in the system.collections.concurrent namespace are generally more preferred over standard collections like List<T> or Dictionary<TKey, TValue> when accessed by multiple threads is because they are designed to be thread-safe incorporating internal synchronization mechanisms to prevent data corruption during concurrent access and modifications. The concurrent collection class that is specifically highlighted as being exceptionally useful for implementing producer-consumer scenarios offering built in blocking and bounding capabilities is the BlockingCollection<T>.