**Week 6: Parallel and Concurrent Programming**

**Overview**

In our previous class, we looked at how two programs can communicate over a network.  If nothing special is done, a process can only handle one connection at a time.  This is because reading from connections cause the process to block if there are no data available. We need to look towards concurrent and parallel programming techniques to handle multiple connections. Some useful definitions:

Concurrency: A condition that exists when during a given *period* of time, two threads are making progress.

Parallelism: A condition that arises when, given a particular *point* in time, two threads are executing simultaneously.

It's useful to think of concurrency as a superset of parallelism; anything running in parallel is concurrent, but not everything that is concurrent is running in parallel.

There are four main ways to implement a network server so that it can handle multiple connections at a time:

**Non-blocking I/O.**  In this case, the operating system's select() (or similar) functions are used to identify which connections have data available to read or room to write data.  The program will loop call select() and loop through the sockets that are ready.  The main challenge with this approach is that the server needs to keep track of where it is for in terms of the handling of each connection.  For example, if the server only receives part of a SQL query string, it will need to save what it received to an internal buffer.  Once the server has a complete query, it can execute the query and return the result.  The need to track state leads adds significant programming complexity.  The main advantage is that the program is still effectively single-threaded so the programmer doesn't need to think about synchronizing access to shared resources. The main disadvantages are that multiple connections are not served in parallel and that if any blocking I/O is performed, the entire server is blocked.

**User-level threads.**User-level threading libraries attempt to simplify the programming of concurrent but not parallel code. The threads don't ever run in parallel.  Rather, they employ cooperative multitasking in which individual threads decide when to yield control.  One thread can never interrupt another thread.  User-level threads are seeing a resurgence in the form of programming language constructs for "[structured concurrency.](https://en.wikipedia.org/wiki/Structured_concurrency)" such as [asyncio in Python](https://docs.python.org/3/library/asyncio.html" \t "_blank).  Since threads save their current state of execution when they switch, the programmer does not need to do as much explicit work to manage the state of individual connections.  The main advantage of user-level threads compared with kernel-level threads is that switching between user-level threads is much lower latency since it can be done completely in user-space.

**Multiple processes.** The traditional Unix way of handling multiple connections was to fork a process to handle each new connection.  From a programming perspective, this approach reduces complexity in the response handling since blocking I/O can be used and the OS will automatically pause and restart the process as needed.  The other advantage of this approach is that Unix operating systems tend to define security and resources at the process level.  The system administrator can limit the amount of network bandwidth, share of CPU, or fraction of memory or disk for each individual process and thus for each connection.  In practice, however, there is often an impedance mismatch between the database's and OS's security models, so rate limiting and memory limits are handled by the database itself.

There are several challenges with the process model.  In particular, the multiple processes need to access the same underlying data.  Postgres and other systems used shared memory segments or inter-process communication (IPC) to accomplish this.  In the case of a shared memory segment, if one process is reading data while another is writing data, the reading process may read partially updated data which is not self-consistent.  The programmer needs to use synchronization primitives to control access.  In the IPC approach, you might design your server so that there are helper processes that handle reading and writing data.  This requires the respond handler process to communicate with a second process.  By using messages, there is no way for two processes to directly access the same resource (messages contain copies of the data) but the performance of this approach tends to be low.  Lastly, there is a lot of work for an operating system to swap processes that can limit the number of connections that can be handled at one time.

**OS-level threads.**OS-level threading has become a popular alternative to using multiple processes.  Threads share the memory of their containing process, making it easier to communicate between threads.

In summary, the choice of concurrency versus parallelism is going to depend on the nature of the computation.  Non-blocking I/O and user-level threads are great for bursty behavior.  They allow the program to make progress on other tasks while waiting for data to be read from disk or for a client to send its next request. They tend to be favored in some modern web application frameworks. Processes and OS-level threads are well suited for heavy computational work in which progress needs to be made on multiple tasks that are not waiting on anything.  Processes and OS-level threads are used in scientific computing applications, while user-level threads are not.

When dealing with concurrency and parallelism, synchronization becomes an issue. The readers-writers problem is particularly relevant to databases.  We could allow a single thread to read or write the database at a time.  This does not allow any parallelism, however.  Readers don't change the databases, so we should be able to allow multiple readers at the same time.  We need to ensure that writers, however, get exclusive access to the database.

There are several solutions to the readers and writers problem.  One solution prefers readers.  A writer is not given exclusive access until all of the readers have finished.  That said, until the writer has exclusive access, new readers may appear and are given access.  The writer could effectively wait indefinitely.  This is the readers-preference solution.

The second solution is called the writers-preference solution.  Once a writer appears, all existing readers are allowed continue but new readers are blocked.  Once the writer is complete, the additional readers are allowed to continue.

A third solution, based on queues, offers a balance between the two previous solutions in which requests are served in the order they arrive.

**Reflection Questions**

* Give examples that illustrate the differences between concurrency and parallelism.
* What are the relative strengths and weaknesses of processes, kernel threads, and user threads?  Which approaches allow concurrency?  Which approaches allow parallelism?
* Can you explain the readers-writers problem? Why do writers need exclusive access? What can go wrong if two writers make changes at the same time?
* What are mutexes? What are semaphores?
* What are deadlocks?
* Trace the algorithm for the reader-prioritized solution to the readers-writers problem and explain what happens at each step.