Introduction

File uploads and downloads are two of the most important tasks done every few minutes. Although large files are dealt with very easily nowadays, the issue of handling concurrent users is still a problem, as many people upload files at the same time. The problem is relevant to computer architecture because the solution of this problem is dependent on how the computer is designed on a hardware level, such as the use of single-threading or multi-threading, the use of memory or cache, the use of blocking or non-blocking disk access inside, and so on. My focus is to improve the issue of parallel I/O architecture when dealing with the problem of file operations carried out by many people at the same time from the architectural side of things.

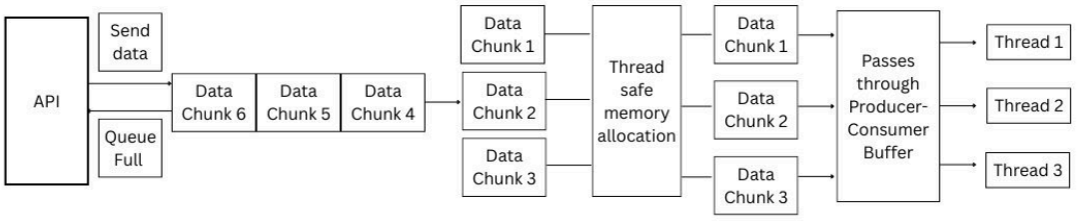
Background/Related Work

First, the difference between blocking and non-blocking disk access must be understood. In case of disk access being blocked, the OS dictates whether it has to wait before operating. However, in the non-blocking case, there is no waiting time, but in this case, it is more difficult to implement (GeeksforGeeks, 2023). The Thread Per Request model is one of the popular models, and it falls under the blocking disk access type because here, each incoming request is managed by a single thread, and there is a queue as well (Taank, 2024). If the number of threads comes up short, then the thread pool is configured to allow more work to be done by more threads, and this is used in the case of a Servlet container (Tomcat). The Worker Pool model, although similar to the previous one, has a glaring difference, which is the fixed number of workers with respect to the continuous generation of threads in thread-per-request mode. Here, there is a job queue as well to keep track. But each time a certain number of workers are assigned, they complete the tasks in rotation (Zepeda, 2025). This goes on and on till there is some kind of terminating case. The main takeaway from this model is that it saves a tremendous amount of memory as the number of workers is limited every time, and also reusable. The use of this model can be found in the case of the Go language. Lastly, the asynchronous event loop architecture is a non-blocking system. There are three important stages here (Node.js — the Node.js Event Loop, n.d.). Firstly, the timers fix the time of the callbacks, like how long each function runs for and in the queue, when it will finish or start. Moving on to the poll queue, we observe that the waiting time for I/O is determined, and new incoming tasks are queued in a systematic way. After all the tasks have been done, the poll queue will be terminated, and the next stage is to check where the callbacks that were defined in the initial phase will be executed accordingly.

There are many potential bottlenecks that arise while dealing with different kinds of scenarios. The thread starvation problem is one of them. This happens when the high-priority threads keep executing, and as a result the low-priority threads never perform (GeeksforGeeks, 2025). This results in high usage of memory and slows the system down in general. The next common problem is the shared queue problem. When concurrent programs are run, they try to access a shared memory to complete their respective tasks (*Shared Queue | Our Pattern Language*, n.d.). But a race condition must be maintained, or else threads can overwrite each other, and there are many ways to deal with this problem. For the socket backlog, there must be an agreement between the client and server. Veithen (2015) shows how the “Listen” phase takes place, which ensures that receiving connections will be dealt with, and which ones will remain in the queue. A handshake system is employed in this case. From the user side, it requests the server for service, and the server side also confirms it, and if it does not confirm, those requests remain in the waiting line. BillWagner (n.d.) explains different kinds of file system locks. Hence, the permissions are modified when it is done, which means not everyone will share the same privileges. For example, some files are kept out of bounds for the user and handled by the administrator in the operating system. NVMe is used to deal with the I/O operations (*Queue Associations | an In-Depth Overview of NVMe and NVMe-oF | Dell Technologies Info Hub*, n.d.). The main limitation in this case is the overwhelming number of I/O operations. Because of this, fast communication is possible throughout the whole architecture.

Methodology/System Design

The whole file is divided into chunks and then uploaded in a queue format, which follows FIFO (First In, First Out) (Mohit, 2025). Freitas (2024) explains how back-pressure helps to maintain the overloading of data, when the next batch of data may cause overloading, a feedback loop signals that



the queue is full. Thread-safe memory allocation is used to manage threads optimally. For each thread, there is a bin, and in that bin, there is data of the next bin location, thread ID, and so on. Now, without modifying the bin data, if any thread is freed, the remote_free_count is increased, which means the number of freed bins. Remote_freed indicates the ones that have been freed. For further security, a mutex lock is present so that the shared memory is not accessed by two threads at the same time (Bauroth, 2024). The Producer-Consumer architecture ensures that in shared memory, either processing data or producing data is done in the consumer and producer, respectively, by a queue. A pool queue is also there that generates any "x "number of threads to handle the workload, and after everything is done, they destroy themselves (*Lecture 17: Concurrency—Producer/Consumer Pattern and Thread Pools*, n.d.). The core I/O thread system is used here. That means, each thread operates independently using its own resources but also communicates with others when needed (*The Impact of Thread-Per-Core Architecture on Application Tail Latency*, 2019b). Large files more than 100MB can be sent through chunks, and back pressure will handle 20 users effectively. Thread per core I/O reduces tail-latency, read tail-latency and update tail latency by 71%, 16% and 47% with respect to Memcache (Bauroth, 2024). SJMalloc which uses thread safe allocation works 5.76% more than the GliBc allocator. As the following architecture is highly scalable, the throughput requirement, that is, >400MB/s and <200MB/file, can be met.

The per-core I/O thread ensures the resources are used efficiently with very low overheads, and there is no need for synchronization, which eliminates many threats like problems related to context switches. Sending data in parts means it will be scalable, fast, and memory efficient. Back pressure can reduce data overloading and be more risk-free to use. The threads are handled meticulously.

Discussion

The concurrent use of thread-safe memory allocation, a producer-consumer buffer, and a per-core I/O system is the novel idea here, along with sending data in parts with back pressure. The impact is estimated to be influential if this system can be implemented because of the memory efficiency in each stage. Although there are some limitations in this case. The per-core I/O thread may prioritize smaller tasks over larger tasks at times and cannot deal with situations when synchronization is needed. The Producer-Consumer model will fail when a large amount of data is sent, and will cause strain on the network as well. The queue will take more time to handle data at that time (*Limitations of the Producer Consumer Template*, 2018). Sometimes data cannot be sent in chunks due to network limitations, and overheads may cause problems (*What Is HTTP Chunked Encoding*, n.d.). Thread-safe memory allocation requires a complex mechanism in the coding phase. Memory will be wasted as well at times for continuous allocation and deallocation (Bauroth, 2024). In the case of a cloud environment, network packets may cause problems as data is sent in chunks. The cost is expected to be less because the whole system is scalable. But failures may increase the cost. The I/O storage is expected to be fast and the whole thread scheduling process is easy to implement as well, which reduces redundancy.

The impact and sustainability of modern, fast-changing computer architecture and technology are increasing day by day. With the improvement of GPU and fast CPU processing power, the potential is immense. On the other hand, with the ascension of cloud technology, shared computing is also a thing.

Conclusion

The architecture is a layered architecture with a lot of sensitive steps to execute. The main purpose of this system is to be resource-friendly and be able to handle big files. The only concern is to deal with network pressure when sending files and how to use back pressure wisely.

References

1. GeeksforGeeks. (2023, January 7). *Blocking and nonblocking Io in operating system*. https://www.geeksforgeeks.org/operating-systems/blocking-and-nonblocking-io-in-operating-system/
2. Taank, V. (2024, September 11). *The thread-per-request model*. Medium. https://medium.com/@vikas.taank_40391/the-thread-per-request-model-dffcdefbfff8
3. Zepeda, E. (2025, August 20). *Worker pool design pattern explanation*. Coffee bytes. https://coffeebytes.dev/en/software-architecture/worker-pool-design-pattern-explanation/
4. *Node.js — The Node.js event loop*. (n.d.-b). https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick
5. GeeksforGeeks. (2025, July 11). *Deadlock, starvation, and Livelock*. GeeksforGeeks. https://www.geeksforgeeks.org/operating-systems/deadlock-starvation-and-livelock/
6. *Shared Queue | Our pattern language*. (n.d.). https://patterns.eecs.berkeley.edu/?page_id=587
7. Veithen, A. (2015, March 14). *How TCP backlog works in Linux*. Andreas Veithen's Blog.
8. https://veithen.io/2014/01/01/how-tcp-backlog-works-in-linux.html
9. BillWagner. (n.d.). *FileSystem.Lock Method (Microsoft.VisualBasic)*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualbasic.filesystem.lock?view=net-9.0
10. *Queue associations | An In-Depth Overview of NVMe and NVMe-oF | Dell Technologies Info Hub*. (n.d.). https://infohub.delltechnologies.com/nl-nl/l/an-in-depth-overview-of-nvme-and-nvme-of/queue-associations/
11. (Mohit, 2025). Medium.com. https://medium.com/@mohitrohilla2696/chunk-upload-in-react-native-using-file-read-and-write-from-storage-8554607e2bf9
12. Freitas, W. (2024, December 5). *Applying back pressure when overloaded: Managing system stability*. DEV Community. https://dev.to/wallacefreitas/applying-back-pressure-when-overloaded-managing-system-stability-pgc
13. *Lecture 17: Concurrency—Producer/Consumer Pattern and Thread Pools*. (n.d.). https://www.cs.cornell.edu/courses/cs3110/2010fa/lectures/lec18.html
14. *Limitations of the producer consumer template*. (2018, March 12). https://forums.ni.com/t5/LabVIEW/Limitations-of-the-Producer-Consumer-Template/td-p/3764915

15. Bauroth, S. (2024, October 23). *SJMalloc: the security-conscious, fast, thread-safe and memory-efficient heap allocator*. arXiv.org. https://arxiv.org/abs/2410.17928
16. *What is HTTP Chunked Encoding*. (n.d.). https://www.ioriver.io/terms/http-chunked-encoding
17. The impact of Thread-Per-Core architecture on application tail latency. (2019, September 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8901874