Models for Organization of Threads

Dhara Tamhane

Femina Shah

Krima Shah

Malavika Iyer

Preface

There is substantial difference between theoretical knowledge of Operating System and implementing it in real-time scenario. Mending differences between them is a quest.

Knowledge gained from a book can provide immense amount of conceptual knowledge while implementing it in real-time scenario can be challenging. A through and practical study can be useful to develop instincts to use the knowledge in a given setup.

Keeping this point of view in mind every student of Operating System class of Santa Clara University undertakes his / her choice of project related to the subject. This initiation enables them to understand this pivotal subject plus preparing them to be competent enough to face all future challenges that will be encountered.

Thus as per syllabus Santa Clara University has specially included this project.

Acknowledgement

First, we would like to thank Santa Clara University, which has given us the opportunity to work on this project. We would also like to thank our institute, School of Engineering.

Secondly, we would like to express our gratitude and thank the Dean of our Institute, Mr. Godfrey Mungal.

We are very thankful to the entire staff of the library for their guidance and allowing us to utilize the resources of the library.

Last but not the least, we are grateful to those who have directly or indirectly helped us to make this project a remarkable experience.

Table of Contents

List of Figures

Abstract

Any application aims to complete the required task in the least amount of time. But if an application has too many tasks at hand, it slows down the entire process since just one entity has to execute all the tasks. To overcome this issue, we make use of multithreading. Using multithreading, multiple tasks or threads can be executed almost in parallel. When multiple threads are being handled, there arises the issue of how to handle the different threads and thus the models for organization of threads was developed. The models - dispatcher-worker, team and pipeline - outline methods as to how threads can be handled when are queued for execution.

Summary

Threads are one of the most integral parts of an operating system. The working of the entire Operating System tends to rest on the execution of threads. Threads are integral in enabling us to achieve parallelism for a single job.

The need for methods to organize threads arises from the need for threads itself. If a process could not have multiple threads, then the process can only handle one request at a time. This would result in the slowing down of the entire application or many other issues like one process being executed ahead of other processes.

Hence, there is a need for multithreading. With the implementation of multithreading, there are a few questions that arise. These are associated with the order in which the threads will be handled. As many threads simultaneously and continuously need to processed, there comes a need to organize the process into models. The project implements three different models for the organization of threads according to the demands of the application. The three models are -

- Dispatcher - Worker model
- Team model
- Pipeline model

We examine these models to understand the properties of each model and from there to infer which model can be applied to which particular situation.

We look at the implementation of the models, the advantages and disadvantages of each and the variations in implementation associated with each model.

1. Introduction

An application such as the mail server handles multiple requests at the same time. If the application could only handle one request at a time, there would be a large number of requests waiting for extended periods of time. This would happen in the absence of threads. On the other hand, when there are multiple threads to handle the requests, multiple requests can be handled at the same time or concurrently which would lead to a huge improvement in performance. This is the ideology behind multithreading.

Multithreading has several benefits. Performance and concurrency for a system implementing multithread is enhanced. It allows access to multiple applications simultaneously. Multithreading in an application increases the responsiveness of the entire application since many threads handle the incoming requests. It also minimizes the impact on system resources since there is lesser overhead involved with threads when compared to traditional processes.

Multithreading improves performance and has now become a common practice. And this leads to the need for models to organize threads. These models help us to organize threads as a way to optimize the usage of threads.

2. Dispatcher-Worker Model

A single thread acts like a dispatcher and creates worker threads and delegates work to all the worker threads. This model is also called Delegation Model. It is called the Boss/Worker or Master/Slave model since one thread acts as the Boss or Master and allocates work to the other threads which are Worker or Slave threads.

The dispatcher thread and all the worker threads execute concurrently. Each worker is assigned a task and it is the worker's responsibility to complete the task and produce output. The dispatcher thread executes an event loop to keep a check on all the requests coming in and assigns tasks to worker threads accordingly. The dispatcher can create a new worker thread every time a new request comes in, but using this approach may cause the process to exceed its thread limit.

The primary purpose of the dispatcher thread is to continuously check if there are any requests, create worker threads, place the tasks in the queue, awaken the worker threads when task is available. The purpose of the worker threads is to keep checking the queue for work, execute the assigned task and suspend itself if the queue is empty.

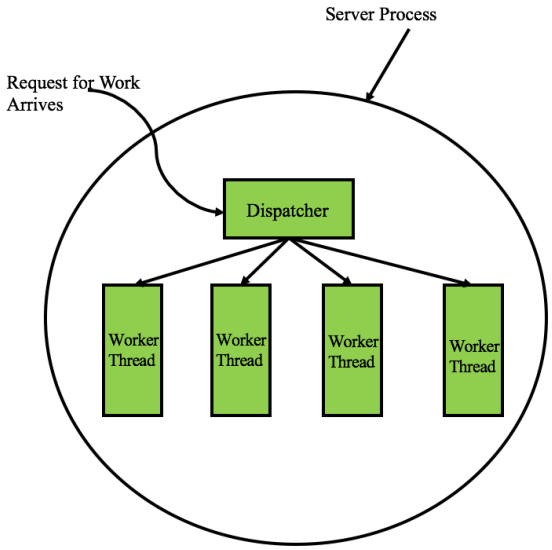

Figure 1: Dispatcher-Worker Model

2.1 Advantages-

- There is maximum utilization of worker threads since the threads can perform any type of job, and therefore as and when work arrives it is completed by any available worker thread.
- Threads can be recycled and reused which reduces the overhead or continually creating new threads.
- Failure of one thread does not affect the performance of other threads and the entire system. If one thread fails, the work of the thread is done by any of the other available threads since all threads can handle all kinds of jobs.

2.2 Disadvantages-

- There is contention for resources if there are interdependencies between threads. If one of the worker threads is dependent on another worker thread to perform its job, then the performance of that thread depends on the other thread and also how resources are shared.
- May consume extra CPU cycles to perform infinite loop while waiting for new requests. If a large of threads are continually performing an infinite loop while waiting, it might take a significant toll on system performance.

2.3 Implementation Variations-

A different approach to implement the Dispatcher-Worker model is by creating a thread pool where the dispatcher thread can create a fixed number of threads during initialization and each new task request is given in a queue. Every time a worker thread completes its tasks it checks in the queue to see if there is any other request waiting to be executed. If no requests are available, the worker

thread suspends itself until the dispatcher signals the worker that more tasks are available in the queue.

It can also be implemented by having an event pool to continually check for requests. When a new request arrives, a new worker thread is created to execute the request.

2.4 Applications-

The Dispatcher-Worker model is most suitable for application in servers like web servers and mail servers. In the web server application, the dispatcher performs an infinite loop for accepting work requests and assigning it to worker threads. The worker threads also continually check for new work requests being assigned to it and when a request is assigned, the worker thread checks the Web cache to see if the page is present and retrieves it. Also, servers are more efficient when implemented using this model. This is because in this model, many requests can be processed concurrently and a single thread doesn't get overloaded with requests. If one of the threads fail, the other threads can take over the functions of the failed thread since no thread does a specialized function.

2.5 Psuedocode for the Dispatcher-Worker Model-

```
DISPATCHER-WORKER-MODEL(inputFile){

Initialize display Frame

DispatcherThread {

while(! inputFile) {
Create the worker thread pool
take the task from the input file
put it on the queue
execute the worker thread
}

WorkerThread{

if (! Queue){
Take the task from queue
perform the task
display the result
}
else {
go to sleep till the queue is non empty again
}
```

2.6 Screenshot

The dispatcher has a sent of jobs that it keeps assigning to the worker threads. The worker threads execute the incoming jobs and sleep when they are idle.

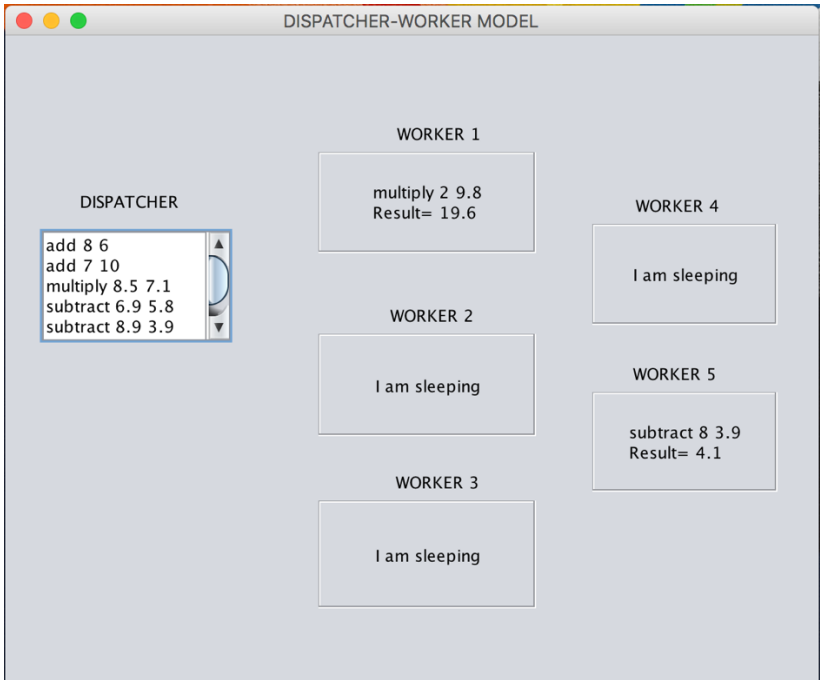

Figure 2: Dispatcher-Worker Model Screenshot

3. Team Model

In this model, each thread behaves equally while processing its respective input data and all of them work concurrently on their tasks. There is no centralized control. All the threads in a team model can share a single input or each thread can have its own input. Each thread knows the type of input it has to work on ahead of time and each thread processes it on its own. All these threads need to work in a team and it's possible that they need to communicate and share allocated resources too. This model is used for implementing some specific type of tasks done within a process i.e. each thread is specialized in doing a specific type of task. Hence different types of requests can be handled together in this model. Such a type of model is useful when we have fixed and defined inputs.

This model is also called the Worker-Crew model since all the threads work together to complete the same task working concurrently like a work-crew.

It is also called the Peer-to-Peer model since different parts of a process are divided among different threads and all these threads have equal status.

The diagram below explains how the team model works where one process can have three different types of inputs for three different activities and each of this request is handled by a different thread.

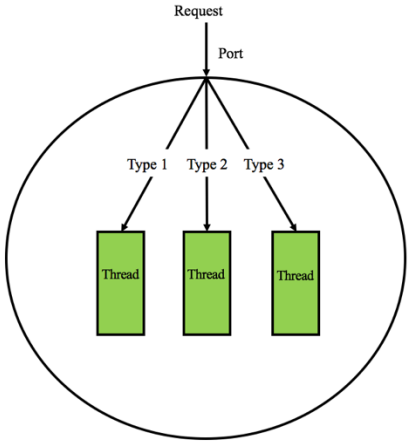

Figure 3: Team Model

3.1 Advantages-

- Each thread has a specialized function and therefore works independently while processing its respective input data.
- Failure of one thread does not affect the performance of other threads since each thread is independent.
- The model is easily scalable; more threads can be added when functionality increases.
- Since no thread knows about the working of another thread, there is no interference between threads.

3.2 Disadvantages-

- One particular thread can get overloaded due to high number of jobs while other threads remain idle.

3.3 Implementation Variations-

In the traditional implementation, each thread works independently but the input for all the threads in obtained from a single input stream. A variation to this is to have a dedicated input stream for each thread to get the input only for that particular thread.

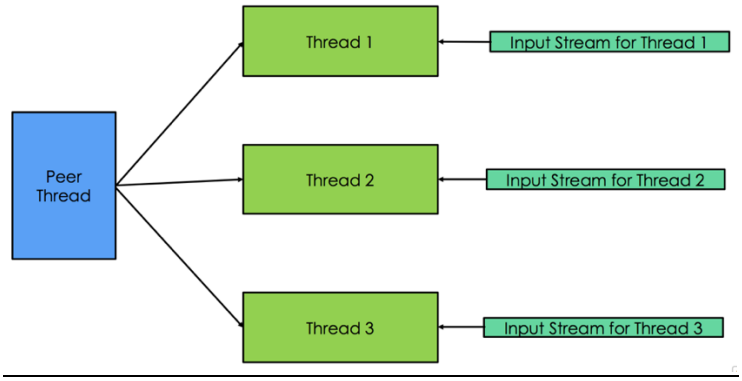

Figure 4: Implementation Variation of Team Model

3.4 Applications-

This model is suitable to applications like word processor. Since each thread performs a specialized task, each function can be assigned to a particular thread. For example, one thread can wait for input, another thread can do reformatting of data while the third thread periodically saves the state of the document. This is most effectively done using the Team model since each function is carried out independently and concurrently by different threads.

3.5 Psuedocode for the Team Model-

Team model {

create thread 1 to n and assign activity 1 to n to each one
signal all threads to start

}

Activity1{
Wait till the thread 1 starts
Do the activity and communicate to synchronize when necessary with other threads
}
.
.
.
Activity n {
Wait till the thread n starts
Do the activity and communicate to synchronize when necessary with other threads

}

3.6 Screenshot

In the implementation of this model, four threads are performing four independent tasks-converting a CSV file to XML, zipping a file, unzipping a file and playing music-concurrently.

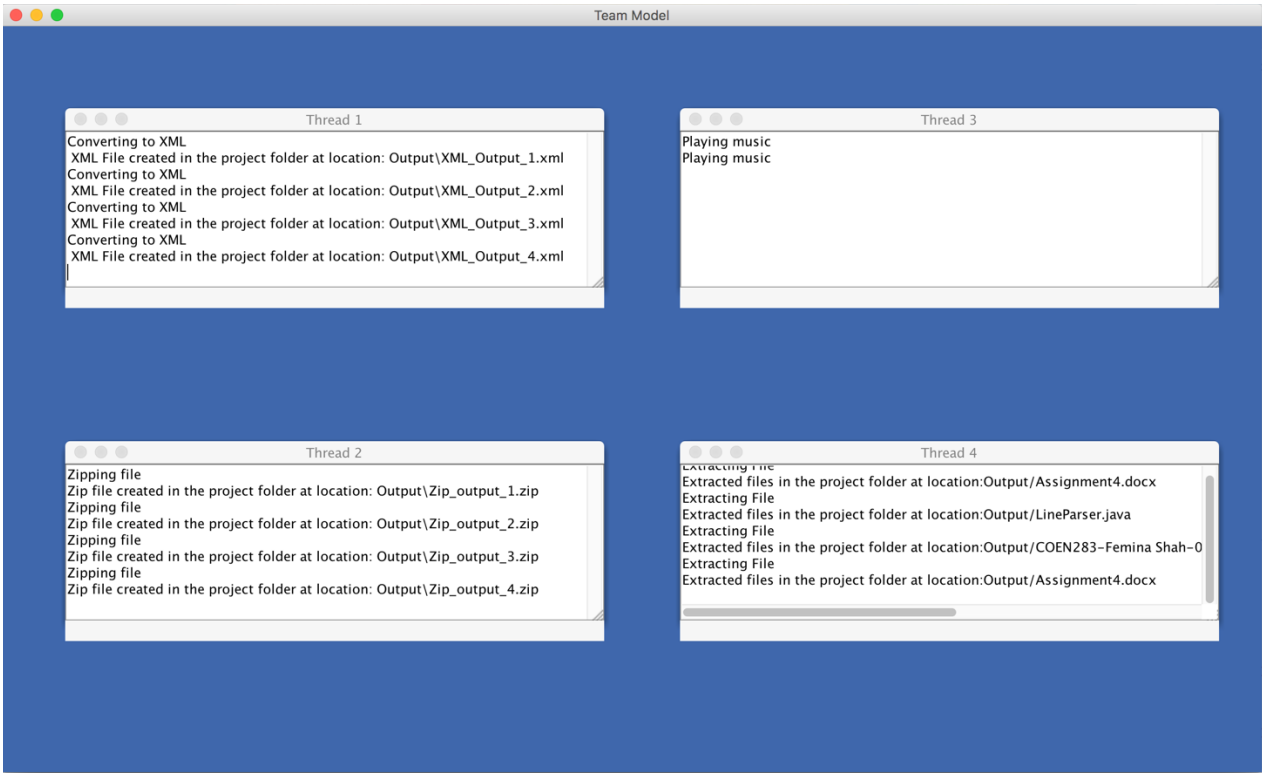

Figure 5: Team Model Screenshot

4. Pipeline Model

In the Pipeline model, all the threads are arranged as a pipeline. From the port, input is given to the designated first thread. Once the first thread completes processing the given input, the result is given to the second thread. Second thread processes its input and passes the output on to the third thread. This process goes on until last thread receives an input from the thread before it. A single output is obtained from the last thread.

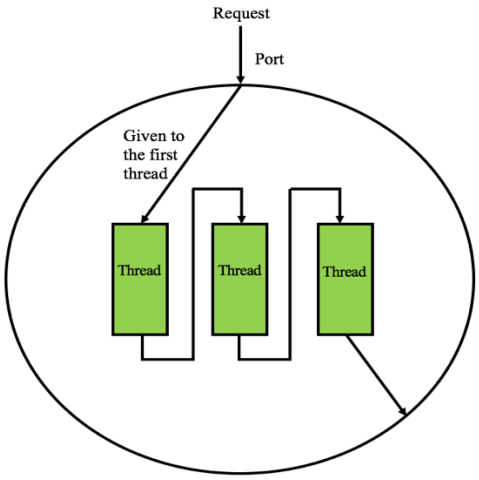

Figure 6: Pipeline Model

The following is an illustration of how the pipeline model works. Let C_i represent a multistep computation on data element *i*. $C_i(j)$ is the *jth* step of the computation. The idea is to map computation steps to pipeline stages so that each stage of the pipeline computes one step. Initially, the first stage of the pipeline performs $C_1(1)$. After that completes, the second stage of the pipeline receives the first data item and computes $C_1(2)$ while the first stage computes the first step of the second item, $C_2(1)$. Next, the third stage computes $C_1(3)$, while the second stage computes $C_2(2)$ and the first stage $C_3(1)$. Above figure illustrates how this works for a pipeline consisting of four stages.

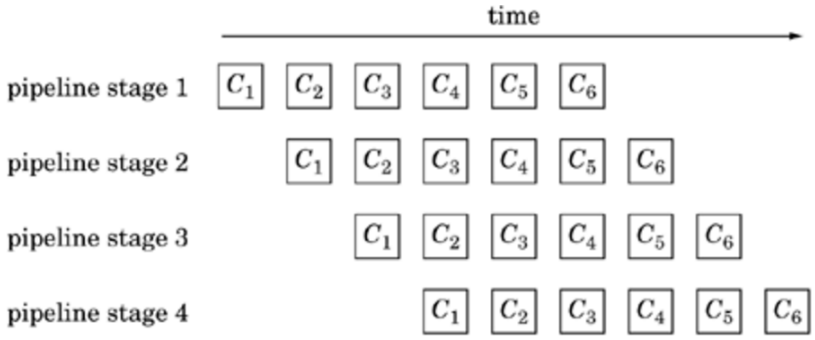

Figure 7: Working of Pipeline model

4.1 Advantages-

- Synchronization required is trivial.
- Overhead for creation of threads is minimum.

4.2 Disadvantages-

- Throughput may be reduced due to a slow moving thread.

- Failure of one thread will lead to overall process failure.

4.3 Implementation Variations-

The pipeline model can also be implemented as Parallel Pipeline Computation Model. Here, the input to the first task is obtained from input devices and the input for the other tasks are the outputs of the previous tasks. Each block in the pipeline represents one parallel, which in turn is parallelized on multiple nodes. The same pipeline is then repeated on subsequent input data sets.

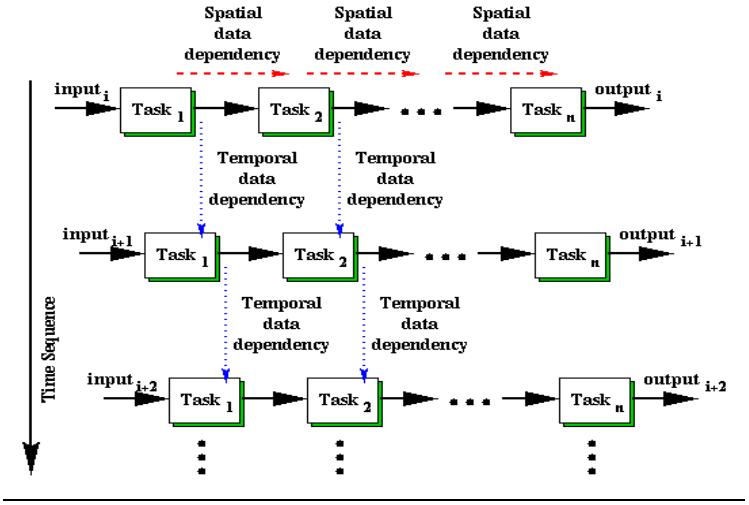

Figure 8: Parallel Pipeline Computation Model

From a single task point of view, the execution flow consists of three phases: receive, compute, and send phases. In the receive and send phases, communication involves data transfer between two different groups of compute nodes. It also involves message packing in the send phase and unpacking in the receive phase. Data redistribution strategy plays an important role in determining the communication performance. In the compute phase, work load is evenly partitioned among all compute nodes assigned in each task to achieve the maximum efficiency. For the parallel systems with multiple processors in each compute node, multi-threading technique can be employed to further improve the computation performance.

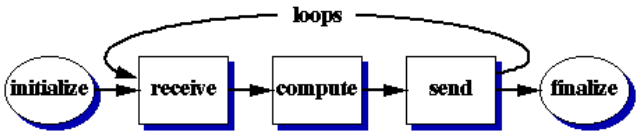

Figure 9: Three Phases for a task

4.4 Applications-

Most CPU architecture is modeled using the Pipeline model. Consider the example of DLX architecture for CPU.

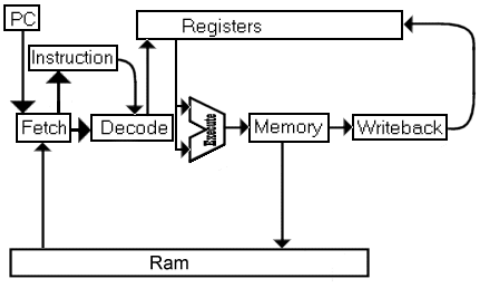

Figure 10: DLX Architecture for CPU

The instruction pipeline is broken up into stages of a pipeline. The stages of the pipeline are-

- Instruction Fetch (IR)
- Instruction Decode (ID)
- Execution (EX)
- Memory Access (MEM)
- Write-Back (WB)

Each instruction is executed in stages. The Pipeline model is most suitable as for a particular instruction, the instruction should be executed only in the order of the stages.

4. 5 Psuedocode for the Pipeline Model

```
processRequest_R1() {
        ...
        while(true) {
                inputStream.read(data, LENGTH);
                resultR1 = processDataR1(data);
                addQueueR1(resultR1);
        }
}
processRequest_R2() {
        ...
        while(true) {
                resultR2 = getFromQueueR1();
                resultR2 = processDataR2(resultR1);
                outputStream.write(resultR2, LENGTH);
        }
}
```

Basic Structure of a pipeline stage

```
initialize
while (more data)
{
  receive data element from previous stage
  perform operation on data element
  send data element to next stage
}
finalize
```

4.6 Pipeline Model Screenshot

Three stages of a pipeline are being implemented. In the first stage, a keyword for which search has to be performed is entered. In the next stage, the keyword is being searched among the contents of the files. In the final stage, the output of the search is displayed and is also written to a file.

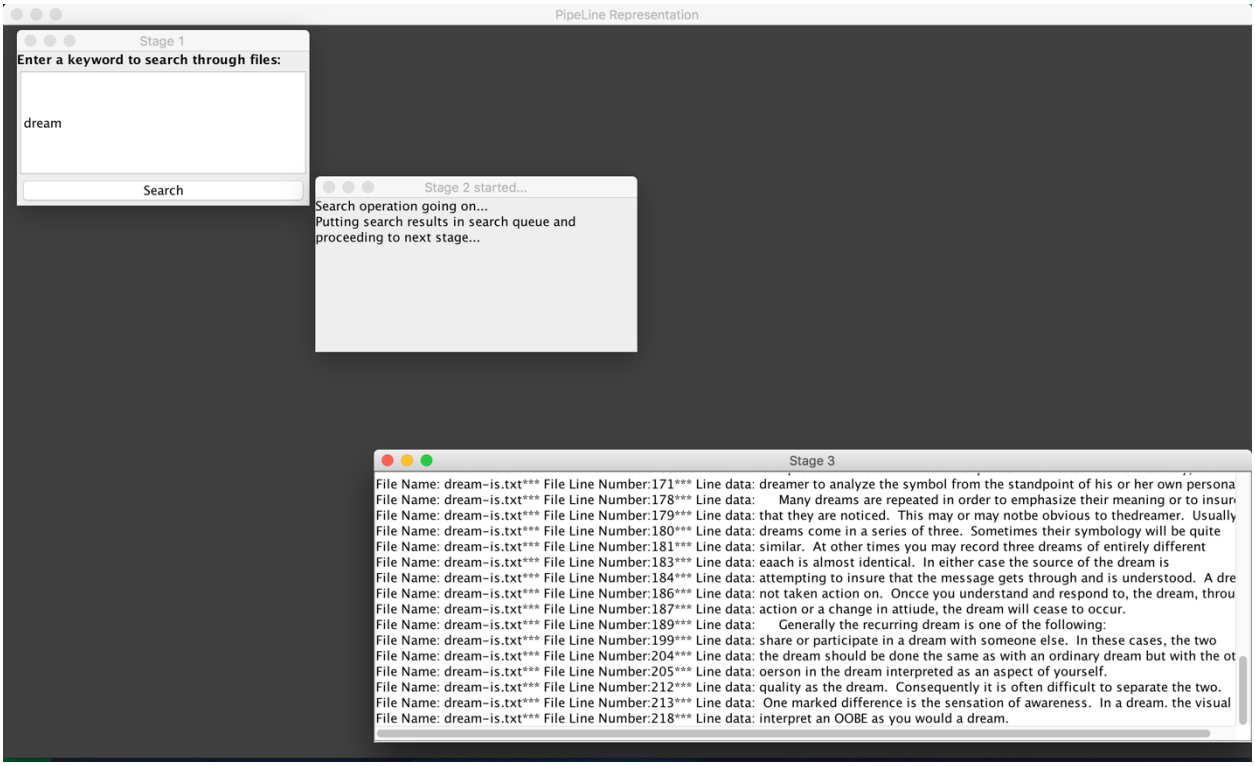

Figure 11: Pipeline Model Screenshot

5. Implementation

5.1 Thread Pools

We make use of thread pools. A thread pool manages all the threads. It is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads. It contains a queue that keeps all the tasks which are required to be executed by the thread. When a thread pool is used, the inherent overhead of time and memory in thread creation is overcome.

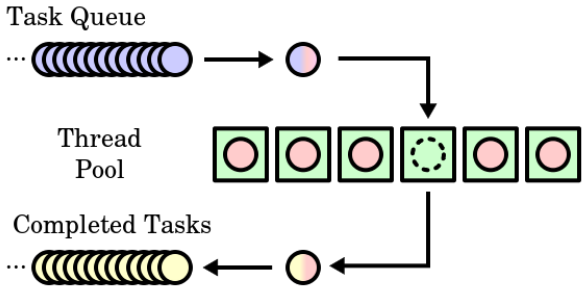

Figure 12: Thread Pool

The threads in the pool take tasks off the queue, perform them, then return to the queue for their next task.

java.util.concurrent.Executors provide implementation of **java.util.concurrent.Executor** interface to create the thread pool in java. A thread pool manages the collection of Runnable threads and executes the threads from the pool created by Runnable interface. Thread pool executors are instances of ThreadPoolExecutor class which manages a pool of threads that are waiting to be executed.

Some of the methods provided by Executor class which were used are-

- newFixedThreadPool(int threads) – Creates a thread pool with fixed number of threads.
- newCachedThreadPool() – Creates a thread pool that creates threads as needed. It can also reuse the previously created threads if they are available.
- newSingleThreadExecutor() – Creates a single thread.
- newScheduledThreadPool(int corePoolSize) – Creates a thread pool that can schedule all the threads to run after a given time delay or execute periodically.

5.2 Blocking Queue Interface

A thread safe queue waits for the queue to become non-empty while retrieving an element and waits for a space to become available in the queue when storing an element.

Blocking Queue is an implementation of thread safe queue. The Blocking Queue interface is a member of the Java Collections Framework.

A Blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

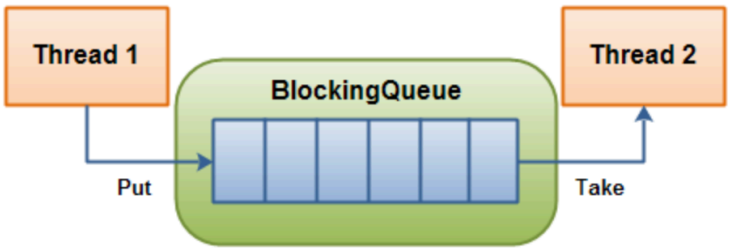

Figure 13: Blocking Queue

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future. These methods are summarized in the following table:

	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

We have used the third method for all model implementations; as it blocks the current thread indefinitely until the operation can succeed and doesn't throw exception or timeout. BlockingQueue implementations are thread-safe. **So all the methods used for queuing in our implementations are atomic using internal locks or other forms of concurrency control.**

The implementations of thread models use the LinkedBlockingQueue and the ArrayBlockingQueue class .Both classes implements the BlockingQueue interface. Linked Blocking queues use the linked structure and it stores the elements internally in FIFO (First In, First Out) order. ArrayBlockingQueue is a bounded, blocking queue that stores the elements internally in an array. Because it is bounded it has a restriction on number of elements which is decided during instantiation. The ArrayBlockingQueue also stores the elements internally in FIFO (First In, First Out) order.

5.3 Producer-Consumer

All the model implementations are based on Producer-Consumer scenario. The producer and the consumer shares the same buffer. The producer produces data and the consumer consumes that data. If there is no data in buffer; consumer waits for the data and when the producer puts in the data and it notifies the consumer about it. Similarly, if there is no place in buffer to put in more data, then the producer waits for consumer to consume some data and then the consumer notifies producer about it.

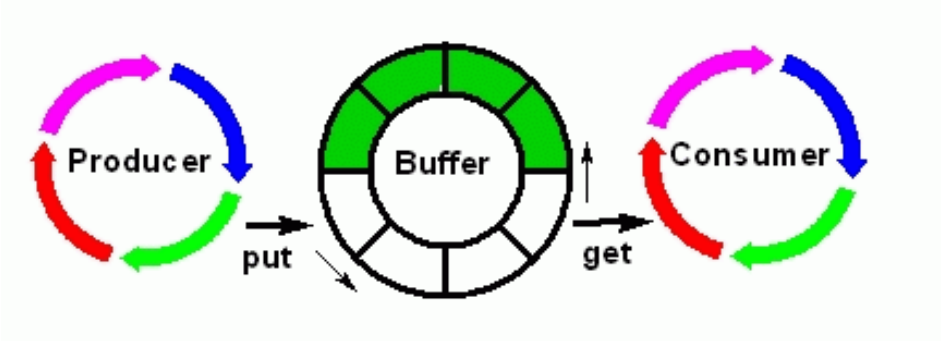

Figure 14: Producer-Consumer Scenario

We can also have one producer and n consumers which we have used in Team model implementation. Here the producer produces data and keeps in the respective queues; and there are n consumers who consumes data from their respective queues.

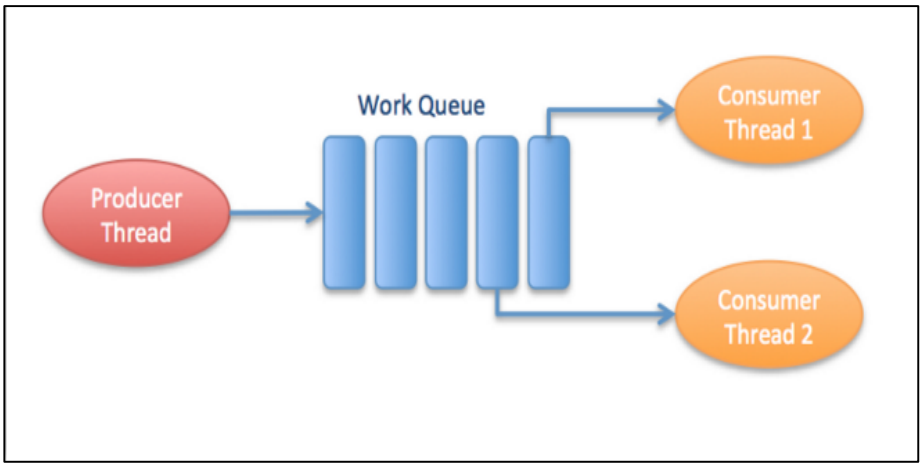

Figure 15: 1-Producer, n-Consumers

6. Combination of Models

The three models discussed form the theoretical basis for the organization of threads. However, in real life applications, the models are seldom implemented as specified. Most applications involve the combination of models. Different models can be combined in arbitrary ways in order and often beyond recognition to fit the requirements of individual, complex applications.

Consider the following example.

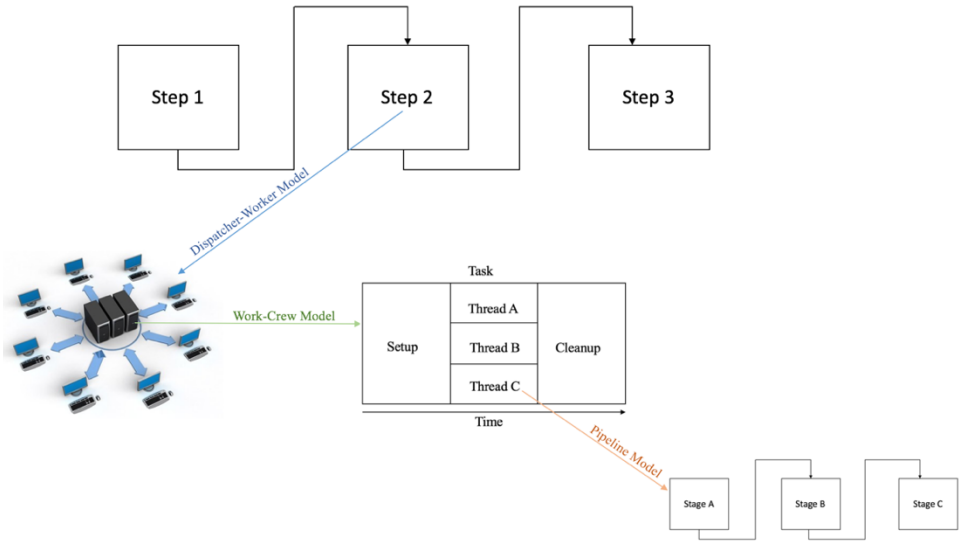

Figure 16: Combination of Models

In this example, different models are nested among each other. A step in the pipeline model could involve requesting a service from a server thread invoking the Dispatcher-Worker model. The server in turn might use a work-crew which is a variation of the Team model. One of the workers in the Team model might require a process to be performed in stages, thus involving the Pipeline model.

7. Conclusion

Each model on its own merits is effective according to the type of application but for most applications, a combination of models is more effective. Different models can be combined in different ways in order to conform to the requirements of specific applications.

8. References

[1] Pthreads Programming *by* Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell

[2] Distributed Operating Systems *by* Andrew Tanenbaum

[3] Distributed Operating Systems: Concepts and Design *by* Pradeep K. Sinha

[4] Programming with POSIX Thread *by* David R.Butenof

[5] Understanding Operating Systems 6 *by* Ann McIver McHoes and Ida M.Flynn

[6] Implementing Thread Priorities in a HTTP Server *by* Soumith Chintala, Yuri Soussov, Randolph Chiu Tan, New York University

[7] Multithreading - An Efficient Technique for Enhancing Application Performance *by* Mrs. M. Shanthi and Dr. A. Anthony Irudhayaraj

[8] Parallel Architectures and Parallel Algorithms for Integrated Vision Systems *by* A. Choudhary

[9]https://www.cs.umd.edu/class/fall2001/cmsc411/proj01/DLX/aboutDLX.html#what

[10]https://www.cs.umd.edu/class/fall2001/cmsc411/projects/DLX/proj.html#3

[11] http://cucis.ece.northwestern.edu/projects/STAP/model.html