Improved Concurrent Java Processes

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Mbalentle Apelele Wiseman Ntlahla

Grahamstown, South Africa
December 2020

Abstract

The rise in the number of cores in a processor has resulted in computer programmers needing to write concurrent programs to utilize the extra available processors. Concurrent programming can utilize the extra processors available in a multi-core architecture. However, writing concurrent programs introduces complexities that are not encountered in sequential programming (race conditions, deadlocks, starvation, liveness, etc., are some of the complexities that come with concurrent programming). These complexities require programming languages to provide functionality to help programmers with writing concurrent programs. The Java language is designed to support concurrent programming, mostly through threads. The support is provided through the Java programming language itself and Java class libraries. Although concurrent processes are important and have their own advantages over concurrent threads Java has limited support for concurrent processes.

In this thesis we attempt to provide the same support that Java has for threads through the `java.util.concurrent` library to processes. This is attempted to be done through a Java library (`za.co.jcp`). The library will provide synchronisation methods of multiple processes, Java process shared variables, atomic variables, process-safe data structures, and a process executors framework similar to that of the executor framework provided by Java for threads. The two libraries' similarities, and performance is analyzed. The analysis between the two libraries is performed to compare the code portability, ease of use, and performance difference between the two libraries.

The results from the project have shown that it is possible for Java to provide support for concurrency through processes and not only threads. In addition from the benchmarks performed the performance of the `za.co.jcp` library is not significantly slower than the current `java.util.concurrent` thread library. This means that Java concurrent applications will also now be able to use cooperating processes rather than be confined to using threads.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System[1] (2012 version, valid through 2020):

- **Software and its engineering → Process management**
- **Computer systems organization → Multiple instruction, multiple data**
- **Computer systems organization → Multicore architectures**
- **Computing methodologies → Concurrent computing methodologies**
- **Computing methodologies → Parallel computing methodologies**

[1] http://www.acm.org/about/class/2012/

Acknowledgements

I would like to give my thanks to everyone who has contributed to the production of this thesis, both directly and indirectly.

I would like to thank and express my deep appreciation to my research supervisor, Prof George Wells, for giving me this opportunity to do this research and providing me with the support, and crucial guidance I needed throughout this research. It has been a great privilege and honour to work and study under his guidance.

I am thankful to my family, friends, and the Rhodes department of Computer Science for the kindness, support, and encouragement that they have given me throughout this period.

I would also like to thank Mr Gys Booysen, and Telkom Centre of Excellence for providing me with the financial support needed to carry out this research.

This work was undertaken in the Distributed Multimedia COE at Rhodes University, with financial support from Telkom SA, Tellabs, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107). The opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard

Contents

List of Figures

List of Tables

Listings

Part I

Research Background

Chapter 1

Introduction

1.1 Research History

Concurrent programming is the future of programming. Concurrent programming allows programmers to utilize multi-core architectures [64]. Currently two forms of processor-level concurrency exist: thread concurrency and process concurrency. Concurrent programming support in Java is limited to thread concurrency and limited support for process concurrency is currently provided by Java [47]. The lack of support for process concurrency in Java may make Java limiting for concurrent programs. This research will attempt to provide greater support for process concurrency in Java.

Since the invention of the computer, there has been a demand for faster processors to decrease the execution time of programs. As the processor (central processing unit [CPU]) has always been thought of as the computer's brains, increasing the speed of the computer has always been directly linked with increasing the speed of the processor. Historically increasing the speed of the processor has been achieved by increasing the clock frequency of the processor allowing for more instructions to be executed in a second, or alternatively decreasing the size of the transistors in the chip, allowing a chip to contain more transistors [64].

On April 19, 1965 Gordon Moore predicted that the number of transistors in a chip would double every year. Doubling the number of transistors in a chip means that consumers would get twice as much computer power at almost the same cost. In 1975 Moore modified his prediction to double every two years. Gordon Moore's prediction became known in computer science as "Moore's Law" [11]. Moore's law held up until the early 2000's; at

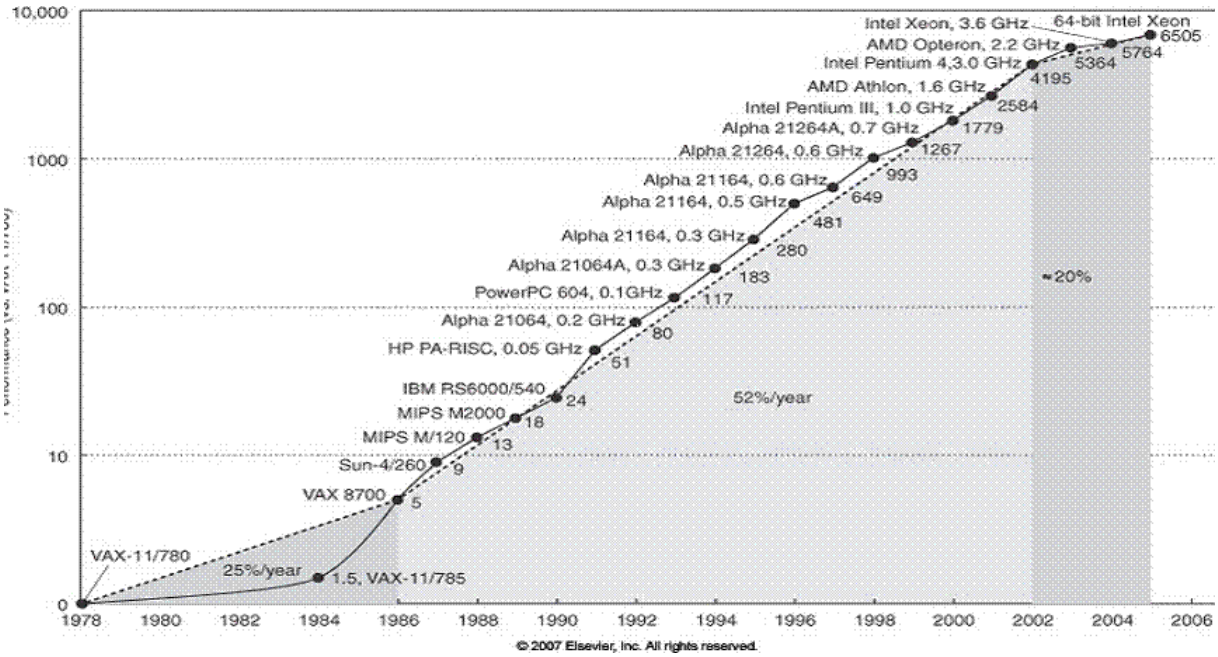


Figure 1.1: Processor speed over time [25]

this time, the rate of increase of processors' speed decreased; this trend is also shown in Figure 1.1 [49]. The reason for the slowed-down increase is because static power dissipation increases as the size of the circuits shrinks [4].

Another alternative for increasing the speed of the processor is to increase the clock frequency of the processor [16]; this became the primary method of improving performance in the late '90's to the early 2000's. However, increasing the processor's clock frequency also has limitation. When increasing the clock frequency, dynamic power dissipation (power leakage), and design complexity also drastically increase. These issues have limited the clock frequency a processor may be clocked at to a limit of around 4 GHz [48].

Due to power dissipation, designers hit the limit on the maximum performance they could get out of a single processor chip. This limit is called the power-wall. As the maximum speed that could be obtained from a CPU has been reached, new innovative ways were needed to continue making computers faster, and for manufacturers to keep their competitive edge on the market [16], [48], [54].

The solution that is broadly employed by modern computer manufacturers to deal with the power-wall issue is a multi-core architecture. The multi-core architecture deals with the power-wall problem by using multiple processors instead of a single processor. The numerous processors allow for the clock frequency to be lowered while still gaining improvement in performance. Improvement in performance is achieved through the multiple

processors in the chip working simultaneously [54]. Intel adopted the multi-core architecture as its long-term solution to continue improving the performance of computers [16].

Although multi-core architectures continue to improve performance without the issues faced by a single-core architecture, an application will not automatically run faster on a multi-core architecture [6]. With architectures containing 2, 4, 8, or 16 cores, the applications will need to be modified to gain more performance on the multi-core architecture [22]. Applications now need to be run in parallel as opposed to the traditional sequential programming. This means that application need to be broken into discrete smaller parts and solved concurrently [64].

A concurrent program is a program that consists of multiple entities (processes, or threads) that cooperate to achieve a common goal. In order to cooperate the entities will share objects, called concurrent objects [53]. Processes contain a complete set of resources that are required to run the process. On the other hand, threads are lightweight and share a single process's resources [65].

Having multiple entities executing at the same time forces concurrent programmers to deal with complexities that are not encountered in sequential programming. Complexities experienced in concurrent programming include non-determinism, communication, synchronization, load balancing, race conditions, and deadlocks. These complexities make concurrent programming more difficult for programmers, than sequential programming. Programming languages and operating systems offer concurrency support to alleviate these complexities from the programmers and assist programmers to write bug free concurrent programs [38], [64].

Currently (2020), the Java programming language is the most popular programming language used by programmers [63]. Java offers support for concurrent programming through concurrent threads and distributed programming; however, Java has limited support for concurrent programming using processes [70]. This means that in order for programmers to use the concurrent support offered by Java they are forced to use threads, even if the program would benefit more using concurrent processes.

The main advantage of concurrent processes over concurrent threads is modularity. Modularity is important as it allows for a program to continue even when one entity crashes and dies. The other entities will not be affected by the entity that has crashed, the functionality the entity provided will be temporarily unavailable until the entity is successfully

restarted. Modularity is native with concurrent processes as each process has its own address space. With concurrent threads modularity is limited, and if a single thread crashes, the other threads will be affected as well, and the whole program will crash and terminate. This type of behavior may be detrimental for some programs, therefore making concurrency with Java undesirable. If concurrency through processes in Java was supported this would allow more concurrent programs to be supported by the Java language [60].

1.2 Research Objectives

The primary objective of the project is to improve the concurrency support of the Java library to natively support concurrent process programming efficiently and as effectively as it supports concurrent thread programming.

The primary objective may be broken into five sub-objectives. These sub-objectives include:

1. Support for process shared variables and atomic variables.
2. Support for process synchronization.
3. Support for process shared data structures.
4. Task scheduling framework supporting scheduling of tasks through processes.
5. Similarity, and performance comparison between the current `java.util.concurrent` library and the `za.co.jcp` (JCP) library.

1.3 Success Case

The project will be a success if it can support the functions required by the projects sub-objectives and meet the project's primary objective. However, we still do not expect the JCP library to be more efficient than the Java concurrent library. The advantage of this library will be that Java applications will now have the option of using processes to perform concurrency, rather than being limited to using concurrent threads. Applications using this library will then gain the advantages of using concurrent processes.

1.4 Thesis organisation

The remainder of this thesis is composed of the following chapters:

Research Background (Chapter 2): This chapter discusses how computer programming transitioned from sequential programming to concurrent programming, and the reasons behind the transition. The chapter goes on to talk about the issues faced by concurrent programmers and what operating systems, and programming languages have done to help with these issues.

Design (Chapter 3): Discusses how the project's objectives will be achieved as well as an overview of the JCP library. This chapter also discuses the software and hardware that is required to carry out the research.

Process Shared Variables (Chapter 4): Variables that are shareable across cooperating Java processes are introduced and discussed in this chapter. This chapter also introduces atomic variables to be shared between cooperating Java processes.

Synchronisation (Chapter 5): This chapter discusses the design and implementation of the synchronisation tools implemented in the JCP library.

Data Structures (Chapter 6): The design and implementation of the concurrent data structures offered by the library are discussed in this chapter.

Executors Framework (Chapter 7): The design and implementation of the executors framework as well as the components that make up the framework are discussed.

System Testing (Chapter 8) How the testing of the JCP library was performed is discussed in this chapter.

Results and Discussion (Chapter 9) The performance of the JCP library is analyzed and compared against the Java concurrent library. The similarity between the code using the libraries, as well as how the code differs, and what changes need to be done to existing code to port to the new library is also discussed in this chapter.

Conclusion (Chapter 10) A conclusion on the findings of the thesis as well as future work is discussed in this chapter.

Chapter 2

Research Background

2.1 Background of Concurrency and Parallelism

First, let us define two related but different terms, concurrent and parallel programming. In concurrent computing, multiple instructions have the potential to be executed simultaneously. On the other hand, in parallel computing, multiple instructions are executed simultaneously [7]. Two kinds of parallelism may exist in an application: data-level parallelism and task-level parallelism. Concurrency and parallelism exist in almost all computation levels, from hardware level to highest levels of programming language abstractions [7], [25], [39].

Data parallelism is defined as splitting the work that needs to be done by sub-dividing the data across multiple processors [42]. Each processor will perform the same task on different pieces of data. An example of data parallelism is having an array of N elements; each element in the array needs to be doubled. Data parallelism may be achieved by dividing the N elements across the available processors, and each process will then sequentially work on a portion of the array. The execution time of the task should be reduced by a factor of the number of processors in the system [50].

Task parallelism reduces the total run time by splitting program execution into parts (tasks) and running the tasks in parallel across different processors. The program is separated into distinct areas of functionality, then computed by different execution entities at the same time [62]. An example of task parallelism is having a an array of images that need to be re-scaled, filtered, and grey-scaled. A pipeline could be created with three processes. One process will re-scale the images, the second process will filter the images,

and the third process will grey-scale the images. The first process will re-scale the image and place it in a collection to be filtered by the following process. The first process will then work on the next image while the image is getting filtered, and grey-scaled by the other processes.

2.1.1 Hardware Parallelism

Numerous computer architectures exist, with each architecture implementing and exploiting parallelism in its unique way. Michael Flynn developed a taxonomy to classify the architectures called Flynn's taxonomy. Flynn's taxonomy classifies the architectures into four distinct categories [5], [9]. The categorization is based on the number of instruction and data streams present in the system [3]. The taxonomy identifies the following categories:

1. **Single Instruction Single Data (SISD):** These are the uni-processor systems. These systems are limited in parallelism; however, they are able to use instruction-level parallelism to exploit parallelism. This is the traditional sequential computer architecture [1].

2. **Single Instruction Multiple Data (SIMD):** SIMD architectures use multiple processors to execute a single instruction on different data streams. SIMD exploit data-level parallelism by applying the same instruction to multiple data items in parallel. There is generally a single control unit and multiple processing units. The control unit reads in the instructions, decodes the instructions, and sends control signals to the processing units. The processing unit accesses the data from memory. Each processor contains a dedicated path to the memory [57], [25]. Examples of SIMD architectures include vector systems and graphic processing units.

3. **Multiple Instructions Multiple Data (MIMD):** These systems contain multiple processors functioning asynchronously and independent from each other. Each processor fetches its own instructions and operates on its own data. This system primarily targets task-level parallelism, although it may also exploit data-level parallelism making this model flexible. This flexibility makes this system the most popular system for general-purpose computers. Examples of systems employing this architecture include multiprocessor systems, clusters, and warehouse-scale computers [25], [41], [31], [68].

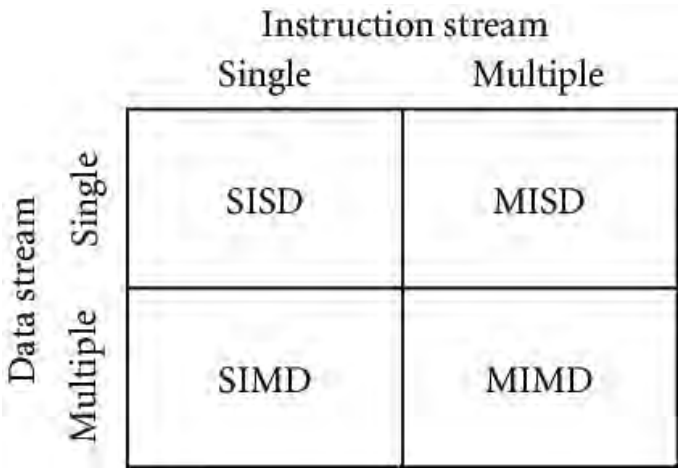


Figure 2.1: Flynn's Taxonomy [8]

4. **Multiple Instructions Single Data (MISD):** There have been no commercial systems built for this category as of yet. MISD is only there for completion of the taxonomy [25].

Flynn's taxonomy describes the four types of architectures that a system may fall into; however, this is not carved in stone as a system may implement a hybrid between the architectures [25]. Figure 2.1 shows a simple visual representation of the taxonomy.

Instruction level parallelism (ILP) incorporates processor and compiler design techniques to speed up execution by causing individual machine operations to execute in parallel. The main techniques used by ILP include pipe-lining, super-scalar execution, and speculative execution [25], [52]. In order for the instructions to be executed in parallel, the instructions are required to be independent of each other, i.e., the instructions should not be referencing the same memory address. This requirement greatly limits the amount of parallelism that may be obtained from ILP [66].

Multiprocessors are defined as computers containing tightly coupled processors that share memory through a shared address space [25]. There are several types of multiprocessor systems, with the most prominent type being the multi-core processor.

A multi-core system is a single chip consisting of multiple cores. The cores are connected to a shared address space (DRAM) and access data from there. Accessing data from DRAM has proved to be a bottleneck and tends to affect the processor's performance as the speed of the processor is vastly greater than that of the memory [37]. To bridge the speed difference between the processor and DRAM a cache is used to speed up data accesses [25], [37]. In modern systems, each core will typically contain its own local cache

to store memory locations that it expects to access in the near future. The processor will now access memory from the faster cache, making data accesses faster. Although the local cache speeds up data access time, they come with a price, namely the complexity of implementing the cache coherence protocol [25].

Cache coherence is a problem that may arise if multiple cores access the same data in their own local cache. If a core modifies the data, the other copies of the data are then stale (no longer valid). If the other cores attempt to access the data from their own local cache, they will be accessing incorrect data (incoherence). Incoherence is prevented with a cache coherence protocol, a set of rules implemented to guide data access in a system. Although the cache coherence protocol deals with the incoherence issues, it in turn introduces overheads [58].

In addition to the cache coherence problem, there is another issue with using multi-core systems. A multi-core system will not automatically be faster than a uni-processor system. The available processors need to be utilized in order to gain the benefits of the multi-core system [48]. Computer hardware works closely with the operating system to ensure the available processes of a system are utilized effectively. An operating system manages running processes, and ensures that the available processors of a system are utilized by assigning tasks to the available processors.

2.1.2 Operating System Parallelism (Linux)

In the early days of computing, computers did not have an operating system. This resulted in computers executing a single program from beginning to end, and the program in execution would have direct access to all the machine resources. Not having an operating system proved inefficient as only a single program could be run at any given time. If the process stalled, usually due to an Input/Output (IO) operation, the whole system would stall. Operating systems were developed to allow for multiple applications to run on a system at once [19]. An operating system is a program that manages the computer's hardware. The operating system shares the hardware between the running programs efficiently and fairly. An operating system may run multiple programs concurrently through time-sharing. Time-sharing gives each process limited time on the computer's limited hardware and removes the process when the time is up and runs another process [55].

Traditional multi-processor operating systems' primary performance concern was the maximization of concurrency for applications running on the system [13], which involves using

the available processors as efficiently as possible. To fully utilize the available processors in a system, there have to be enough execution entities available to assign a task to each processor. Operating systems, therefore, need to provide multi-programming functions to software developers so they may develop concurrent applications to utilize the processors. These include multi-threaded programming functions, inter-process communication (IPC), and process coordination [55]. Different operating systems implement these functions differently. The operating system considered here is Linus Torvald's UNIX (Linux) operating system.

A thread is a unit of execution within a process. Traditionally a process contains a single thread of control resulting in the process executing sequentially. To run a process concurrently, it must have multiple threads of control with each thread performing work. The threads will be managed and scheduled by the operating system to run on the available processors. An operating system provides functionality for programmers to create and manage multiple threads [33].

Multi-threading is an operating system's ability to support multiple concurrent paths of execution within a single process. Uniplexed Information and Computing System (UNIX) traditionally supported only single threads per process. Modern UNIX systems, however, have evolved to support multiple kernel-level threads per process. As Linux is based on UNIX, older versions of the Linux kernel do not support multi-threading [59].

Traditionally UNIX and Linux only supported a single thread of execution per process. To implement threads in a Linux operating system, applications needed to use user-level library functions, with the Portable Operating System Interface (POSIX) threads library (or simply referred to as the pthreads library) being the most popular library (modern versions of UNIX however have started offering threads). Linux's solution is to not differentiate between threads and processes. When creating a new thread a kernel-level process is created. The multiple user-level threads under the same user-level process are mapped into kernel-level processes sharing the same group ID. In Linux processes and threads are created with the `clone` system call. This includes processes created by the `fork` function, and threads created using the `pthread_create` function. The `fork` and `pthread_create` functions trigger the `clone` system call, `pthread_create` passes more arguments to share the virtual memory, file system, open files, shared memory, and signal handlers with the parent process or thread [59] [60].

Creating a new process in Linux copies the current process's attributes and uses them on the new process. It is also possible to clone the process to share resources such as files,

signal handlers, and virtual memory. Sharing the same virtual memory, they function as threads within a single process. When performing a context switch, the Linux kernel compares the address of the page directory of the current process with the next process to be scheduled. If the page directory addresses are the same, the switch is a simple jump [59].

Cooperating processes and threads need to communicate their actions with each other. With threads, this is simple as they share their address space. On the other hand, processes are not able to share memory because of memory protection imposed by the operating systems. Memory protection states that a process may not access memory that has not been allocated to it [56]. This means that cooperating processes need another way of communicating their actions as opposed to using shared variables. The way to do this is to use IPC mechanisms. Linux provides various IPC methods to allow for fast, efficient communication between processes. The IPC mechanisms Linux currently possesses are:

1. **Primitive Communications**
   Primitive Communications are basic limited form of communication mechanisms between processes. Lock files and signals may be used as a basic means of communication between processes.

   Lock files allow processes to communicate using a file at a predetermined location. The processes use the presence of the lock file as a means of communication. Lock files are primarily used to communicate the availability of a resource. This form of communication is limited, similar to that of a binary semaphore [21].

   Processes or kernels may send signals to notify a process that an event has occurred [56]. Signals may be sent by one process to another to communicate. The meaning of the signal ought to be agreed upon beforehand by the participating processes. When comparing signals to lock files, signals are more efficient; however, cooperating processes need to know each other's Process ID (PID) to use signals [21].

2. **Pipes**
   Pipes are unidirectional byte streams connecting the standard output of one process to another process's standard input. A pipe is implemented using two file data structures pointing to the same temporary Virtual File System (VFS) inode. The VFS in turn points to a physical page within memory.

   When the writing process writes to the pipe, the bytes are stored on the page. When the reading process reads from the pipe, the bytes are copied from the page. Having

two processes accessing the page requires synchronization methods. Linux manages access to the pipes using a lock, wait-queues, and signals.

In addition to pipes, named pipes (also called FIFOs, due to the First-In, First-Out nature of the communication) are supported by Linux. FIFOs are handled in a similar fashion to pipes, with the main difference being that the FIFOs are not temporary objects. FIFOs may be created using the `mkfifo` command; the entity will then exist in the file system and may be used by any process in the system as long as they have the appropriate access rights to it [67].

3. **System V IPC**
   The UNIX designers added System V IPCs after finding signals and pipes to be too limited. System V IPC increases the flexibility and range of IPC [21]. The communication mechanisms implemented by the introduction of System V IPC are message queues, semaphores, and shared memory.

   (a) Message queues are used to send and receive messages to and from processes. A message queue is first created using the `msgget` system call. The call takes in a key and an integer value (`msgflag`) . The operating system uses the key to create a unique message queue identifier. The `msgflag` argument is used to determine the access permissions of the queue as well as special creation conditions. The `msgget` call returns the unique identifier created by the operating system, or alternatively a negative one value if the call failed.

   The message struct (structure/template of the message) is defined by a program. The system assumes a message always contains a type field (long integer) followed by zero or more bytes. This means that the first member of the message structure ought always to be a long value, the rest of the message may consist of any valid structure members.

   Messages are placed on the message queue using the `msgsnd` system call. The `msgsnd` system call requires four arguments. The first argument is the message queue identifier obtained from the `msgget` system call when creating the message queue. The second argument is a pointer to the message struct to be sent. The third argument is the size of the message to be sent, which is the size of the message less the size of the type field. The last argument is used to indicate the action the system should take if the system limits for the message queue have been reached. The process may wait by setting this argument to zero, or return immediately by setting `IPC_NOWAIT` as the argument. If the message has been successfully sent, the call returns with zero, otherwise, a negative one

is returned.

Messages placed on the queue may then be retrieved using the `msgrcv` system call. Retrieving a message from the message-queue requires five arguments to be supplied. The first argument is used to identify the message queue, which is the message queue identifier. The second argument is a pointer to the location where the received message should be stored. The third argument is the maximum byte size of the message. The fourth argument is the type of message to be received, this value is used by the system call to identify the message to receive. If the value is zero, the first message in the queue is read in; if the value is greater than zero, a message containing the value supplied as the type is read in; if the value is less then zero, then the first message with a type equal to or less than the absolute value of the value specified is read in. The final argument indicates what should happen if a message is not found in the queue. There are three values that may be supplied, `IPC_NOWAIT`, `MSG_EXCEPT`, and `MSG_NOERROR`. `IPC_NOWAIT` indicates that the call should not block if there is no message. `MSG_EXCEPT` reads in the first message with a type not equal to that of the fourth argument if it is positive. `MSG_NOERROR` truncates messages that are too long. The call returns a zero on success, and a negative one on failure.

(b) Semaphores control multiple processes' access to a common non-shareable resource, thus synchronizing process access to the resource. Semaphores help avoid race conditions and shared data corruption.

A semaphore is a non-negative integer that is stored in the kernel. When using a semaphore to synchronize access to resources, the semaphore is set to the number of available resources. Each time the resource is used up, the semaphore is decremented until it reaches zero. This signifies that the resources have been used up, and any process that attempts to use the resource should wait for it to be freed up by another process first. Freeing up a resource involves increasing the value of the semaphore. A process may gain access to the semaphore using system calls. Using the system calls ensures the semaphore is atomically incremented and decremented.

(c) Shared memory is designed to allow multiple processes to share a virtual memory space as processes are otherwise unable to share memory with another process. A process creates a shared memory segment. During creation, the size and access permissions for the shared memory segment are defined. The shared segment is then attached to the process's current data space. Other

processes may then gain access to the shared memory and map it to their own data space if permissions permit. Processes may detach from the segment when they are done using it. The segment may be removed from the system when it is no longer needed. Generally the creator of the segment is responsible for removing the segment [21], [59].

The `shmget` system call is used to create a shared memory segment or access an existing segment based on the key, and the flags passed into the call. The size of the segment is also passed into the call. The call returns a unique shared memory identifier, which may then be used to access the shared memory [21].

A process may attach the shared memory segment into its data segment using the `shmat` system call. The `shmat` system call takes in the memory identifier obtained from `shmget` to recognize the shared segment. The second parameter passed into the call is the desired location of the shared memory segment. Alternatively, a zero may be supplied, in which case the system will determine the address. The third argument is the flags. The flags are used to specify the shared memory access permissions and request special attachment conditions. Detaching from an address space involves calling the `shmdt` system call. The system call only takes in a single argument, the reference to the attached memory segment [21].

4. **Sockets** The previously discussed forms of IPC mechanisms assume the processes reside on the same host. Interprocess communication may also need to occur between cooperating processes not residing in the same computer host (distributed computing). Sockets allows for processes to communicate regardless of the workstation they may reside in.

   A Socket is an abstract data structure used to create a bidirectional channel between two unrelated processes. The processes may use the channel to send and receive data to each other. When using sockets, the server creates a socket, maps the socket to a local address, and waits for clients' requests. The client will create its own socket and connect with the server using the server's location. The location of the server is determined by its Internet Protocol (IP) address and a port number. Depending on the connection protocol used, the client and server may begin to communicate with each other with or without a formal acknowledgment from the server process. The two protocols typically used for communication between the client and server are the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) [28].

TCP sockets, also called stream sockets, are reliable sockets. Data sent using these sockets will arrive, and in the order it is sent in. Stream sockets are connection-oriented, meaning the processes using the socket create a logical connection [30].

UDP sockets are unreliable sockets. These sockets make no promise of the data arriving, and the data may also arrive out of order. UDP does not create a connection between the process; each message is processed independently [30].

Sockets may be created using the `socketpair` system call. Four arguments are passed into the call. The first argument is an integer value specifying the domain. The second argument indicates the type of socket to use, which may be a datagram socket or a stream socket. The third argument indicates the type of protocol to use. Supplying a zero as the third argument indicates that the default protocol should be used, which will be TCP for a streams socket and UDP for a datagram socket. The fourth argument is the base address of an integer array. The array references the two socket descriptors that will be created if the call is successful [15].

Writing and reading to the socket involves calling the `write` or `read` method, respectively. The methods take in a socket descriptor obtained during socket creation. The `write` method also takes in a character array and the size of the message. The number of characters specified by the size of the message will be taken from the array and written to the socket. Conversely, reading from the socket retrieves the data from the socket and places it in the character array [15].

As an alternative to using the `socketpair` system call, the `socket` system call may be used to create a socket. The `socket` call takes in three arguments identical to those of the `socketpair` call without the fourth argument [15].

5. **Remote Procedure Calls**
   Remote Procedure Calls (RPC) allows for the ease of communication between unrelated applications residing in different distributed environments.

   The model uses a client-server model. The client process (caller) invokes a method on the server process (callee) and waits for the results. The callee executes the method and returns the results to the caller. The RPC resembles a standard local procedure to call to the caller, i.e., the caller interacts with the RPC in the same manner as it would with a local procedure call [61].

   When invoking an RPC method, the client process invokes a local procedure known as a client stub. The client stub will contain the network communication details as well as the actual RPC. On the other side, the server will also include a server stub similar to that of the client stub. The network transport protocol used for

communication is not something that the client and server processes need to concern themselves with [59].

Creating the method stubs usually involves using a protocol compiler. A protocol definition file is written in an RPC language (the language is typically a C-like programming language) and is passed on to the protocol compiler. The file will contain a definition of the remote procedure, its parameters with data types, and its return data type [59].

From this discussion we can see that operating systems have attempted to provide functionality which may be used by concurrent applications, to synchronise their actions, and communicate with each other. However, applications are written in programming languages. The programming languages needs to then provide the functionalities to the programmers.

2.2 Parallel Programming in Languages (Java)

High level programming languages are used to simplify the expression of complex algorithms, increase code portability, and reduce the risk of programming errors. Hardware and operating systems provide the functionality to allow programmers to write and execute parallel programs; however, a parallel program contains complexities that are nonexistent in a sequential program. Such complexities can include but are not limited to, creation and destruction of concurrent threads of control, synchronization of thread activities, thread communication, and so forth. Languages play a role in simplifying and mitigating the complexity arising from parallel programming [10]. Numerous languages support parallel programming, but the language that is under consideration is Java.

2.2.1 Java Concurrency Support

In this section, we will be looking at Java and how it supports parallel and concurrent programming. As stated in Section 1.1 Java provides concurrency support through the use of multithreading and distributed programming, with limited support for multiprocessing applications.

Java Threads

Threads in Java are integrated into the language. In a Java application, there is always at least one thread to execute the code. The thread will start the execution of the `main` method. In addition, some of the Java classes are multithreaded; therefore, a developer may be creating and using threads implicitly. A developer may also explicitly create threads through the use of the Java `Thread` class, `Runnable` interface, or the `java.util.concurrent` library [44].

In Java, multithreading may be achieved by extending the `Thread` class and overriding the `run` method. A thread may then be spawned from the derived class. To spawn a new thread from the class, an instance of the class needs to be created; and the `start` method inherited from the `Thread` class is called on the class's instance. A new thread is then spawned, which will begin the execution of the `run` method. The thread will terminate after it has completed executing the method. Listing 2.1, and Listing 2.2 show an example of how a thread is created using the `Thread` class. In the example, a thread will be created to print out the phrase "Hello World" a hundred times then terminate. The main thread, in the meantime, will continue its execution while the second thread prints out the phrases [44], [19]. If the main thread needs to wait for the execution of the thread to complete before continuing, the `join` method could be called on the instance used to create the thread. In addition, the main thread may also prematurely terminate the thread by calling the `stop` method on it, although this has been deprecated as the method is inherently unsafe [45].

```
1  public class ThreadClass extends Thread {
2      public void run() {
3              for (int I = 0; I < 100; I++) {
4                  System.out.println("Hello World");
5              }
6          }
7      }
```

Listing 2.1: Example of a Java Thread Class

```
1  public class mainClass {
2          public static void main(String args[]){
3              ThreadClass threadClass = new ThreadClass();
4              threadClass.start();
5          }
6      }
```

Listing 2.2: Java Main Class to use Thread Class

In Java a class may only extend a single class. If a class extends the `Thread` class, that means it may not extend any another class, and this may prove to be limiting at times. A preferred alternative to creating a thread is to implement the `Runnable` interface. Implementing the `Runnable` interface requires the `run` method to be implemented. The class may then be used in a similar fashion to that extending the `Thread` class [19].

Executor Framework

Creating threads using the two methods discussed requires a thread to be created every time the `run` method needs to be run, which may prove to be inefficient at times. Consider the classic example of a web-server that creates a thread every time a new request is received. The thread will handle and service the request while the main thread continues to prepare for other requests. Although this system does work, it will greatly struggle under a heavy load. The reason for this is because the overhead of creation and termination of threads can become excessive. Threads also consume resources. Having plenty of idle threads may use up a lot of memory and forces the threads to compete for CPU time, which may have performance costs [19].

The Executor framework provides a solution to constantly creating threads through the provision of thread pools. Thread pools allow for threads to be created and reused when they are needed. The threads may be created using one of the thread pool factory methods. After the threads have been created, they wait for tasks to be submitted to the pool. When a thread finishes executing a task, it does not terminate; it waits for another task to be submitted to the pool. The pool's size depends on the implementation of the pool, although most implementations keep the size constant, restarting threads that have prematurely terminated [20].

The tasks to be submitted are classes implementing either one of two interfaces (`Runnable` or `Callable`). Tasks implementing the `Runnable` interface are similar to those discussed previously. A class implements the `Runnable` interface and implements the run method. Alternatively, a class implementing the `Callable` interface may be submitted for execution. The `Callable` interface behaves the same way as the `Runnable` interface. The primary difference between the two interfaces lies in the implemented methods the thread will run. The `Callable` interface implements and runs a `call` method. The method differs from that of the `run` method in that the method allows for the return of a value from the method back to the caller. This provides flexibility to the system, as in some

situations it is desired to have a value returned by an executed task. The returned value is obtained through a `Future` interface [19], [20].

A `Future` object is obtained when submitting a `Callable` task to an executor. From the acquired Future, it is possible to control the execution and status of the task and obtain the result of the call after its execution has been completed. The control operations available from the future include canceling the task for execution, checking if it has been canceled, and checking if it has completed execution [20].

2.2.2 Remote Method Invocation

Remote Method Invocation (RMI) allows for an object running in one Java virtual machine (JVM) system (client) to access and invoke the methods of an object running on another JVM (server). The server will run the request of the client and return the results back to the client. The JVMs may reside on the same host or on different hosts [72].

Invoking an object through Java RMI is similar to the RPC system adopted by the Linux operating system discussed in Section 2.1.2; the Java RMI also uses the stubs previously discussed in Section 2.1.2.

With Java RMI, the client and the server are connected and communicate through Java sockets. The request, along with the method's arguments, is sent to the server through a socket, and the server returns the result through the socket. The communication between the client and server is all taken care of by the RMI system, and is hidden from the programmer; the programmer simply calls the remote object's method as if it was a local object [23].

2.2.3 Concurrency Issues and Advanced Synchronisation Methods

Writing concurrent/thread-safe code is fundamentally managing access to a shared mutable state [19]. This is done through the use of synchronization mechanisms to coordinate access to an object's state. Failing to synchronize access to a shared state may result in concurrency problems. Concurrency problems that could arise are race conditions, deadlock, starvation, and livelock problems. Programming languages need to provide synchronization mechanisms to programmers so they may avoid these issues.

Race conditions

The definition of a race condition is a condition of a program where its behavior is dependent on the timing or interleaving of multiple threads, or processes [19]. Race conditions arise in parallel programs when access to shared memory is not explicitly synchronized [43]. There are two types of actions that may result in a race condition occurring. The two types of actions that result in race conditions include "check-then-act" operations and "read-modify-write" operations [18].

Listing 2.3 shows an example of "check-then-act" operation that could result in a race condition. Consider two threads, both executing the code. One thread checks if the instance object is null and finds that it is null and creates it, which the program intends to do. Consider another scenario whereby the second thread checks if the object is null while the first thread is creating it. The second thread will find the object to be null as well and also create the object. As this is not the program's intended purpose, it is said to contain a bug, or the program is not correct.

```
1    public class LazyInitRace {
2        private ExpensiveObject instance = null;
3        public ExpensiveObject getInstance() {
4            if (instance == null)
5            instance = new ExpensiveObject();
6            return instance;
7        }
8    }
```

Listing 2.3: Race condition example [19]

From the race condition example in Listing 2.3 we can notice that the race condition occurs because the check and creation of the object were done one after another. In order to fix this issue, the actions need to be atomic (indivisible actions). If the check and the creation of the instance in the previous example were atomic, then a race condition would not occur. To ensure thread-safety, "check-then-act" operations and "read-modify-write" operations need to always be atomic [19].

Java provides a built-in mechanism for enforcing atomic operations, the built-in synchronized block, also referred to as an intrinsic lock. An intrinsic lock guards an entire block and ensures that only a single thread may be executing the block at a time. To use the intrinsic lock, an arbitrary Java object is required to act as the lock. The lock is automatically acquired when the thread enters the block and automatically released when exiting

the block. An intrinsic lock may also be used at the method level, in this instance the method is said to be a synchronised method and only a single thread may execute the method at any given time [36].

An intrinsic lock is useful for making a block of code atomic to guard a shared resource. However, it may prove to be limiting at times as it does not allow for a thread attempting to acquire the lock to cancel the operation if the wait takes too long to conclude. Since Java 5.0, another locking option has been added, ReentrantLock, for added functionality. In addition to added functionality, the ReentrantLock also requires locks and unlocks to be explicit. The most notable added functionalities are the polled, timed, and interruptible lock acquisition mechanisms [19], [32].

Deadlock

Locks are useful to handle race conditions in concurrent applications. However, applications using locks are prone to deadlock if the lock acquisition is not handled properly. Deadlock is a situation whereby two or more threads are unable to continue because they cannot acquire the locks required to continue. A classic example of a deadlock is where thread 'A' holds lock 'x' and needs lock 'y' to continue. At the same time, Thread, 'B' holds lock 'y' and needs lock 'x' to continue. At this point, neither thread may proceed as they need a lock held by the other thread. A process in this situation is said to be in deadlock as it can not proceed [29].

In order to detect and avoid the possibility of a deadlock in a concurrent application, a resource allocation graph may be used. Resource allocation graphs contain a set of vertices and a set of edges. There are two types of vertices: resource vertices and thread vertices. The resource vertices will typically represent the locks in the system, and the thread vertices will represent the threads that may exist in the system. There are also two types of edges, a request edge and an assignment edge. The request edge indicates the thread attempting to acquire a resource, and the assignment edge represents a thread holding a resource. The direction of the edge denotes the type of the edge. If the edge is directed from the thread to the resource vertices, the edge denotes a request edge, and vice versa [26].

Two resource allocation graphs are depicted in Figure 2.2. In order to determine if a system may deadlock or not, the graph is analyzed. From the analysis of the graph, we realize whether or not the graph contains cycles. If the graph does not contain any cycles, the system will not deadlock. The system may deadlock if there are cycles in the graph.

Figure 2.2a depicts a graph that contains a cycle. From the figure, Process one (P1) is requesting resource one (R1) while R1 is allocated to P2, making it unavailable. At the same time, R2 is being requested by P2 while the resource is allocated to P1, making it unavailable as well. Both processes may not proceed as they cannot obtain the resource they need because it is unavailable. This scenario depicts the deadlock example given before; the graph also proves that a system containing cycles is prone to deadlock.

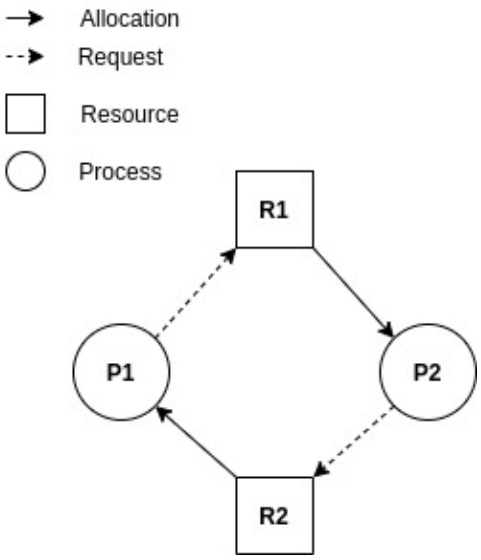
Figure 2.2b shows a scenario where the system will not deadlock. The graph shows P2 attempting to acquire resource R1, which is allocated to P1. P1, on the other hand, is attempting to acquire resource R2. As R2 is not allocated to any process, P1 will eventually acquire R2 and finish what it is doing, eventually freeing R1 allowing P2 to proceed. As both processes will eventually be able to proceed and do their work, the system is not deadlocked, which is also depicted by the graph.

If threads are acquiring the same locks, they should acquire the locks in the same order; otherwise, a deadlock may occur [19]. This is evident in Figure 2.2 as in Figure 2.2a the threads acquire the locks in opposite order, while in Figure 2.2b the locks are acquired in the same order, making the program deadlock free.

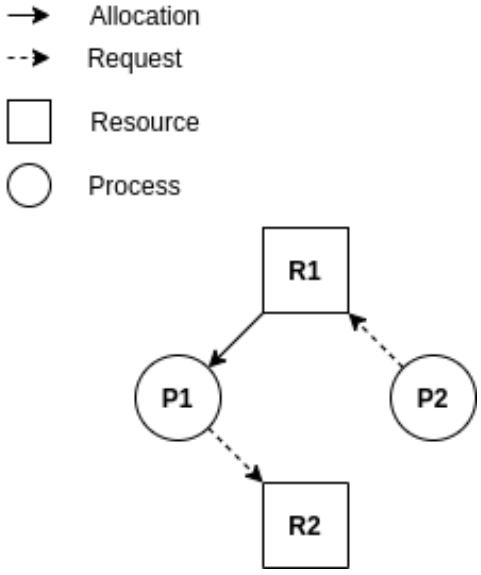
Other concurrency issues closely related to deadlock include livelock and starvation. Starvation occurs when there is a thread that is denied access to resources it needs in order to make progress. The most common resource a thread may be starved of is CPU cycles. Livelock is a situation whereby a thread may not proceed even though it is not blocked. The process is stuck because it keeps attempting an operation that always fails [17], [19], [35].

Using locks to synchronize and manage the state of a dataset may prove to be cumbersome in some instances. Java 5 introduced synchronized collections and other advanced synchronization methods to allow for easier synchronization of threads.

Synchronized collections are built-in Java collections that allow for the synchronization of data sets. This makes it easier to manage and synchronize large data sets that are accessed by different threads. The synchronized collections include `Vector`, `Hashtable`, `ConcurrentHashMap`, `BlockingQueue`, and `CopyOnWriteArrayList` data structures. Having these thread-safe data structures allows the programmer to use them without dwelling on the synchronization mechanisms of the threads accessing the data structure. Each of these data structures has its own advantages and disadvantages, allowing one to be better suited than the other in different scenarios, and making the library more versatile to deal with different problems [19], [20].



(a) Deadlock



(b) DeadLock Free

Figure 2.2: Resource Allocation Graphs

Java includes multiple synchronization methods that may be used to control the flow of threads. These include semaphores, barriers, and latches. A latch is a synchronizer that acts as a gate, blocking threads until it is open. After it is open, all waiting threads may then proceed. Latches are useful for waiting for resources to be available. A semaphore controls the number of threads that may enter a critical section at the same time. Barriers are similar to latches in that they block multiple threads until a certain event has occurred. However, barriers differ from latches in that they wait for threads to reach an execution point as opposed to a resource to be available. A barrier allows for a set of threads to wait for each other to reach a common barrier point before any thread may proceed beyond that point [12].

2.3 Summary

In this chapter we have discussed the history of computers and how it has affected the computers of today, in particular how multi-core architectures have became popular over last few decades. In addition the chapter discusses how programs being written today have had to adjust and exploit multi-core architectures.

Computer programs of today need to be concurrent (multi-threaded, or multi-processing) to fully utilise the numerous processors available for use in a multi-core system. However writing concurrent programs has proved to be difficult and contains complexities not evident in sequential coding mainly race conditions, deadlock, liveness, and starvation. Therefore operating systems, and programming languages need provide support to software developers so they are able to deal with the complexities of concurrent programming.

Java provides great support for multithreading concurrency, and distributed computing. However, Java has limited support for process concurrency. As multiprocessing has its own advantages compared to multi-threading this may prove to be limiting for programmers wanting to make concurrent programs using Java. Therefore a Multi-processing library would be beneficial if implemented for Java. This thesis attempts to provide the multi-threaded functionality provided by Java to multi-processing concurrency. In the chapters that follow the design and implementation of the library is discussed.

Part II

Design & Implementation

Chapter 3

Design

This chapter discusses and explains the approach taken to perform this research. The overview of the planned library is discussed in this chapter, as well as the modules that make up the library. The chapter also covers the hardware and software required to perform this research.

3.1 High-level Research Design

The primary objective of the research (discussed in Section 1.2) is to improve the concurrency support of Java to natively support multi-process programming. Multi-processing support will be provided through a Java library called Java Concurrent Processes (JCP) (`za.co.jcp`). The JCP library will closely follow the existing Java concurrent threads library (`java.util.concurrent`). The classes implemented in the JCP library will be similar to those offered by the concurrent threads library. This will allow programmers who are familiar with the concurrent threads library to easily adapt to the use of the JCP library. Although Java is a cross-platform language and may be used on various operating systems, the project will specifically target the Linux operating system and attempt to exploit the IPC mechanisms offered by Linux.

3.2 JCP Library Overview

The JCP library will offer classes similar to those offered by the Java concurrent threads library. The JCP library will offer the `java.util.concurrent.atomic` package through

the `za.co.jcp.atomic` package. The synchronisation classes offered in the `java.util.concurrent` library will be offered in the `za.co.jcp.synchronisers` package. The library will also offer the `BlockingQueue`, and `ConcurrentMap` classes of the concurrent collections of the concurrent threads library through the data structure package (`za.co.jcp.datastructure`). The data structure package in additions to these two data structures will also implement list and array data structures. A framework to execute tasks concurrently similar to that of the concurrent thread library will also be provided by the JCP library through an executors package (`za.co.jcp.executors`). The JCP library in addition to the functions provided by the concurrent threads library will provide a shared variables package (`za.co.jcp.sharedvariables`). The shared variable package will provide variables that may be shared and accessed by cooperating processes in the system.

The JCP library contains four distinct modules to be implemented (process shared variables, synchronisation mechanisms, process shared data structures, and an executor service framework). The design and implementation of these modules will be discussed in more detail in the following chapters.

The Java platform is a programming environment which consists of the Java virtual machine (JVM), and the Java Application Programming Interface (API). Java programs are coded in the Java programming language, and compiled into a machine independent binary class format. A Java class file may be executed on any machine with Java platform installed through the JVM. The Java API contains a set of built-in classes to be used by Java applications [34].

To provide support for process concurrency the library will use the IPC mechanisms offered by the Linux OS, discussed in Section 2.1.2. However, as these IPC are Linux dependent and Java is machine independent, Java does not offer these IPC mechanisms. To access these IPC mechanisms we need to use another programming language (C++) to write the libraries. These may then be accessed and used in Java through the Java Native Interface (JNI) [34] [69].

The JNI is a feature of the Java platform that allows applications to incorporate native code written in another programming language (mainly C, or C++), with code written in Java. The native code will typically be a C/C++ library with functions which may be called by a Java class. The passing of the arguments and acceptance of the return values will be handled by the JNI [34] [69].

Most of the the C++ functions used by the JCP library require a key to be passed in as

an argument. To assist in creating a key the JCP library contains a `getKey` static method in the `ConcurrentProcesses` class. The `getKey` method creates a unique key from an existing file and a unique character ID. The method in turn uses a JNI library to call the `ftok` function using the file path and character ID passed in as arguments to the `ftok` function.

The library contains a `SystemException` class to indicate something has gone wrong while performing an action. The exception is needed because most of the classes will make use of C functions (for example the previously discussed `ftok` function), which may fail. When a C function fails, it will return an error code indicating the failure, and the reason for failing stored in `errno`. An exception is then required to be thrown in the Java code to indicate failure of the operation and the reason for failure. To throw an exception the `SystemException` class is used. The exception message (obtained from calling `strerror` function with `errno` as the argument) is used to return the reason for failure.

The JCP library contains a `Serializer` Java class. The `Serializer` class is used to serialise and de-serialise objects to and from String objects. The class is implemented as there are a lot classes in the library that require an object to be serialised into a String object for storing and sharing the object with other processes. The class contains two static methods, `serialize`, and `deserialize`. The `serialize` method accepts an object of type `Object` (i.e. any type of object), serialises the object to a String and returns the serialised String representation of the object. The `deserialize` method accepts a String value and de-serialises the value to an `Object` type, the de-serialised object is then returned.

Some of the functions used in C libraries require an absolute time to be passed in, while its JCP counterpart methods use a relative time (for example the `await` method, of the `Condition` class, and the `pthread_cond_timedwait` function). This requires the offset time to be added to the current time to obtain an absolute time. The library has a C `getOffsetTime` utility function to add the offset to the current time of execution, and return the absolute time.

3.3 System Infrastructure

Various hardware and software are required for the development, testing, and benchmarking of the library. This section will cover the computer hardware and software that is required to perform this research.

3.3.1 Hardware

The hardware requirement of the research include a computer with a multi-core architecture. A multi-core architecture is required as the project will require multiple processors to run the multiple processes used by the library during the testing, and benchmarking phases.

The specifications of the computer hardware chosen for the project is shown in Table 3.1.

Component	Description
Motherboard	GIGABYTE AB320M-S2H
Processor	AMD Ryzen 5 1600
Cores	6
Ram	16 GB
Ram type	DDR4
Graphics	GeForce GT 710

Table 3.1: System hardware specification

The chosen computer architecture satisfies the need for a computer with a multi-core architecture as the processor contains six cores and twelve threads. Therefore this computer hardware will be able to perform the tasks required by the project.

3.3.2 Software

The software requirement of the project is a machine with the Linux operating system installed. The Java development kit (JDK) needs to be installed on the machine to compile and run the developed classes. Furthermore the Java version that needs to be installed on the machine needs to have the `java.util.concurrent` library on it as the library will be used in the benchmarking of the project. As the project will be compiling and using C, and C++ libraries, GNU Compiler Collection (GCC) is also required to be installed in the machine. To write and edit the code of the project a text editing software is required.

The software installed on the system is shown in Table 3.2.

The operating system installed (Ubuntu 18.04.5 LTS) is a Linux operating therefore satisfying the requirement of a Linux operating system. The Java version used includes the `java.util.concurrent` library fulfilling the requirement that the Java version used must

Software	Version
Operating System	Ubuntu 18.04.5 LTS
Java version	openjdk 11.0.9.1
GCC version	7.5.0
Text editing software	Visual Studio Code

Table 3.2: System software specification

include the concurrent library. The GCC version used is the latest released version as of January 2020. The code editing tool used in this project is Visual Studio Code which is a free, cross-platform source-code editor offered by Microsoft.

3.4 Summary

This chapter presents an overview of how the research of this thesis will be conducted. The library (JCP) to be built is also introduced in Section 3.2. Moreover, the approach the JCP library will take to support Java concurrent processes is discussed in this chapter. The chapter goes on further discuss the high-level overview of the JCP library. The hardware and software required to carry out the research is also presented in this chapter.

Chapter 4

Process Shared Variables

Most applications' cooperating tasks need to share variables; this is done through sharing an address space. This means that multiple tasks need to have access to the same address space. With multi-threaded applications, this is not an issue as threads share the same address space. However, with multi-processing applications, this proves to be a difficult task due to the restrictions imposed by the operating system's memory protection mechanism [56]. In this chapter the implementation of process shared variables is discussed. In addition to discussing process shared variables, the design and implementation of atomic variables is discussed in this chapter.

4.1 Shared Variables

The JCP library should enable Java processes to share a variable. The data types the library ought to support are boolean, int, double, long, char, and String types. The data types should be assignable an initial value, the value of the variable should be modifiable, and retrievable from any Java process running on the system.

Each shared variable data type will be implemented in its own Java class. JCP (Java Concurrent Processes) will be prepended to the default name of the type. For example, with the boolean data type, the name of the class will be `JCPBoolean`.

The variables will have a method to retrieve the current value stored in the variable, the method's name is `get`. Java basic data types will be returned by the `get` method and not the JCP type. Returning the Java basic data type will allow the returned value to

be used in the rest of the Java program without having to perform additional casts and conversions on the value. The method declaration of the `get` method for the `JCPBoolean` class is:

```
public boolean get();
```

A `set` method is also required to set the value after the object has been initialized. The method will accept the relevant Java basic data type. The value may then be retrieved by any Java process with a reference to the variable. When the variable value is set, the previous value shall be discarded and will not be retrievable after the `set` method has been called. The method declaration of the `set` method for the `JCPBoolean` class is:

```
public void set(boolean value);
```

A String in Java is immutable [24], therefore the `set` method is omitted for the `JCPString` class. The value of the string may only be specified at initialization time; after that, the string may only be retrieved using the `get` method and not modified. In order to modify a string, the current string will need to be destroyed as a string is immutable. Failing to first destroy the existing string and creating a new string while the current string exists will cause a `SystemException` to be thrown.

There are two constructor methods for the classes. The first constructor takes in a long key (for brevity this constructor is called the connect constructor). The connect constructor does not create a shared variable. The connect constructor attempts to obtain a reference to an existing shared variable. The second constructor in addition to taking a long key, it takes in a value the shared variable will be initialised to (for brevity this constructor is called the create constructor). The create constructor attempts creates a shared variable, if the key supplied in this constructor is already in use the constructor throws a `SystemException`.

The shared variable needs to be accessed by different Java processes running in the system, shared memory is used to store the variable and allow multiple processes to access it. Shared memory is created using a unique key, with the create constructor. In the case of the connect constructor the key is used to obtain the identifier of an already existing shared memory segment. As previously explained, the key used to create the shared memory or gain access to existing shared memory is passed on as an argument by the client at the initialization of the variable.

Since the function to create shared memory is not available in Java, a C library is created to perform the functions not available in the Java language. The Java class then calls

the C library through the JNI to perform the required actions. The actions performed by the C library include creating shared memory, assigning the value to shared memory, retrieving the value, and detaching shared memory. If the C library encounters any issue while performing the required function, a `SystemException` is thrown back to the Java class to be handled by the client application. Issues that may be encountered by the C library include attempting to use a an invalid key when creating the shared memory.

The constructors use the `shmget` function (Section 2.1.2) to create or obtain the identifier of an existing shared memory . To create the shared memory, the arguments passed in to the `shmget` function are:

1. A unique key.
2. Size of the struct used to store the variable.
3. `IPC_CREAT | IPC_EXCL | 0666`

The unique key is the key passed in is the key passed in as an argument to the constructor. The size of the struct of the struct is obtained from the shared variable data type. The size of the struct is obtained from the C `sizeof` function. The `IPC_CREAT | IPC_EXCL` flags allow for the shared memory to be created but throw an exception if the shared memory already exists. The `0666` flag allows the shared memory to be accessible by any process running on the system, as long as they have the key. To connect to existing shared memory, the flags passed in to the `shmget` function are modified. The only flag passed in is the `0`. This attempts to obtain the identifier of an existing shared memory

The `JCPString` class has an extra protection mechanism to the other JCP classes. The `JCPString` type has a read-write lock associated with it, implemented in the C library (stored in the shared memory struct). The lock is primarily intended to block processes from accessing the String value before it has been fully initialized. This is because it may take a while for a string to be initialized and if there are processes waiting to read the value of the `JCPString` they may end up reading incorrect data, causing a race condition. As the lock is a read/write lock after the string has been initialized, the processes will not block to read; therefore, the only additional overhead that will be experienced is the overhead of locking and unlocking the read-lock. Presumably there is some overhead to check the lock status on subsequent reads.

4.2 Atomic Variables

Process shared variables provide a variable that is accessible to different Java processes running in a system; however, these variables are prone to race conditions and are not safe to use without additional protection. Synchronization methods need to be used in order to ensure safety in shared variables. Atomic variables have been developed to be process-safe without the use of synchronization methods.

Atomic variables provide the same functionality as shared variables previously discussed; however, they go a step further and provide process-safe features. Synchronization of the variable is handled by the library in atomic classes. The data is stored and retrieved in an atomic operation, therefore preventing a race condition from occurring. The GCC built-in functions for atomic memory access are used to retrieve and perform these operations. The GCC functions are again accessed through a C library called using the JNI.

The atomic types that are supported by the library are boolean, integer, and long. These types are implemented in `AtomicBoolean`, `AtomicInteger`, `AtomicLong` classes respectively. The basic functions supported by all the Atomic classes include:

1. `get`:
   Retrieve the value currently set in the atomic variable. This function is performed using the GCC "`__atomic_load_n`" function. Using the GCC function to retrieve the variable stored in the atomic variable allows the operation to be performed in an atomic manner

2. `set`:
   Update the value currently stored in the atomic variable to the supplied value. Similarly to the `get` method this method also uses a GCC function ("`__atomic_store_n`") to perform the action in an atomic manner.

3. `getAndSet`:
   Retrieve the currently set value and update the value to the supplied value. As these are two different operations that need to be performed by the library some form of synchronization is required for this function. An atomic swap operation may be used to perform this function correctly. The swap operation used is the "`type __atomic_exchange_n (type *ptr type val, int memorder)`" function which is an atomic exchange operation. The function stores the new value in a specified location and returns the old value stored in that location.

4. `compareAndSet`:
   Compare the current value to the value expected to be there, and if they match, update the value to the supplied value. If the update is successful, true is returned; otherwise, false is returned. These are also two different operations needing to be performed as a single operation. The "`bool __atomic_compare_exchange_n (type *ptr, type *expected, type desired, bool weak, int success_memorder, int failure_memorder)`" function is used to perform the two operations atomically. The function compares the expected value to the value currently stored in memory, updating memory to the new value if they are the same and returning true, otherwise simply returning false and not changing the value stored in memory.

`AtomicLong` and `AtomicInteger` extends these basic functions to include arithmetic functions. The arithmetic functions supported by the library are:

1. `addAndGet`:
   This method adds a value to the current value stored; after the value has been added, the final result is returned. This method also contains two separate operations that need to be performed as a single operation. The GCC built-in function "`type __atomic_add_fetch (type *ptr, type val, int memorder)`" is used in the C library to perform both functions atomically. The result of the function is then passed back to the Java class to return to the client.

2. `decrementAndGet, incrementAndGet`: These methods decrement/increment the value stored by the atomic variable and return the result of the operation to the client. The GCC functions used to perform these actions are "`__atomic_sub_fetch (type *ptr, type val, int memorder)`" and "`__atomic_add_fetch (type *ptr, type val, int memorder)`". These functions take in a value and a memory location, subtract/add the value to the value currently stored in the memory location and return the result. We know the value to subtract/add is always one; therefore, the constant one is always passed in as a parameter for the value.

3. `getAndAdd, getAndIncrement, getAndDecrement`:
   These methods add/subtract a value to the current value in memory and then return the original value. If we were to write code for this, we would need to store the current value in a temporary variable, then add/subtract the two values, store the value, and then return the temporarily stored variable. As these functions contains multiple individual statements, they need some form of synchronization in the form

Method	GCC Function
get	*type __atomic_load_n (type *ptr, int memorder)*
set	*void __atomic_store_n (type *ptr, type val, int memorder)*
getAndSet	*type __atomic_exchange_n (type *ptr, type val,int memorder)*
compareAndSet	*bool __atomic_compare_exchange_n (type *ptr, type *expected, type desired, bool weak, int success_memorder, int failure_memorder)*
addAndGet	*type __atomic_add_fetch (type *ptr, type val, int memorder)*
incrementAndGet	*__atomic_add_fetch (type *ptr, type val, int memorder))*
decrementAndGet	*__atomic_sub_fetch (type *ptr, type val, int memorder)*
getAndAdd	*type __atomic_fetch_add (type *ptr, type val, int memorder)*
getAndIncrement	*type __atomic_fetch_add (type *ptr, type val, int memorder)*
getAndDecrement	*type __atomic_fetch_sub (type *ptr, type val, int memorder)*
intValue	N/A
longValue	N/A
doubleValue	N/A
floatValue	N/A

Table 4.1: GCC Functions used for atomic methods

of a lock. As an alternative to using locks, we may again use the built-in atomic functions provided by GCC to perform all of these functions in a single atomic statement. The functions used to perform these actions are "`type __atomic_fetch_add (type *ptr, type val, int memorder)`" and "`type __atomic_fetch_sub (type *ptr, type val, int memorder)`". Again, for the increment and decrement functions, the value one is always passed in as the value parameter.

4. `intValue, longValue, doubleValue, floatValue`:
   These methods trivially convert the current type to the required type. These methods do not directly use the JNI framework. The `get` method is simply called by the Java code, and the results are converted to the required type by using the `valueOf` method provided by Java.

For the atomic classes, no lock has been used; only the atomic functions provided by GCC were used to ensure processes safety. Table 4.1 shows a Java method and the GCC function that is associated with each method. There are also no overflow exceptions thrown by these classes. This is by design to alleviate the exception checking from the library, thus reducing the execution time of the critical path.

Chapter 5

Synchronisation

When processes need to compete for shared resources or cooperate with each other when carrying out their activities, they need to synchronise their actions. Synchronisation refers to one of the two concepts: synchronisation of processes and data synchronisation [14].

Sharing variables across processes (Section 4.1) results in concurrent access to an address space; this may result in inconsistencies [56]. One of the classic examples of this situation is a race condition. Listing 5.1 shows code that may run into a race condition, if the for loop is executed concurrently by multiple execution entities. The shared variable x may not always reflect what the program is intended to do. If, for example, x is updated by two different processes simultaneously, one of the updates may not show. Data synchronisation tools help ensure that inconsistencies do not arise in shared address spaces [56]. Data synchronisation tools offered by the JCP library include locks (Section 5.1), read-write locks (Section 5.2), semaphores (Section 5.3) and condition variables (Section 5.4).

Processes may sometimes need to perform a task after a certain task has been completed. This may be difficult to ensure in a multiprocess system as process scheduling is managed by the operating system and is undetermined by the running processes [56]. In order to control the execution order of tasks in a non-deterministic scheduling system synchronisation tools are used. Synchronisation is used to block the processes from advancing prematurely and causing a bug. The tools provided to facilitate the management of task execution are latches (Section 5.5 and barriers (Section 5.6)).

```
1       for ( int i = 0; i < 10000000; i++ )
2       {
3           x = x + 1; // Shared Variable
```

```
4       }
```

Listing 5.1: For loop that may run into a race condition if run in parallel

5.1 Lock

The code shown in Listing 5.1 is not process safe. If line 3 (`x = x + 1;`) is run by multiple processes in parallel, some of the updates may not occur; such a program is race condition prone and contains a bug. Race conditions are avoided by using mutual exclusion. The JCP library will provide mutual exclusion functionality through a lock object. JCP will contain a lock that closely follows the `ReentrantLock` class that is provided by the `java.util.concurrent` library. The lock should be accessible to any Java process running on the system. The implementation of the lock will use the pthreads mutex provided by Linux (`pthread_mutex_t`). The mutex will allow JCP to provide mutual exclusion between processes if it is used in conjunction with the `PTHREAD_PROCESS_SHARED` attribute. As Java code is not able to use the pthreads library directly, a C library is needed, which will be used through the JNI.

JCP provides a lock class that may be shared by different Java processes running on the same system. The Lock object is built on the mutex provided by the pthreads library (`pthread_mutex_t`). In order for the different processes to be able to acquire the mutex, the mutex needs to be in a shared location. The IPC mechanisms provided by Linux will be used to allow for the mutex to be shared. In addition to the mutex being in a shared location, the mutex also needs to have the `PTHREAD_PROCESS_SHARED` set through the `pthread_mutexattr_setpshared` function. Additional attributes of the lock also need to be stored and retrieved when needed by the JCP library; these include:

- `isRecursive` (A boolean set if a single process may lock the lock multiple times making the lock Reentrant).
- `isLocked` (boolean to keep track if the lock is currently held by any process).
- `collectData` (This is set to true if the lock collects data about the owner, and waiting processes).
- `owner` (The process id of the process currently holding the Lock).
- `waitingProcesses` (Number of processes waiting to acquire the lock).

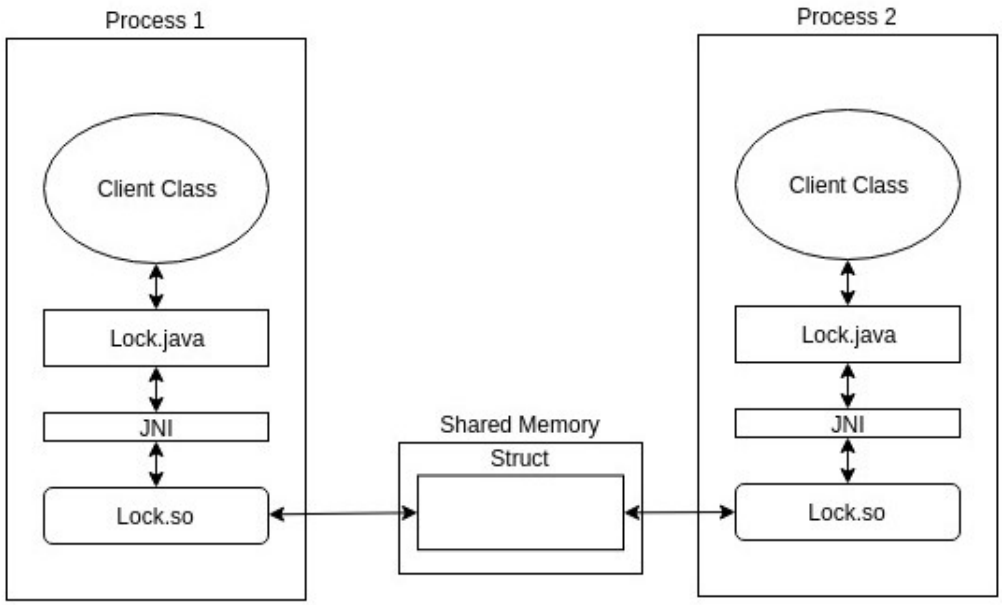


Figure 5.1: Representation of a lock

A struct (`LockMem`) is created to encapsulate the internal data of the lock. As the struct will be used concurrently by different processes, there needs to be a protection mechanism to the access and modification of the internal data structures. The GCC built-in functions for atomic memory access are used to modify the struct's internal data structures. For example, when increasing the number of processes waiting, the `__atomic_fetch_add` method is used.

Although multiple processes may use the lock on the system, only the process that created it will have access to it [56] as other processes do not have a reference to it. In order to solve this issue, shared memory is used. `Lockmem` is placed in shared memory, and the C library may then access the struct from shared memory. The Java library is then able to use the lock through the C library and JNI.

The shared memory is created using a unique long passed in by the client. The long is passed through to the `shmget` method, and `IPC_CREAT | IPC_EXCL` as a flag parameter. The shared memory may then be accessed using the unique integer returned by the method.

In addition to the `ReentrantLock`, JCP provides a `Mutex` class, which is not found in the Java concurrent library. The mutex provides similar functionality as the `Reentrantlock`. The `Reentrantlock` provides a locking mechanism with a high-level functionality; on the other hand, the mutex provides a locking mechanism's bare functions. The mutex

provides methods to lock, unlock, try-lock, and a method to test if the mutex is locked. With this limited functionality, the mutex will provide a simple, more efficient locking mechanism compared to that of the `ReentrantLock`. However, the `ReentrantLock` is still provided in the case where the client needs higher-level functionality than that provided by the mutex lock, and for compatibility with the Java concurrency library.

5.2 Read-Write Lock

In an application, you may have data that is frequently read by concurrent processes and seldom modified. In this situation, the mutual exclusion locks may prove to be inefficient. A mutual exclusion lock provides mutual exclusion to the readers, although the data is not modified. In this situation, the ideal lock should only block access to the data while it is being modified, and allow readers to read the data without blocking if the data is not being modified. The read-write lock allows multiple process readers to share lock access in order to allow multiple parallel data reads. On the other hand, a writer has exclusive access to the data, and no other process may access the data while it is getting modified [40]. This makes a read-write lock better suited for some applications compared to mutual exclusion locks; for this reason, a read-write lock has also been implemented in the JCP library.

The pthreads library contains a read-write lock (`pthread_rwlock_t`), which could be used to implement the read-write lock. However, because the read-write lock also needs to be able to be used with condition variables (in the same way as the `java.util.concurrent.locks.ReentrantReadWriteLock`), the pthread read-write lock will not work as pthreads condition variables (`pthread_cond_t`) may only be used with pthread mutex (`pthread_mutex_t`). The read-write lock is therefore implemented on top of the pthreads mutex.

Implementing the lock on top of the mutex means that the management of the read and write lock has to be handled by the library. The design of the read-write lock is most clearly shown in a state diagram (see Figure 5.2). The lock may exist in several states, which determine how the lock will behave when a request from a client process is received. The states and behavior of the lock are explained below.

Start

The *Start* state is the lock's resting state. In this state, the lock may be locked by either a reader or a writer. If a reader locks the lock, the lock's state changes

to *Reading* state. On the other hand, if a writer locks the lock, the lock goes into the *Writing* state. If both readers and writers are waiting to acquire the lock, the process to acquire the lock will be the process that acquires the `guardLock` first. The `guardLock` is the lock that is used to guard and protect the internal data structure of the read-write lock.

Reading

In this state, any reader that attempts to lock the lock is allowed to continue without experiencing any blocking. The lock stays in the same state if a reader locks the lock. When all the readers have released the lock, the state of the lock moves back to the *Start* state, and a signal is sent on the `readsComplete` condition variable. On the other hand, if a writer attempts to lock the lock, the lock's state moves to the *Waiting For Reads To Complete* state, and the writer waits on a `readsComplete` condition variable.

Waiting For Reads To Complete

The lock is locked by readers in this state, but you have a writer(s) waiting for the lock. If the lock is in this state, no additional readers are allowed to acquire the lock. Additional readers attempting to acquire the lock wait on a `writesComplete` condition variable. When all the readers have released the lock, a writer acquires the lock, and the state of the lock moves to the *Writing* state.

Writing

In this state, a single writer is holding the lock. If an additional writer attempts to acquire the lock, the new writer is suspended and waits on a condition variable (`writesComplete`). Readers trying to acquire the lock are also put on the same condition variable as that of the writers. When the writer releases the lock, the lock goes into the *Start* state.

As with the lock discussed in Section 5.1, the mutex is the first attribute in the underlying struct used (this is also a requirement for locks used with conditions discussed in Section 5.4). As previously mentioned, the struct contains a `guardLock` lock to protect the struct's internal data structure. The `guardLock` is also used with the `writesComplete` and `readsComplete` condition variables.

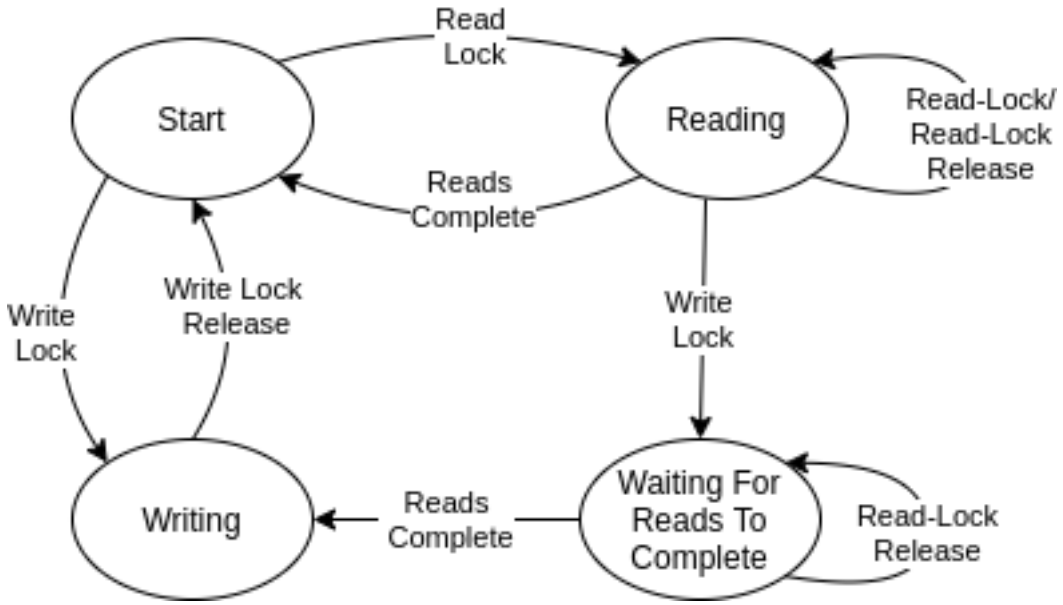


Figure 5.2: Read-Write Lock Diagram

5.3 Semaphore

Sometimes it is permissible to allow up to a certain number of concurrent accesses to a critical section. In this situation, locks are too restrictive; they only allow a single process into a critical section. Another synchronisation method is required to solve this problem. Counting semaphores solve this and may be used in this situation. Semaphores are used to control the number of processes that can access a certain resource or perform a given action at the same time [19]. The design and implementation of the JCP `Semaphore` class is based on the semaphores provided by the Java multithreaded library (`java.util.concurrent.Semaphore`. The actions that are permitted by the semaphore are:

1. `availablePermits`: returns the number of available permits
2. `acquire`: attempts to acquire a permit from the semaphore, blocking if there are no permits available. There is an overload-method accepting the number of permits to acquire.
3. `drainPermits`: acquires all permits and returns the number of permits it was able to receive.

4. `tryAcquire`: attempts to obtain permits returning false if there are no more permits currently available to acquire. An overload-method takes in the number of permits to attempt to acquire. If all the permits are acquired, the method returns true; otherwise, no permit is acquired, and false is returned.

5. `release`: releases a permit. If multiple permits need to be released, the number of permits may be supplied.

Named semaphores are used in the implementation of semaphores. As named semaphores are natively process-shared, using named semaphores is simple. When needing to perform an action on the semaphore, JCP library calls the semaphore's appropriate method. Table 5.3 displays the Java method and the named semaphore counterpart used to perform the action.

As the named semaphore does not account for multiple permit acquisition and releases, the methods need to be modified to account for methods that attempt to acquire/release multiple permits in one call. For most methods, this is simple as the acquisition/release is simply performed in a loop inside the method. However, the try acquire method has a timeout; this is not simple as all the permits need to be acquired or none. In a loop, all the permits are attempted to be acquired before the timer times-out. If the timer times-out while attempting to acquire the permits, if any permits have already been acquired they are released, and false is returned. Otherwise, if all the permits have been successfully acquired true is returned.

Method	**Named Semaphore Function**
Semaphore(create)	sem_open(const char *name, int oflag, mode_t mode, unsigned int value)
Semaphore(open existing)	sem_open(const char *name, int oflag)
availablePermits	sem_getvalue(sem_t *sem, int *sval)
acquire	sem_wait(sem_t *sem)
tryAcquire	sem_trywait(sem_t *sem)
tryAcquire (timed)	sem_timedwait(sem_t *sem)
release	sem_post(sem_t *sem)

5.4 Condition Variables

Condition variables are synchronisation primitives that enable a process or thread to wait until a desired condition is met [2]. Using condition variables is advantageous because the waiting process goes to sleep and gives up the CPU to other processes to utilize while it waits. The process may then be woken up when the condition is met through a signal. Using condition variables allows the CPU to be utilized by other processes while the process cannot advance in execution. JCP provides condition variables through the `Condition` class.

The `Condition` class is implemented on top of the pthread condition variable (`pthread_cond_t`) . The condition variable needs to be linked to a lock when it is created; this is because when you call the `wait` method on the pthread condition variable, you need to pass it a pthread mutex. A condition is then associated with a pthread mutex at initialisation, and waiting on the condition requires the process to have the mutex locked.

As the struct of the lock is in shared memory (see Section 5.1), only the shared memory id is needed to get access to the lock. The shared memory obtained is cast to a pthread mutex as the mutex should always be the first element. The Java `Condition` class constructor accepts a Lock interface (`ILock`). The Lock interface contains a `getLockShmID` method used to get the lock's shared memory id. Using the `getLockShmID` method of the (`ILock`) interface the `Condition` class constructor may then obtain the lock's shared memory id. The lock's shared memory id is then passed on to the C library to create the condition variable and link it to the relevant lock.

As the condition variable and lock are built on top of the pthread condition variable and pthread mutex respectively, the condition variable will also behave the same way as the pthread condition variable. This means that when the `wait` method is called, the associated lock will be released; if the calling process does not have the lock locked, then the resulting behavior is undefined [51].

The condition variable supports both unblocking a single waiting thread or all threads waiting on the variable through the `signal` and `signalAll` methods. The unblocked threads will contend for the lock as it had been released when calling the `wait` method. Only after a process has acquired the lock will the process be unblocked. To implement these methods the `pthread_cond_signal` and `pthread_cond_broadcast` methods are used to send a signal to the blocked processes.

The JCP `Condition` also supports timed waits. There are two types of timed waits available to use with the condition variable. The first wait method (`awaitUntil`) waits until a supplied time has passed. Alternatively, the second type of wait method (`await`, and `awaitNanos`) are passed in the amount of time to wait, and will wait until that amount of time has elapsed. The return of the methods is a boolean value representing the success status. If the condition timed out before the signal was received, false is returned; otherwise, true is returned. Both waits call the `pthread_cond_timedwait` method. The method accepts a `timespec` struct, representing the amount of time to wait as an argument. The `timespec` struct represents time in seconds and nanoseconds through the `tv_sec` and `tv_nsec` attributes.

Having to represent time in seconds and nanoseconds means that the amount of time to wait needs to be represented in seconds and nanoseconds before being passed on to the native library. Converting to seconds and nanoseconds is not an issue as a nanosecond is the smallest unit of time; therefore, no rounding off issues will be experienced. In `awaitUntil` the time to wait for is first determined and then converted to seconds and nanoseconds. If the value is too large to be converted to seconds and nanoseconds, an `OverflowException` is thrown.

The `awaitNanos` method works the same way as `await`, except instead of only returning whether the condition wait was a success, the method returns the amount of time left before the wait would have timed out. A positive value indicates that the wait did not timeout and was rather signaled, while a negative one value indicates a timeout. A timer is used to calculate the amount of time spent while waiting. The value returned by the `pthread_cond_timedwait` method is used to determine if the wait timed out or not. If the wait timed out, a negative one value is returned. Otherwise, the waiting time is subtracted from the maximum waiting time and returned.

5.5 Latch

Latches prevent processes from progressing until dependent tasks have been completed. This is achieved by allowing the latch to have two states (*open* and *closed*). The latch starts in the *closed* state and proceeds to the *open* state. The latch keeps a counter that is assigned during initialization (an integer value greater than zero). The counter is decremented by processes calling the `countDown` method on the latch, until it reaches zero. A process calling the `wait` method blocks until the counter reaches zero and the

lock's state changes to *open*. At this point, the latch is said to be in an *open* state, and all waiting processes are unblocked and free to continue. When the lock has been opened, it may never be closed again. If a process calls `wait` on the latch after it has opened, the process continues normally and experiences no delay.

In order for the latch to be accessible to multiple processes, the latch needs to be stored in shared memory. Processes may then access the latch through the JNI. The latch struct needs to store a counter (`count`). As the struct will be accessed by multiple processes concurrently, to protect the latch's internal data the struct also needs to store a pthread mutex.

When a process calls the `wait` method, the state of the latch is first checked. Checking the status of the latch involves testing if `count` is equal to zero. If it is zero, the process continues and experiences no blocking as the latch is in an *open* state. Alternatively, the latch is in a *closed* state, and the process should block until the latch's state changes to *open*.

In order to block the process until the latch is in an *open* state, a pthread condition variable is used. The process waits on the condition variable until a signal is received. The mutex that is required to be used by the condition variable will be the mutex used to protect the struct's data. This will not be a problem as the mutex will be locked at this stage, therefore conforming to the pthread condition usage standards.

There is an overload to the `wait` method. If a wait takes too long the process could timeout of the wait by providing the maximum duration of time it wishes to wait for. If the timeout lapses before the latch's state changes to *open*, the process will unblock and return. This method uses the `pthread_cond_timedwait` method to wait for the latch to be opened. The conversion of the time to the required `timespec` struct by the `pthread_cond_timedwait` method is identical to that for the `Condition` class, discussed in Section 5.4.

The class's two other methods include `getCount` and `countDown`, with the former returning counter's value, and the latter decrementing the value of counter until the latch's counter reaches zero. At this point, the latch is in an *open* state. When the latch gets the *open* state, processes waiting on the latch are woken up using the `pthread_cond_broadcast` method.

5.6 Barrier

Barriers provide similar functionality to a latch, except as opposed to waiting for a latch to be opened, barriers wait for a number of concurrent processes to call `wait` on it, and trip the barrier. After the threshold (parties) of the number of waits called is reached, the barrier is said to be *tripped*. A process calling `wait` is blocked until the barrier is tripped. After the barrier is tripped, all waiting processes are unblocked. The remaining parties count is then restored to the number of parties. Subsequent processes calling `wait` will be blocked until the barrier is tripped again.

The barrier uses an all-or-none breakage model for failed synchronisation attempts. This means that all blocked processes need to exit the barrier together no matter the reason for exiting. If a process leaves a barrier point prematurely because of a timeout, the processes, along with all other processes waiting at that barrier point will also leave abnormally through a `BrokenBarrierException`.

A `BrokenBarrierException` may be thrown for various reasons. The first reason is if the `wait` method is called while the barrier is in a *broken* state. In this scenario, the process will not block at all; it will simply throw a `BrokenBarrierException`. Alternatively, suppose another process times out while the current process is blocked, waiting for the barrier to open. In that case all processes will also be unblocked, with a `BrokenBarrier-Exception` being thrown. The exception is also thrown if the barrier is reset while a process is waiting on the barrier.

The barrier is implemented in shared memory. The fields of the struct of the barrier (`CyclicBarrierMem`) are:

1. `lock`: This is a pthread lock to protect the internal data structure of the barrier from multiple process access.
2. `parties`: the total number of parties the barrier has. This is the value set when the barrier is initialised.
3. `remainingParties`: The number of remaining parties before the barrier is tripped.
4. `isBroken`: A boolean value to keep track of whether the barrier is broken or not.
5. `prematureWakeUp`: A boolean to identify if the barrier is reset. This value is set to true if the barrier was reset, and false if not.

6. `condition`: The pthread condition variable used to wake up processes that are waiting for the barrier to trip.

Figure 5.3 shows a visual representation of how the `wait` and `reset` methods are implemented by the barrier. The implementation of other methods is trivial as they simply acquire the lock, retrieve the field they are inquiring about, unlock the lock, and return the result.

When the `wait` method is called, first the barrier is checked if it is broken or not using the `isBroken` boolean. If the barrier is broken, a `BrokenBarrierException` is thrown, and the method returns immediately. Otherwise, the number of remaining parties (`remainingParties`) is decremented. The number of remaining parties is checked if it is equal to zero or not. If the remaining parties are equal to zero, it means there are no more parties remaining. The barrier is then tripped, and all the processes are released. Otherwise, if there are still parties remaining, the process goes into a wait.

There are two types of wait the process may go into depending on the type of method that was called. If the `wait` method is called with no timeout supplied, the process goes into a conditional wait by calling the pthread `pthread_cond_wait()` method using the condition variable `condition`. When the process wakes up, it checks if the wake up is a normal wake up or did the process wake up prematurely. If it is a normal wake up, that means that all parties have reached the barrier, and the barrier is tripped. The process will then return and continue with its execution.

If the wait is a timed wait, the process goes into a conditional timed wait. When the process finishes waiting, it is tested to see if the wait timed out or not. If the wait did not timeout, then the `prematureWakeUp` boolean is tested, and the appropriate action is taken in the same way as a wait without a timer. Alternatively, the process may have timed out during the wait. In this case, the `isBroken` boolean is set to true, and other waiting processes are also released. A `BrokenBarrierException` is then thrown, and the method returns.

When resetting the barrier the `prematureWakeUp` boolean is set to true. Any processes waiting on the barrier are then woken up, and the barrier is reset. `prematureWakeUp` will stay true after this even after the barrier reset has been concluded and all the processes have been released. In order to fix this issue, the `prematureWakeUp` variable is set to false every time a process waits on the barrier. This works because at this point it is not possible for the barrier to have been reset, it may only get reset again after the process waits and releases the lock.

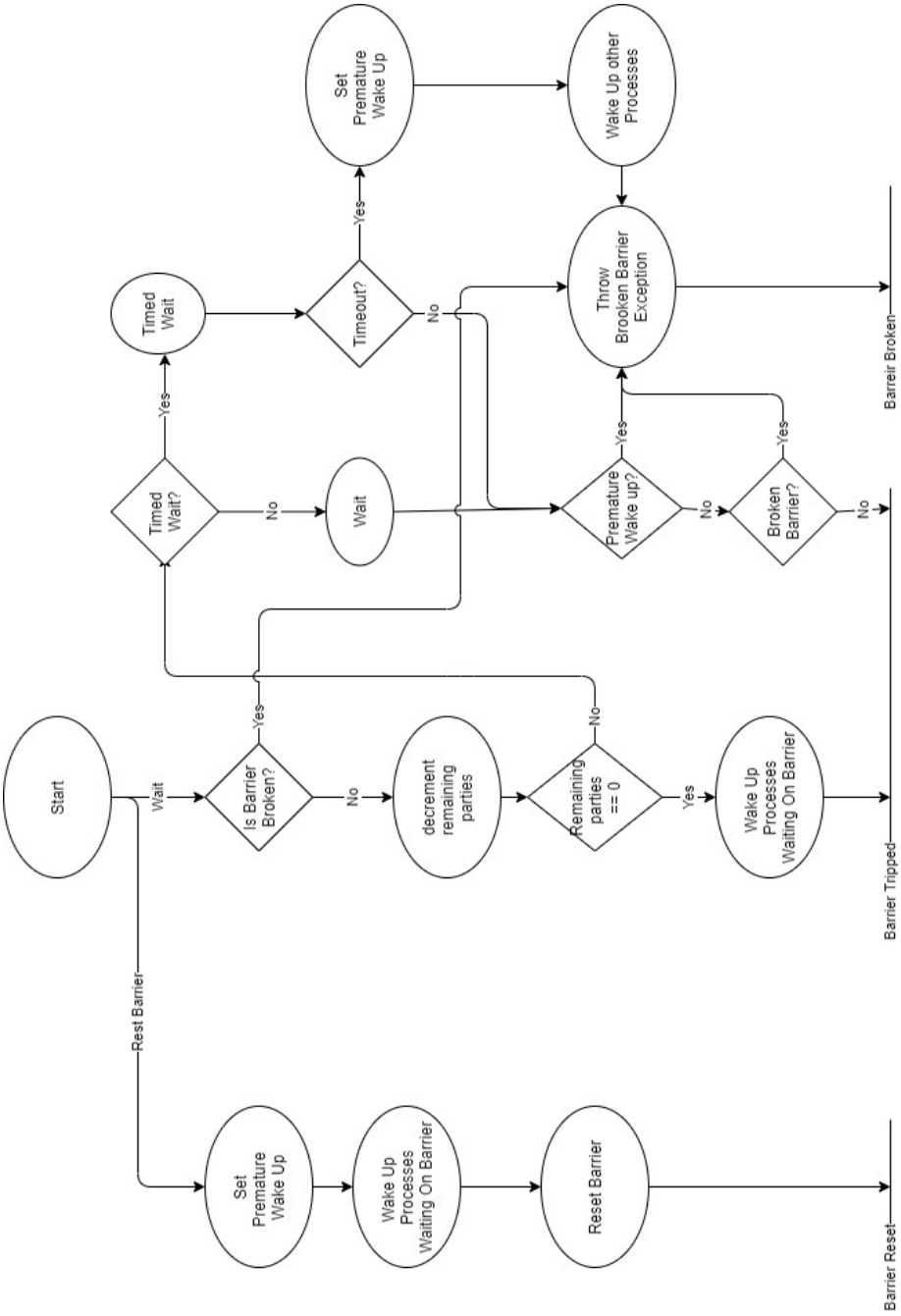


Figure 5.3: Barrier diagram

Chapter 6

Data Structures

In this chapter general data structures to be usable by concurrent process executing on the system are designed and implemented. The general data structures implemented by the JCP library are `Array`, `List`, `ConcurrentHashMap`, and `BlockingQueue`. The `ConcurrentHashMap`, and `BlockingQueue` data structures resemble the `ConcurrentMap`, and `BlockingQueue` classes of the concurrent collections available in the Java thread library.

6.1 Data Structure Implementation

All the data structures follow a similar implementation pattern. The general implementation of the data structures is discussed first in this section. More detailed implementation of each data structure is discussed in the sections that follow.

Unlike the synchronization tools discussed in Chapter 5, data structures are not implemented in shared memory. The reason for this is because with shared memory you have to know the required size of the data structure prior to creating the shared memory. This is not always possible with data structures, especially with a list data structure. This is also not possible with String arrays as you do not know how much space each String will take up, unless you limit the size of each individual String. This has the drawback that you may end up wasting space if the limit is too large or not have enough space if the limit is too small.

Instead of using shared memory an independent process is used to implement and manage the data structure, from here on called the management process. The management process

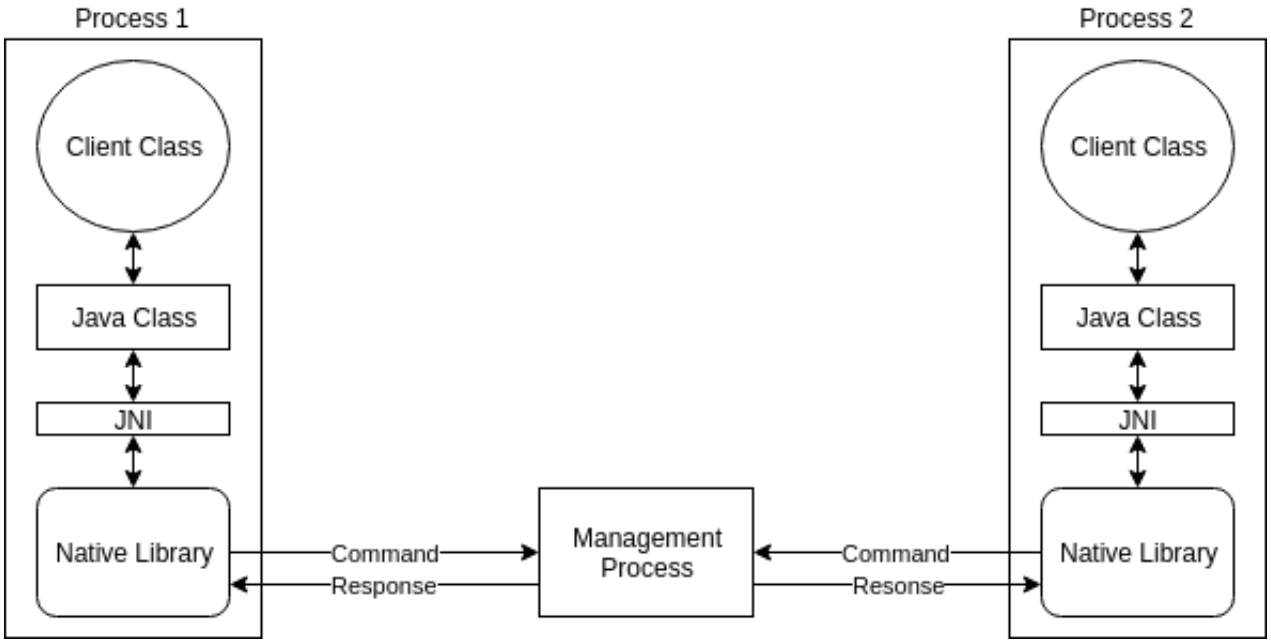


Figure 6.1: Management Process Interacting with Two Java processes

is started when the client creates the data structure. The management process accepts commands from the client processes and modifies the data in the data structure internally and sends the relevant results back to the client process. When the client is done with the data structure the management process is terminated as well. Figure 6.1 shows and illustrates how two Java client processes could connect and interact with the management process.

When the client creates the data structure, the management process is created using the `fork` system call and made to execute the code of the management process through the `execvp` system call. The arguments supplied by the client are passed on to the management process, as command line arguments, to set up the required data structure accordingly.

The client process and the management process communicate through a message queue. The client process will initiate the communication by placing a message on the message queue through a C library, using JNI. The management process will then read the message. The message will contain the command to execute as well as the element that needs to be processed, if any exists.

After the management process has performed the command, it will send a confirmation message (response) back to the client process. The response will contain the success state of the command performed as well as the returned value, if any exists. In order to avoid

the management process rereading the response message itself a second message queue (response message queue) is used to send the response.

It has to be ensured that messages are not mixed up between the multiple clients that may be interacting with the management process at the same time. This in particular is a concern for the responses as the client process needs to read messages pertaining to it and not other client processes' messages. Message queues allow for selection of a message to read from the queue based on the type field. When the management process reads in a message, it takes note of the value of the type field. The response is sent back using the same value, the client will be waiting for a message with the type it sent to the management process. The JCP library ensures that the type value is unique for each of the client processes by setting the value to the client's process id which is unique for each process on the system.

A struct is created to hold the message data. The client application will place the data in the struct and place the struct in the message queue. The management process will then read in the struct from the message queue and process the message by performing the action specified by the message. When the management process is done processing the message, it will create a new struct containing the results of the operation. The result struct is placed in the response message queue for the client application to read. The client application then reads in the struct from the queue and retrieves the data needed.

As previously explained the reason for using two message queues, as opposed to a single message queue, is so that they are able to read in the messages without getting mixed up. This would particularly occur in situations where the client process might put a message in the queue and then read the message in from the queue again without giving the management process the chance to read the message.

6.2 Array

The data types that are natively implemented in arrays are int, char, String, double and long. Each of these types has its own array implementation. This means that each type will have its own Java array class, its own native library as well as its own management process implementation. On top of these types a generic array is implemented to support all other types. As with the other types of data structures there is an interface for array implementations to follow. The interface specifies the methods the arrays ought to support which include:

1. `contains`
2. `get`
3. `set`
4. `indexOf`
5. `lastIndexOf`

It is not possible to implement an interface using primitive data types, the standard Java wrapper classes (e.g. Integer) have to be used instead. This will affect the usage of the classes as the client will now need to create wrapper objects to hold the primitive data types when using the arrays.

The implementation of the different data types is similar in nature, however splitting them as opposed to creating a single generic data array has performance benefits. Performance is improved because the size of the struct is then calculated and known at compile time as opposed to run time. Using a single generic array would also have the draw back that elements would need to be serialised, and de-serialised, as we will see in the discussion of the generic array. The char array is discussed in detail below. Long, double and int arrays are not discussed as they are almost identical to the char array. The String array is discussed as it differs from the other types. The generic array is discussed last as it builds on the String array.

6.2.1 Char Array

To communicate with the management process the array uses the same model as the one shown in Figure 6.1. All the methods use the standard method of execution described previously in Section 6.1. When the array is created two keys (long values) are passed in to the class. These keys are used to create the queue. The queue is created with flags `0666 | IPC_CREAT | IPC_EXCL` . This will make the queue public to be used by any process in the system, however if the queue already exists in the system then a `SystemException` is thrown indicating the queue already exists.

The struct used to facilitate communication between the client process and management process has four fields:

- `long type`
- `int mesg_details`
- `int index`
- `char element`

The `type` field is used to identify the messages as previously described. `message_details` is used to identify the method that should be executed. The `index` field is used to tell which element on the array to work with. Lastly, the `element` field is used to send an element value to work with. The `message_details` field on the response message is used to describe the success of the command. If on the response message, the value stored in the `message_details` field represents an exception, the relevant exception is thrown on the client side.

When the array is initialised the management process dynamically creates an array class using the arguments passed on to it. The class contains an internal array to store the elements, the length of the array, as well as the methods offered by the array class.

The implementation of the methods described by the array interface is discussed below:

`contains`
: This method goes through each element of the array until the element supplied matches one of the elements already stored in the array. If a match is found while going through the elements of the array then true is returned and the `message_details` field in the response message is set to zero. Otherwise, false is returned and `message_details` is set to negative one. The client process then uses JNI, and the native library to read in the response message and checks the value stored in `message_details`. If the value is zero, true is returned, otherwise false is returned.

`get` **and** `set`
: These methods accept an index, a boolean pointer to return the success of the method, and an element. The supplied index is first checked to determine if it is a valid index. If the supplied index is out of bounds the supplied boolean pointer argument is set to false and the method returns `null`. Otherwise the element is

trivially retrieved from or set in the relevant position corresponding to the index. The response struct is then prepared. For the `get` method the retrieved element is placed in the struct, and the struct is then placed in the message queue. Upon receiving the message the client process first checks the `message_details` field of the message. If the value is a negative one value an `IndexOutOfBoundsException` is thrown, otherwise the value stored in the `element` field is returned if the action was a `get` method.

`indexOf` **and** `lastIndexOf`

The `indexOf` method loops through the array searching for the first occurrence of the element and returning immediately when found. The `lastIndexOf` method does the same as the `indexOf` method except it starts from the back of the array, working its way back to the front of the array. The index the element is found at is then placed in the `index` field of the response struct for the client to retrieve.

`destroy`

The `destroy` method removes the arrays from the system. Removing the arrays from the system includes placing a message in the message queue with the `message_details` set to `DESTROY` (negative one). When the management process receives the destroy message from the client the management process cleans up, sends a response to the client and terminates. The cleanup of the management process involves freeing up the memory of the array data structure. After the client has received the response it removes the queues from the system.

6.2.2 String Array

Strings provide some additional complications over the other supported data types. These problems include the following:

1. Unknown String length
2. Strings are reference types as opposed to value types.

Unknown String Length

The data types that have been discussed have all had a fixed data size. Strings on the other hand do not have a fixed data size due to the varying length of a string. The varying

length of a string becomes an issue when sending data to and from the management process. This is an issue because both the sending and receiving processes need to know how much data will be sent through beforehand. If you have different sized data, how large should the message struct be?. The simplest solution to this problem is to set a limit to the size of the string a user may store. However with this solution there is a set of problems that are not supported, problems with strings larger than the limit that has been set. Problems that do not have large strings also waste space as the rest of the allocated space will still be sent even though it will not be used.

Another solution is to first send a message specifying the length of the string (metadata message) and then follow with the a message containing the string. This solution will use three queues. The first queue is used to send a message to the management process (metadata). The second queue is used to send a response messages from the management process to the client (metadata), and the third queue is used to send strings between the management process and client process (raw data). Two structs are used to facilitate this, one to hold the raw data and one to hold the metadata.

Before the raw data is sent a metadata message is first sent. The metadata message will contain data about the raw data to be sent, in particular the length of the string to be sent in the raw data message, if any exists. After reading in the metadata message the management process will then know how much data to expect from the raw data message queue. The client will send the message containing the raw data directly after it has sent the metadata message. The management process may then read in the correct raw data message using the `type` field received from the metadata message and process the message accordingly.

After the message has been processed, a response needs to be sent back to the client application with the results of the operation. Sending a response to the client entails first sending a metadata message to the client, followed, if need be, by a message containing the data of the result.

Strings are reference types

The second issue with strings is that they are reference types. This means that when storing a string element in the management process space needs to be allocated and deallocated when done with the element. The space is created on the heap after the metadata message is read in using the `malloc` function. The metadata message contains

a field to specify how long the string is, i.e. how much space is needed to store the string. The pointer to the string is then passed around and stored in the array if need be.

The string is deallocated when it is no longer needed. For the `contains`, `indexOf`, and `lastIndexOf` methods the string parameter is no longer needed when the method exits. The string is then deallocated when the method exits. However for the `set` method the string is only deallocated when another string is getting stored in the index. The index is checked if it contains any string, if it does the existing string is first deallocated and the new string is stored in that place.

Multi-threaded Or Single Threaded Strings

Three message queues will typically be used by a single request in a string array. Reading in and sending three messages per request may prove to be communication-intensive and become a bottleneck for an application. The effects of this problem could possibly be reduced by making the management process a multi-threaded application. The number of threads the process would use would be defined by the client application during initialisation of the array.

A multi-threaded management process, however poses a problem. Access to the array would now need to be monitored and guarded against race conditions. There are two possible methods to manage access to the array. The first method is to lock the entire array, and monitor any access to it. The second solution is to lock the individual elements of the array.

Locking individual elements would allow multiple threads to access the array concurrently as long they are accessing different indexes in the array. However the `contains`, `indexOf` and `lastIndexOf` methods would prove to have a significant slow down as they would have to lock each and every element they check. When a thread is executing the `contains`, `indexOf` and `lastIndexOf` method, the whole data structure would also need to be prevented from modification during that time. In order to solve this issue, a second lock would need to be present, going back to locking the entire array. Therefore locking individual elements would only effectively parallelise the `get` and `set` methods.

Locking the entire array would be a simple solution to the problem. Locking the entire data structure would only allow a single thread to access the array even if they are modifying different indexes. This would have a negative impact on concurrency when multiple threads are fighting for the single lock [19] while they will be modifying and

working on different sections of the data structure. This means that this data structure may experience lock contention.

In order to alleviate lock contention the pthread read-write lock may be used instead of a mutex. This would mean the inefficiency of locking the entire data structure would be experienced only in the `set` method. As the `set` method accesses the affected element directly locking the entire array using a read-write lock would not have a significant impact on performance.

Another issue with a multi threading application is with methods that return a string, for example the `get` method. The issue is that while the thread is preparing to send the data to the client another thread may acquire the lock and modify the data, therefore creating a race condition. In order to fix this issue the lock may only be released after the result has been sent to the client process. This solution will result in the reading in of the data being the only parallelised section of the code.

Considering the fact that only the reading in of the data would be parallelised and the overhead associated with multi-threading and using a lock to ensure correction, the use of a multi-threaded application would most likely prove to be inefficient. Therefore the implementation of a multi-threaded application has been disregarded.

6.2.3 Generic array

In order to make the library is as versatile as possible, a generic array is needed. The array ought to work in the same manner as the Java array, supporting all types. The type (called type `T`) ought to be specified at initialisation and only elements of that type may be stored from then onwards.

The generic array is built on top of the previously discussed string array. A string array is created and stored internally in the generic array. The generic array uses the string array to store the elements.

Objects needing to be stored are serialized to a string object and stored in the string array. The object is then de-serialized back to type `T` in Java when it is being retrieved from the array. De-serialization and serialization of objects throws multiple exceptions (`ClassNotFoundException`, `IOException` and `ClassCastException`). These exceptions are not handled by the array class, rather the exceptions are passed up to be handled by the client code.

Serializing and de-serializing of the objects requires the generic array to constrain acceptable classes to only serializable classes. This restriction is enforced in the Java generic class with `T` required to extend the `java.io.Serializable` class.

De-serializing the string returns an `Object` type rather than type `T`. The object has to then be cast to type `T`. The object is then cast using the class of type `T` through the `Class.cast` method. In order to enable this the class of type `T` is passed in at initialization and stored internally in the class.

6.3 List

Although lists are often slower than arrays, they are better suited for some problems. A prime example of such a problem is where the size of the data structure is not known in advance. For this reason it is decided that the library in addition to having arrays ought to also have a list data structure. The implementation of the list is similar to that of the array, supporting the same data types. The nature of the message passing is identical, as well as the protocol that is followed in the message passing. The difference between the lists and the array comes in how the data structure is implemented in the management process. Therefore only the implementation of the list in the management process will be discussed.

The list is implemented through a doubly-linked list. Each element is represented with a node. The nodes are linked with each other through pointers from the first element to the last. In addition to storing the data, each node keeps a pointer to the element before it and the element after it. The system then keeps pointers to the first element (head) and the last element (tail). The system also keeps count of the number of elements in the array. The head and tail node pointers are initially made to point to null, and the length is set to zero. Having head and tail point to null represents the list being empty.

There are five basic methods supported by the list: `get`, `set`, `add`, `remove`, and `indexOf`. The rest of the methods are variations of these methods. The basic methods are the methods that will be discussed here.

`get` and `set`

The `get` and `set` methods are similar in their design and implementation therefore they are discussed together. `get` and `set` are supplied indexes and return the value

currently in that index, also setting the value at the index to the new value, if the method is a `set` operation. Just like arrays with `get` and `set` methods, we first check if the index supplied is in the bounds of the list. The list is then traversed using a for-loop, starting from the head of the list and moving on to the next element with every iteration. A counter to keep track of the number of elements that have been traversed is used. When the counter and the supplied index are equal the desired node has been found. If it is a `set` operation the modification to the data is made and the old value is returned. Otherwise the value is simply returned.

`add`

The `add` method inserts elements into the list. The position to insert is supplied through an index argument. The position to insert is found the same way as the `get` and `set` methods, using a for loop. After the position to insert the element is found, a new node is created to store and hold the value, and is then linked into the correct position in the list [71].

`remove`

The `remove` method accepts an index and removes the element at that index from the list. The first step is the standard check to see if the index is not out of bounds. The victim (node to remove) is then found by looping through the elements and stopping at the right index. The victim is then removed. There are three possible scenarios that need to be accounted for when removing the victim, removing from the front of the list, removing from the middle of the list, and removing the last node of the list. Each of these cases is handled using the standard linked-list techniques [71].

`indexOf`

The `indexOf` method accepts a value and if the value is found it returns the index the corresponding element is found at, otherwise returning negative one. The method trivially loops through the nodes from the head to the tail, comparing the nodes' data to the given value. If there is a match the counter is returned. If the for loop ends without returning a position, negative one is returned indicating that the element getting searched for is not found in the list.

6.4 Concurrent Hash-Map

`ConcurrentHashMap` is a {`key`, `value`} data structure based on the `ConurrentMap` class offered by the Java concurrent library. A `value` is accessed through the `key` associated with the `value` as opposed to an integer index like in the array and list data structures. Both the `key` type and `value` type of a `ConcurrentHashMap` may be of any type. This makes the possible type combinations that may be used in a `ConcurrentHashMap` endless. There are no natively supported data types for the JCP `ConcurrentHashMap`, due to the fact that there are so many type combinations that may be used to create the `ConcurrentHashMap`. The only `ConcurrentHashMap` that has been provided in the JCP library is a generic `ConcurrentHashMap`.

The implementation of the `ConcurrentHashMap` uses the same model as the rest of the data structures, shown in Figure 6.1. The values and keys are first serialised, in the same manner as the generic array, and then they are sent over to the management process. The sending of the values is similar to that of the string array and string list previously discussed. Having two values to send over to the management process means that the message passing protocol needs to be modified.

The first modification is in the message buffer struct. Since two data values are possibly sent over from the client to the management process, two length fields are needed. When sending the data from the client to the management process, the client sends over the method specifier as well as the lengths of the two strings (`key` and `value`).

The client then sends over the `key` data using the raw data struct over the raw data message queue. The management process will read in the `key` from the raw data message queue using the `type` field, and `length` value received in the metadata message. Depending on the method that is getting processed the client will also send over the `value` to the management process in the same manner as the `key`. For example if the method is a `put` method the `value` will be sent, however if the method is a `remove` method there will be no `value` sent as there is no value to send over. The management process will decide on whether it should read in a `value` or not based on the length sent over in the metadata message. This check is performed by the management process before attempting to read in the `key`, and `value` from the message queue.

The management process at this point will have all the data that it needs to process the message and send the result back to the client. The client on the other hand will be currently waiting for the result of the operation in the response message queue.

After the management process has processed the message and prepared the results it sends a metadata message to the client containing the result of the operation, whether the operation was successful, and the length of the result data the client ought to expect, if any exists.

If there is a result that is needed to be sent over from the management process to the client process the data is sent through the raw data message queue. The client then reads in the data using its process id and the length of the data obtained in the metadata message.

Processing the message involves decoding the method that needs to be performed and performing it on the data structure with the `key` and `value` provided by the client and returning the result to the client process. The data structure is implemented through the use of a hash table.

Two components make up the hash table, a bucket array and a hash function. A bucket array is an array of size *N*. Each cell in the array is a bucket. The bucket holds the `key`, `value` pair. The hash function is used to identify the bucket to store the `value`. This is done by hashing the `key` and modding it by the size of the array to get the index. Elements may share the same hash value, this is called a collision. If there is a collision two different elements get mapped to the same bucket. In addition, as the size of the array is limited, two different hash values may still get matched to the same bucket.

In order to handle collisions instead of storing the `key`, `value` pair in the bucket, the bucket is implemented as a linked list. The elements are stored in a list using nodes that keep the key and the value of each element. The linked list is implemented the same way as the one previously discussed in Section 6.3. This will mean that when searching for an element we first need to find the bucket it is stored in by hashing and modding the `key`. After the bucket has been found the element is searched for in the linked list by comparing the keys of the nodes.

6.5 Blocking Queue

A blocking queue is a queue that blocks a process when it attempts to retrieve an element from it while the queue is empty, or if a process attempts to inserts into the queue while the queue is full [46]. A blocked process waits for another process to perform an action that should unblock the process. An example of such a situation is having one process

attempting to retrieve data from an empty queue. Rather than throwing an exception, the process will block until another process has put data in the queue for it to read. This data structure is particularly useful for producer-consumer problems.

In the same way as the array and list data structures the primitive data types are natively supported while other types are supported through a generic queue. The JCP library implements the blocking queues in the `za.co.jcp.datastructures.blockingqueue` package. Each supported data type has its own implementation classes. These classes are equivalent to the Java concurrent library's "`ArrayBlockingQueue`", and "`LinkedBlockingQueue`" classes

There are two different queue implementations the client may opt to use: an array based queue or linked list based queue, with the array based queue being the default queue. Each queue implementation has its own management processes. The JCP library starts up the relevant management process based on the type of queue the client has opted to use. After the management process has been started up the client is oblivious to the management process that is in use. The client simply inserts the message in the message queue and waits for the relevant response from the management process. Like most of the data structures the `BlockingQueue` follows the model depicted in Figure 6.1.

The operations supported by the `BlockingQueue` classes are inserts, removes, and examines. These operations are supported through various methods, with each method having its own characteristics.

Inserts

The insert methods insert an element into the back of the queue. The first method that may be used to insert into the queue is the `add` method. The `add` method attempts to add an element to the queue. If the element may not be added, primarily because the queue is full, the method will return without blocking, and throw an `IllegalStateException`. `offer` also attempts to insert into the queue without blocking, however with `offer` if the element cannot be added, the method returns false without throwing an exception. There is an overload of `offer`, with the overload a timeout for the amount of time to wait for the queue to have space to be available is supplied. If the timeout elapses before space is available on the queue, false is returned. `put` is the last insert method supported. The `put` method inserts into the queue. `put` will block the process if there is no space in the queue, therefore not returning without completing the operation.

Removes

The remove methods remove the element at the front of the queue. The `remove` method will throw a `NoSuchElementException` if the queue is empty. The `poll` method will return null if there are no elements in the queue. Returning null works because it is illegal to insert null into the queue, therefore returning null signifies that no element has been found in the queue. The `poll` method has an overload method accepting an amount of time to wait for rather than returning immediately. The `take` method blocks if there are no elements in the queue. `poll` is only permitted for Strings and generic data types as C does not permit value types to have a null value. The other types of queues throw an `UnsupportedOperationException` when attempting to call the `poll` method.

Examine

The examine methods look at the element at the front of the queue, however as opposed to the remove methods, examine methods leave the element in the queue and do not remove it. There are two variation methods of this operation, `element` and `peek`. `element` will throw a `IllegalStateException` if there is no element in the queue. `peek` on the other hand will not throw an exception, but will rather return a null. For the same reason as `poll`, `element` is only permitted for the string and generic data types.

6.5.1 Multi-threaded Blocking Queue

The management process may go into a deadlock if it tries to retrieve an element from the queue (using take) while the queue is empty. As the operation should block until an element is available in the queue the management process will block and wait for an element to be inserted into the queue. However, the management process will not be able to accept elements to add to the queue as it is blocked. As there is no way of entering a new element into the queue and unblocking the management process the system will deadlock. In order to prevent the system from deadlocking the management process ought to be a multi-threaded process.

Having multiple threads in the management process will ensure that when a thread gets blocked there is another thread on standby to unblock it. However, merely making the management process have multiple threads is still not enough as the process may still run into a deadlock. Consider a situation where the management has N threads. If N client processes call `take` (consumers) on the `BlockingQueue`, while the queue is empty. All the

threads of the management process will block because there are no elements in the queue, therefore they cannot continue. Even when a client process (producer) attempts to insert an element in the queue, it will wait forever as all the threads of the management process are occupied and blocked. As there is no way for the management process threads to be unblocked, the process will be in a deadlock state.

In order to prevent this we need to go a step further and prevent all the management process threads from accepting the same kind of work at the same time. The solution that has been implemented is to have two different threads. One thread only manages inserts into the queue. The second thread only manages removes from the queue. With this solution, producers may not use up all the threads, likewise with consumers. The drawback to this solution is that if you only have consumers/producers the other thread will sit idle even if it will not block.

To implement this solution we need to ensure that a thread only receives and manages the type of work that it has been assigned to handle. We could allow a thread to read in a message from the message queue, if the message is not the type of work that the thread handles, it will put the message back to the message queue. However this method may prove to be very inefficient as on average you are expecting to read half the messages incorrectly and insert them back in the message queue.

An alternative and more efficient solution is to have a second set of message queues. Therefore we will have an insert message queue and a take message queue. The relevant thread will then read from the message queue assigned to it. With this solution there will always be a thread available to insert into the queue and a thread available to remove from the queue preventing the system from going into a deadlock.

In order to be able to block when performing an action that is not allowed and unblock when the action is valid we need to use condition variables. There are two condition variables, `empty` (for remove and examine), and `full` (for inserts). When a thread blocks it calls the `pthread_cond_wait` method to wait on a condition variable until its counterpart sends a signal to it that the state of the queue has changed by calling the `pthread_cond_broadcast` method with the condition variable. For example if a remove thread attempts to remove an element from the queue (using the `take` method) and the queue is empty it will block on the `empty` condition variable. When the insert thread finally comes around and inserts into the queue it will send a signal on the `empty` condition variable. The remove thread will receive the signal, get unblocked and remove the inserted element.

When the management process is started a thread to manage takes from the queue is created, and the main thread proceeds to manage inserts. The reason for this is because the message to destroy the queue comes in from the insert message queue. Therefore having the main thread read in the destroy message allows for a clean exit.

Having multiple threads also means that a lock now needs to be implemented to protect the data structure from race conditions. The lock will be locked by the thread every time it attempts to modify or read in from the data structure. The lock is also used for the condition variables.

All of the `BlockingQueue` implementations have four message queues, two insert message queues, to send data to the management process and receive data from the management process when inserting an element, and two take message queues to use when retrieving from the queue. The String queue however has an additional data queue to send the data to and fro the management process. This is done in the same manner as that of the string array.

6.6 Summary

This chapter sets out to design and implement data structures that may be used by cooperating Java processes in a system. The data structures the JCP library attempts to implement are arrays, lists, `ConcurrentHashMap`, and `BlockingQueue`. These data structures are similar to those offered by the Java multithreaded library. The difference with these structures is that they should be usable by cooperating processes. In addition when using these data structures across cooperating processes no synchronisation methods are required as these data structures are process safe.

In this chapter the design and implementation of each data structure is discussed. The chapter first gives a high-level implementation overview of the data structures. The chapter than goes on to discuss in detail the specific implementation designs of each data structure.

Chapter 7

Executors Framework

As is the case with threads, starting and killing processes is a compute-intensive task, and it may very well be more expensive for processes (see Section 2.2.1). An executor framework to create processes that may be reused to perform different tasks without destroying them may alleviate the cost of creating processes. In the same way as the thread executor framework the framework should accept tasks and run them concurrently in a process pool.

The framework should have processes waiting for tasks to be submitted for execution. After a task has been submitted, the framework should execute the task and return a result if need be. When the process is finished with the task's execution, it should not terminate, rather wait for another task to be submitted for execution.

In this chapter, the Executors Framework is designed and implemented. The framework allows a client to submit a task to the framework. The task will be executed by the framework in a different Java process, from the process pool. The Java process will already have been started and running, waiting for the client to submit tasks, therefore relieving the cost of starting processes for each task.

The client starts the framework at the client's convenience. The client suggests to the framework how the framework should manage the process pool (how many processes should always be available in the pool, the maximum number of processes that will be in the pool at any given time, and when to kill off idle processes when they are no longer needed).

The core implementation of the system involves tasks, futures, an executor service interface, an abstract executor service class, an executor service process, and the Java virtual

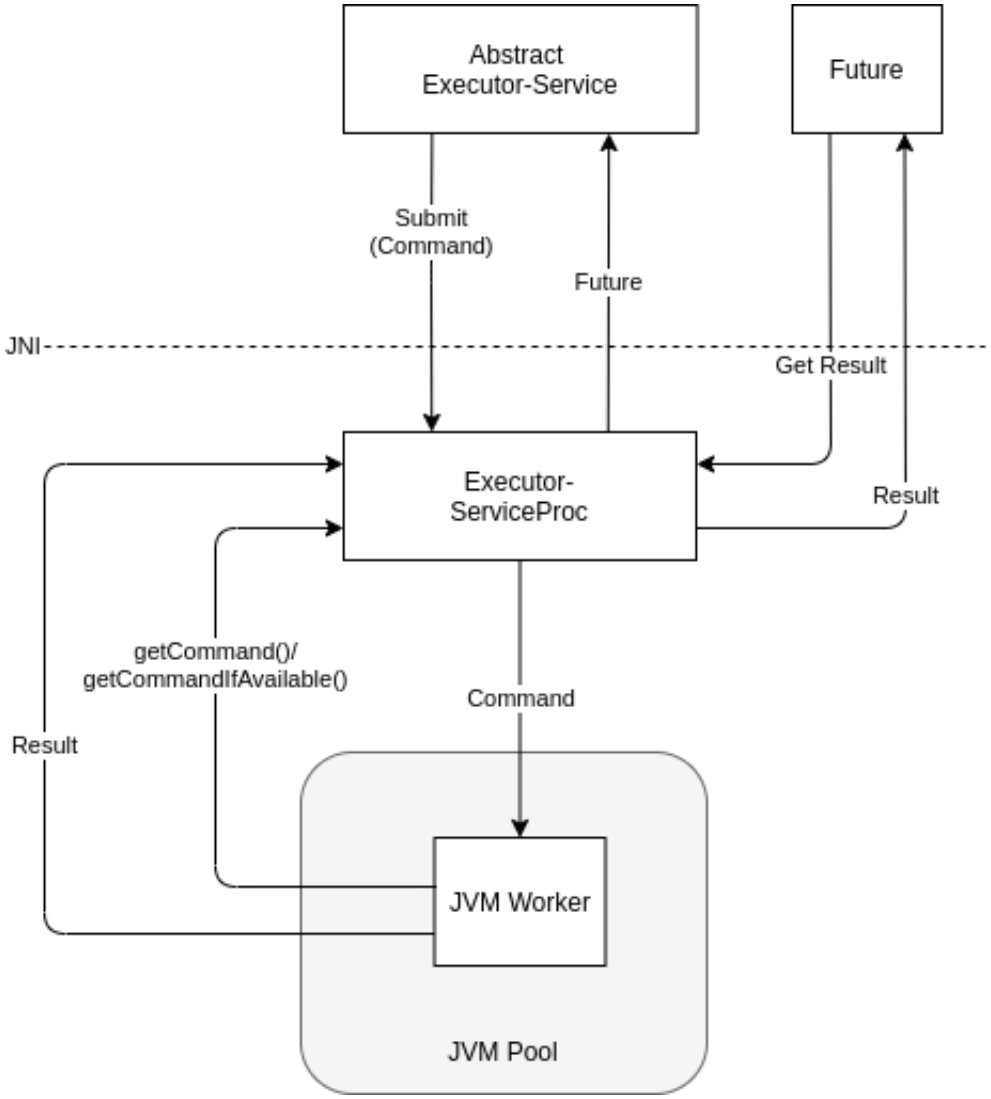


Figure 7.1: Executor Framework

machine(JVM) process. The communication and how these modules work together is shown in Figure 7.1 below.

7.1 Tasks

A task is a form of work stored in a Java class file for execution by a JVM. The framework should accept tasks and execute them in their own JVM process. There are several different tasks that the framework should be able to executable. This means that the library should be versatile in the tasks that it accepts. We could allow the framework to accept any type of class; however, this could be confusing for the framework as to how to

go about executing the tasks.

A standard on the type of tasks that may be executed by the library needs to be set, and all tasks needing to be executed by the framework need to adhere to that standard. The standard of the type of tasks the library will accept will be set with an interface. Classes that wish to be executed by the framework will have to implement the interface and adhere to the standard set by the interface . There are two different types of interfaces that a task may implement to be executable in the framework: the `Runnable` and `Callable` interfaces.

The library will create its own `Runnable` and `Callable` interfaces. The reason for creating new interfaces for the library and not using the `Runnable` and `Callable` interfaces already provided by the `java.util.concurrent` library, is because the functions of the JCP library throw exceptions not defined in the `Runnable` and `Callable` interfaces of the `java.util.concurrent` library. The task interfaces of the two libraries are similar, with the name, and the methods they implement. This is to to allow ease of interchangeability between the two libraries. A client class will only be required to change the imported library to change from one library to the other (`import` line of the code).

7.1.1 Runnable

`Runnable` is the first interface a task may implement in order to be executed by the framework. The `Runnable` interface has a single implementation requirement for the implementing classes: the class needs to implement a `run` method. The `run` method is the method that will be executed by the framework, i.e., the framework will begin execution in this method, and return when it is done with the execution.

The `run` method is a void method; therefore, no value is returned. There is no way for the client to retrieve a value stored in the class with this type of task. They may only retrieve a value by using one of the synchronization tools or data structures previously discussed in Chapter 5 and Chapter 6 respectively.

The `Runnable` interface is particularly useful for tasks that run independently on their own and close down on their own without having to return a result upon completion. The code of the `Runnable` interface is shown in Listing 7.1.

```
1 public interface Runnable extends Serializable{
2     public void run() throws Exception;
```

```
3 }
```

Listing 7.1: Runnable Interface

7.1.2 Callable

The `Runnable` task type previously discussed would perform well for running tasks that are independent of the parent process. These type of tasks, however, may perform poorly if they were to have to communicate with the parent class or return a value to the parent class when they are done computing.

These types of problems exist, for example, a factorial task that has to compute the factorial of a large number and return the results to the parent class when it is done. This would easily become messy with a `Runnable` task. The parent process would have to read in the result from a shared variable, and there would have to be synchronisation mechanisms involved as well. In order to allow for more efficient execution of tasks needing to return a value to the parent, the `Callable` interface is created.

The `Callable` interface allows implementing classes to return their execution result back to the parent process when the execution of the task concludes. The parent class may then obtain and use this result. To ensure the results may be returned to the parent process, the `Callable` interface needs to be modified to have the method return a value. The name of the method to run is now no longer `run`, it is now `call` (this is to ensure the `Callable` interface is compatible with the `java.util.concurrent.Callable` interface). As the parent process and executing process have no means of communication, the `Callable` task uses futures to return the value to the parent process. The final code of the `Callable` interface is shown in Listing 7.2.

```
1 public interface Callable<T> extends Serializable{
2     public T call() throws Exception;
3 }
```

Listing 7.2: Callable Interface

7.1.3 Future

The `Future` interface allows the parent process to gain access to the returned value after a task has completed execution. The `Future` interface is also used to access the task's

execution status. A `Future` is tied to a task when the task is submitted to an executor service through the `submit` method. The `Future` interface specifies four methods that need to be implemented by an implementing class (`FutureJCP`, Section 7.6) `get`, `isDone`, `cancel`, and `isCancelled`.

The `get` method is used to get the task's returned results. If the results are not yet ready, the method blocks until the task's execution is complete, and the results are ready. The method has an overload method, which takes in a timeout. As opposed to blocking until the result is ready, the timeout is used to resume the parent process's execution if the timeout elapses before the results are ready. If the wait timed-out, a `TimeoutException` is thrown. The method may also throw an exception if an exception was thrown while the task was executing. The type of exception, and exception message are the same as the exceptions thrown while executing the task.

The `isDone` method is a simple method that returns a boolean value indicating whether the task has completed execution or not. This method could be called before calling the `get` method to avoid blocking, and throwing a `TimeoutException`.

The `cancel` method allows the task to be cancelled if not already running. The method takes in a boolean that will allow the task to be cancelled even if it is already running. Calling the `get` method on a task that has been cancelled throws a `Cancellation-Exception`.

The `isCancelled` method checks whether the task has been cancelled already. If the task is cancelled, true is returned; otherwise, false is returned.

7.2 Executor Service

The `ExecutorService` is an interface stating that the implementing class should accept tasks and execute them in their own Java process. The process may be created, or it may be pooled depending on the implementation. The interface extends on the `Executor` interface, which has a single method, `execute`. The `execute` method takes in a `Runnable` (see Section 7.1.1) task and executes the task concurrently.

In addition to having the `execute` method the `ExecutorService` has the `submit` method to submit a `Runnable` task to the executor service for execution. The submit method differs from the execute method in that the method returns a `Future` (see Section 7.1.3),

which may then be used to get the status of execution of the task; this is not possible with the `execute` method. The `submit` method also has an overload method that takes a `Callable` task and returns a `Future`.

The `ExecutorService` interface also allows for the submission of multiple `Callable` tasks to the executor service for execution. There are two variations to submit multiple tasks, `invokeAny`, and `invokeAll` each performing differently from the other. Both of these methods contain an overload method with a timeout. When the timeout elapses, all the tasks that have not completed execution are canceled.

`invokeAny` takes in a list of `Callable` tasks and returns the result of the first task that has successfully completed execution. The remaining tasks will be canceled even if the tasks are already running. The parent process will wait for execution to complete when calling this method.

`invokeAll` also takes in a list of `Callable` tasks, and returns a `Future` list, in the order of the submitted tasks. In this method all of the tasks passed in will be executed. Each `Future` from the list may be used to retrieve the results and the status of the relevant task's execution.

The interface also states that the executor service should support methods for termination of the executor service. There are two types of termination methods mentioned by the `ExecutorService` interface, `shutdown`, and `shutdownNow`.

The `shutdown` method allows for the termination of the executor service. The executor service is shutdown after this call. No more tasks will be accepted, and the service will wait for the tasks that are submitted to finish execution. No new tasks will be accepted after this method has been called.

`shutdownNow`, on the other hand, does not wait for any tasks to complete. When the method is called, the service abruptly shuts down and cancels all tasks that have not run or are running. The tasks that have not started executing are returned to the caller with this method.

The interface defines two additional methods that need to be implemented, `isShutdown` and `awaitTermination`. The `isShutdown` method checks if the executor service has been shutdown or not; this would be useful to use before submitting a task to the executor service. `awaitTermination` accepts a timeout and waits until the executor service has terminated or the timeout has elapsed. A boolean is returned, which is false if the timeout passed, and true if the executor service has terminated.

7.3 Abstract Executor Service

The `AbstractExecutorService` is an abstract Java class implementing the `ExecutorService` interface discussed in Section 7.2. The class implements all of the methods declared by the `ExecutorService` interface and declares no additional methods. The `AbstractExecutorService` class is used to implement the interaction of the client with the executor service. Extending classes will then dictate how the executor service will work, and implement methods that will simplify the client's interaction with the framework, in particular starting the framework. The `AbstractExecutorService` class contains methods to start the executor service, submit tasks to the executor service, and shutdown the executor service.

7.3.1 Creating the Executor Service

There are two constructors for the `AbstractExecutorService`. The first constructor (connect constructor) takes in three unique keys (`sndKey`, `rcvKey`, and `dataKey`). These values are used to connect and get a reference to an already existing executor service.

The second constructor (create constructor) is used to create a new executor service in the system. The constructor takes in four additional arguments to the connect constructor. The additional arguments are `corePoolSize` (integer), `maximumPoolSize` (integer), `keepIdleProc` (boolean), and `keepAliveTime` (long). The additional arguments modify and fine-tune how the executor service should work, and behave under different work loads.

The `corePoolSize` parameter is the number of processes that the executor should always have available in the pool of processes for task execution, even if there are no tasks that need to be executed. The number of processes in the pool should never drop below this value, and if a process dies unexpectedly, the executor service should replace that process.

The `maximumPoolSize` parameter is the maximum number of processes the executor service should create regardless of the number of tasks needing to be executed. If all the current processes are busy, and there are still tasks needing to be executed, the executor service should create additional processes until the `maximumPoolSize` is reached. This allows the number of processes to increase as the size of the workload increases. At the same time the executor service will not create processes when there are no processors on the system to run them, causing processes to compete for CPU time and negatively impacting the system's overall performance.

The `keepIdleProc` parameter determines whether or not processes that have been previously created due to an increased workload should be terminated or not when the workload has decreased and there are no longer tasks needing to be executed. If this is not set, the number of processes will decrease back to the `corePoolSize` when the workload is relatively low. However if it is set the number of processes in the pool does not decrease, unless a process dies unexpectedly, which will be replaced.

The `keepAliveTime` parameter is the time unit used to determine the time length (in seconds) for the executor service to check for idle processes and processes that have died unexpectedly. This value may affect how the system performs; if the value is too low, the system performance will be negatively affected because the executor service will constantly be reviewing the work to process ratio, using up CPU resources. On the contrary, if the value is too high, it may take a while before the executor service detects that the number of expected processes in the pool and the number of running processes in the pool do not match.

When the executor service is started, through the create constructor, the required message queues are created using the keys passed in through the JNI. The Executor Service process (`ExecutorServiceProc`) (see Section 7.4) is also started. Starting the process and the message queues is performed similarly to that described for the concurrent data structures (see Chapter 6).

7.3.2 Submitting a task for execution

As specified by the `ExecutorService` interface, a task may be submitted for execution through the `execute` and `submit` methods, with each method having its own characteristics (Section 7.2). As shown in Figure 7.1 a task is passed on to the `ExecutorServiceProc` (see Section 7.4) through the JNI.

The task is passed to the `ExecutorServiceProc` through the message queues created when creating the executor service. Using the message queues requires the task to be converted into a String before being put in the message queue. The tasks may be converted into a String by serializing them to Strings in a similar fashion to the data structures.

Serialising the tasks requires the tasks to be serializable by nature. This constraint needs to be added to the type of tasks that may be created and submitted to the executor for execution. The tasks interfaces (`Runnable` and `Callable`) need to be modified to extend

`java.io.Serializable` so this constraint may be added to the tasks. This means all tasks implementing these interfaces will now be required to be serializable.

The `submit` method returns a `Future` (see Section 7.1.3) when a task is submitted to it. The `Future` may be used later by the client to get the results of the task's execution. The `Future` is created from the task's ID. The submit JNI method returns the task's ID obtained from the response message. Each task submitted to the executor service is assigned a unique task ID. The task ID, along with the message queue keys are used to create and return a `Future` object. The returned `Future` object is an instance of the `FutureJCP` class (see Section 7.1.3). The future's ID will be the same as the task's ID it pertains to.

There are methods allowing for the submission of multiple tasks; these are called `invokeAll` and `invokeAny` (see Section 7.2). These tasks are passed into the `AbstractExecutorService` in a collection. The tasks are all serialized into a String array and passed to the `ExecutorServiceProc` using the message queues. The first message sent is a message containing the method type and the number of tasks that will be sent over.

Each task will then be sent over using two messages, each sent in the data message queue. This ensures that the message is read by the `ExecutorServiceProc`'s thread (Section 7.4) already handling the action and not by another thread. The first message contains the task's length, so the `ExecutorServiceProc` will know how much data to read in from the queue. The second message then contains the data of the task itself. After the two messages have been sent, a confirmation message is read in from the receive message queue. If the method is an `invokeAll` method, the confirmation message will contain the task's ID for the `Future`, which will be stored in a list and later returned as a list of `Futures`.

7.3.3 Shutting Down the Executor Service

There are two ways to shutdown the executor with `shutdownNow` and `shutdown` methods (see Section 7.2). The `AbstractExecutorService` uses the JNI to shutdown the system. With the `shutdown` a message containing *SHUTDOWN* as the `mesg_details` is put in the message queue, a confirmation message is then waited for. The `ExecutorServiceProc` after reading the message will go on to shutdown at its convenience.

The `shutDownNow` method requires an `ExecutorService` to send back the tasks that have not started executing yet, which complicates things. In order to do this, the `Abstract-`

`ExecutorService` sends a message containing *SHUTDOWNNOW* as the `mesg_details`. A confirmation message will then be received. The message will also contain the number of tasks that should be expected to be read. An array to hold the tasks is then created using the number of tasks as the length of the array. Afterward, each task is read individually and stored in the array. The String array is then returned back to the Java side by the JNI library.

The tasks are received on the Java side as a String array. The tasks are then first de-serialised to the commands they were serialized into. If the task is a `Runnable`, then the task is de-serialized into a `Runnable`; otherwise, the task is ignored. This is because the method returns a list of `Runnable` tasks, and the `ExecutorServiceProc` returns all tasks (including `Callable` tasks) as it has no way of differentiating between the tasks. The task is then stored in a `java.util.ArrayList`, and the `ArrayList` is returned when all the tasks have been processed.

As the `shutdown` method does not stop the executor service immediately, the `isShutdown` method is provided. The `isShutdown` method tests if the executor service has been shutdown or not. The method will return true if the `shutdown` method has been called on the executor service. Using the JNI the method first tests if the `ExecutorServiceProc` is still alive using the `waitpid` Linux system call, with the *WNOHANG* option. If the system call does not return zero it means the `ExecutorServiceProc` has died and the executor service has shutdown, therefore true is returned. If the returned value is not zero the **process** is still alive. A message is then inserted in the message queue for the `ExecutorServiceProc` to read and respond with a message indicating whether the `shutdown` method has been called on the executor service or not. The returned result from the executor service will be the result of the method.

If the `shutdown` method had been called on the executor service but the `ExecutorServiceProc` is still in the process of shutting down (removing the message queue), sending and receiving a message on the message queue could fail. If the methods fail, `errno` is checked if it is equal to either `EIDRM` or `EINVAL`. If it is equal to one of them it means the `ExecutorServiceProc` had removed the message queue which indicates that the `ExecutorServiceProc` is in the process of shutting down therefore true is returned.

The `awaitTermination` method waits for the `ExecutorServiceProc` to terminate after calling the `shutdown` method. The method will block and only return when the `ExecutorServiceProc` has terminated. The method takes in a timeout to return even if the `ExecutorServiceProc` has not terminated if the timeout elapses. A boolean is returned to indicate how the method returned, false for a timeout; otherwise, true is returned.

In order to test if the `ExecutorServiceProc` is still alive, JNI along with C system calls are used. In a while loop the Linux system call `waitpid` is used with the *WNOHANG* option in the C code. The *WNOHANG* option allows the wait to return immediately. The method will then return immediately signifying if the process is still alive or not. If the process is no longer alive, it means the `ExecutorServiceProc` has terminated and true is returned. If the process is still alive the timeout time is checked to see if the timeout has elapsed, returning false if it has elapsed. The process sleeps for a second after each iteration of the while loop to save CPU resources.

7.4 Executor Service Process

The Executor Service Process (`ExecutorServiceProc`) is a single independent process that manages the submitted tasks and the `JVMWorkers` (see Sections 7.1, and 7.5 respectively). The process accepts tasks from the client process, assigns the tasks to `JVMWorkers`, accepts the tasks' results, and sends them back to the client upon the client's request. Thus the `ExecutorServiceProc` needs to communicate with both the clients and the `JVMWorkers`. The `ExecutorServiceProc` communicates with the clients and `JVMWorkers` through message queues. As the Executor Service is a complicated module in the system, a diagram of the `ExecutorServiceProc` has been depicted in Figure 7.2, along with the code executed by the process in Listing 7.3, Listing 7.4, and Listing 7.5. In the diagram, we can see the `ExecutorServiceProc`'s flow of execution, starting execution in the main thread.

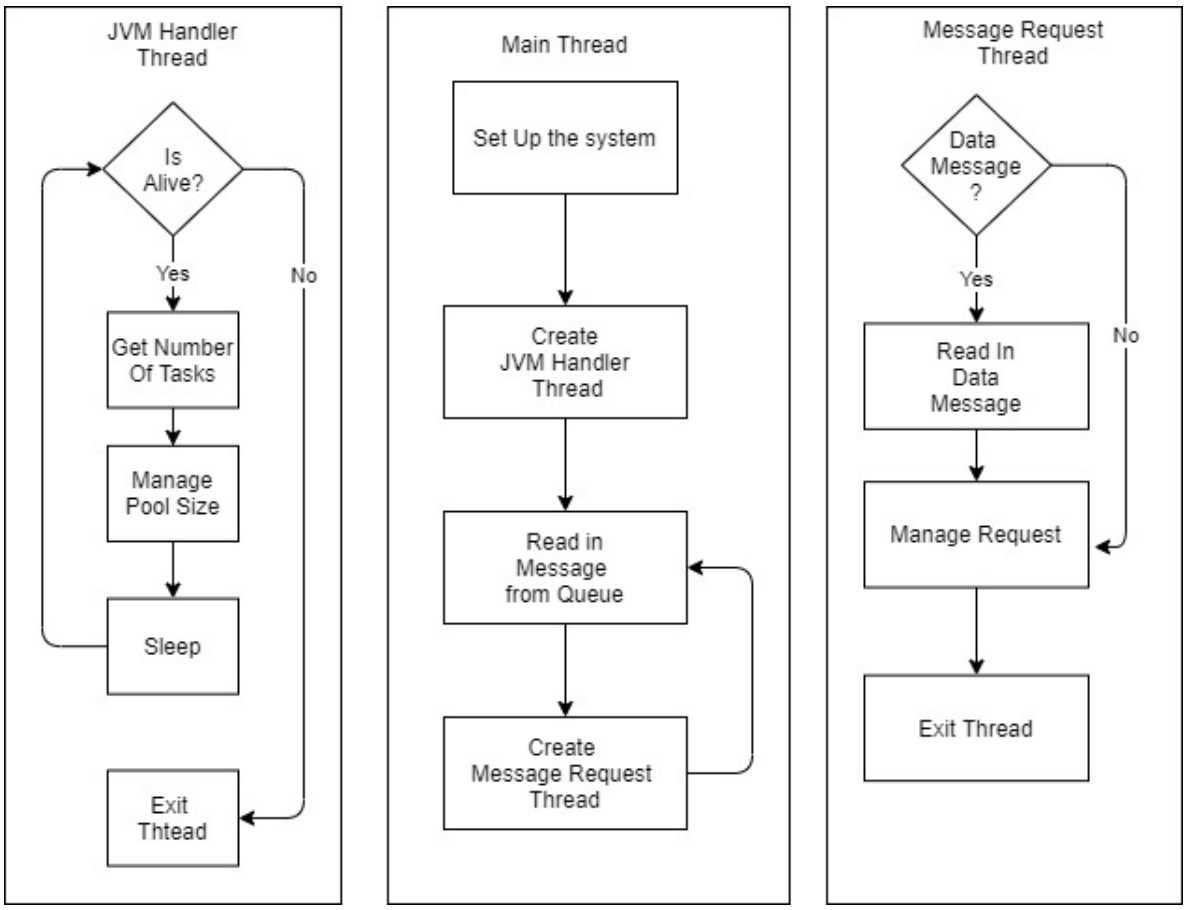


Figure 7.2: `ExecutorServiceProc` Execution Diagram

```
1
2  int main(int argc, char **argv){
3      ...
4  //SetUP
5      executorService = new ExecutorService();
6      JVMPool = new JVMS(corePoolSize,maximumPoolSize,keepIdleProc,
           keepAliveTime, sndKey, rcvKey, dataKey, executorService);
7      pthread_t jvmHandler;
8      pthread_create (&jvmHandler, NULL, JVMHandler, NULL);
9      run();
10 }
```

Listing 7.3: Setup Code

```
1  void run(){
2      while(true){
3          struct mesg_buffer* messenger = (struct mesg_buffer *)malloc(sizeof
               (struct mesg_buffer));
4          size_t size = sizeof(*messenger) - sizeof(messenger->type);
5          int readSize = msgrcv(rcvmsgid, messenger, size, 0, 0);
6          if(readSize==-1){
7              //perror(strerror(errno));
8              exit(0);
9          }
10         pthread_t tid;
11         pthread_create(&tid, NULL, runThread, (void *)messenger);
12     }
13 }
```

Listing 7.4: Main Thread Code

```
1  void * JVMHandler(){
2      while(true){
3          int numberOfTasks = executorService->getNumberOfTasks();
4          JVMPool->managePoolSize(numberOfTasks);
5          sleep(keepAliveTime);
6      }
7  }
```

Listing 7.5: JVM Handler Thread Code

7.4.1 Starting the ExecutorServiceProc

When creating the executor service, the `ExecutorServiceProc` is started by the client. The `ExecutorServiceProc` uses the values passed in the command-line arguments as keys to gain access to the message queues required to communicate with the clients and `JVMWorker` processes. The `ExecutorServiceProc` also reads in the executor service configuration arguments passed in by the client through the command line. The configuration values are then used to create the executor service and the `JVMWorker` processes specified by the client. A second thread is created to manage the `JVMWorker` processes (`JVMHandler`). The main thread waits for message requests on the message queue and handles those messages appropriately.

Starting the executor service requires the `ExecutorServiceProc` to instantiate the `ExecutorService` class to hold the data of the executor service. When the class is instantiated, a data structure to hold the tasks that have been submitted and still need to be executed is created. The tasks need to be stored in a blocking list queue data structure. A blocking list queue provides two characteristics required to store the tasks: an undefined number of elements and a first-in, first-out (FIFO) method. A data structure to store an undefined number of elements is required as we do not know how many tasks the executor system will store at any given moment. An item submitted first needs to come out first from the list to provide fairness to submitted tasks, therefore, a data structure supporting FIFO is required. The data structure also needs to block retrieves by the `JVMWorker` when there are no tasks available for execution.

A second data structure to store futures is needed and created by the `ExecutorServiceProc` at startup. This data structure's only requirement is that the data structure length needs not to be defined at creation and the data structure needs to stretch as more futures are created and need to be stored. A data structure that has an undefined length and grows as needed is a list data structure. A linked-list is created to store future elements. The class then keeps a pointer to the head of the list and the tail of the list.

Creating the `JVMWorker` processes involves storing the configuration parameters in a class. The class used to store this information is the `JVMS` class. The `JVMS` class also stores a reference to the `ExecutorService`, as well as an array that is used to store data about the `JVMWorker` processes, and the tasks they are running (`jvms` array). When the `ExecutorServiceProc` is finished with instantiating the `JVMS` class and setting up the required data structures the `ExecutorServiceProc` continues to start up the `JVMWorker`(s) using the `fork` system call and the `execvp` function.

When the child process starts executing it uses the `execvp` function to start executing the code of the `JVMWorker` class. The String *"java"* is passed on to the function as the name of the file to be executed. The arguments passed on to the function as `argv` include:

1. `JVMWorker` class path
2. send Key
3. receive Key
4. data Key

The `JVMWorker` class file path is the relative path to the `JVMWorker` class file, which is ``za.co.jcp.executors.JVMWorker''. The keys are the keys used by the message queues to pass messages to and from the `ExecutorServiceProc`. The method starts up a Java process. In turn, the Java process attempts to execute the class passed in as the first argument in the `argv` arguments list. The rest of the arguments are passed on as command-line arguments to the executing Java class.

The `fork` system call returns the process id of the child process (`JVMWorker`) to the parent process. The parent process stores the process id of the `JVMWorker` returned by the `fork` system call in the `jvms` array. `ExecutorServiceProc` only starts-up the number of processes indicated by `coreProcesses`, although the `jvms` array is defined to hold up to `maximumPoolSize`. For processes that have not been started up, a negative one value is stored in its position to indicate that there is no process running.

A thread to manage the `JVMWorker` processes is created whose sole purpose is to check the `JVMWorker` processes' status, the workload experienced by the executor service, and manage the running `JVMWorker` processes accordingly. The thread runs as an infinite loop. With each iteration of the loop, the thread gets the number of tasks in the queue waiting to be executed. With this information, the thread manages the pool size (number of running `JVMWorker` processes). The code executed by this thread is shown in Listing 7.5.

Managing the pool size involves checking processes that unexpectedly died while executing a task, killing excess processes, and creating new processes. Checking if a process died unexpectedly is done by sending a null signal to the `JVMWorker` process (sending a null signal involves calling the `kill` method and passing the process id and zero as the arguments). The method will then return a negative one value if the process is no longer

alive, and set `errno` to `ESRCH`. The return value and `errno` are tested to detect whether the process is still alive.

Killing excess processes is done to save resources and not have processes that are no longer doing work using up the CPU resources. The initial proposed solution is to have the thread killing off excess processes if `keepIdleProc` is not set. However, this solution poses an issue as a `JVMWorker` process may get killed while waiting to receive a task. This will mean that when a task is finally put into the message queue for the `JVMWorker` process, it will be lost (stay in the queue and never come out) as the process is no longer alive to read it. A solution to this problem is for the `JVMWorker` process to first be checked if it still alive before assigning a task to the process. This solution will work because the `JVMHandler` thread only kills off processes that do not have a task assigned to them.

If the number of `JVMWorker` processes currently in the pool is less than `maximumPoolSize`, and there are still tasks waiting to be executed, the `JVMHandler` thread ought to create more `JVMWorker` processes. The thread will create the processes up until one of the following conditions is met: the number of processes is equal to `maximumPoolSize`, or there are no more tasks needing to be executed. Creating the `JVMWorker` processes is done in the same manner as that described previously.

The main thread manages message requests from the client process and the `JVMWorker` processes. The main thread executes an infinite loop, with each iteration the thread reads in a message from the message queue and creates a new thread (`runThread`) to handle the message. Handling the message involves reading in data from the data message queue if any exists, performing the action specified by the message, and sending the action's results back to the client or `JVMWorker` process.

The amount of data needing to be read-in will be specified in the length field of the message. If the length is greater than zero, the thread attempts to read a message from the data queue using the type and length field acquired from the received message.

The action to be performed is recognized by the value stored in the `message_details` field. Each action is represent by a unique value. A switch statement determines the action needing to be performed. The action is performed, and the result is stored in a String if there is any.

The results obtained from the operation of the action are then sent back to the client or `JVMWorker` process that initiated the action. A message containing the action's success and the length of the result is first sent back. Even if there is no result to be sent back,

a message is still sent back as a confirmation message. The result is sent back using the data message queue and with the process id of the client/`JVMWorker` process that sent the message request set in the `type` field of the message.

7.4.2 Task Submission

To submit a task, the client may use the `execute` or `submit` methods. `execute` and `submit` store the task supplied by the client process to the queue by calling the `queueTask` method of the `ExecutorService` class. The `queueTask` method returns a future when a task is submitted. The difference between the `execute` and `submit` methods is that `execute` does not use the future returned. A message is sent back to the client to state the operation's success; if the method failed, `FAILED` (negative one) is put in the `message_details` field; otherwise, `SUCCESS` is stored. If there is a future, the future's ID is stored in the message's `taskID` field prior to sending the message back to the client. Although the future is not returned to the client for the `execute` method, the future is still created and stored. The reason for creating a future for execute is because a future is also used to keep track of what tasks the `JVMWorker` processes are running.

The `queueTask` is a method of the `ExecutorService` class. The method takes in a String (task) as an argument and returns a future. The method creates a task and a future for the task. The future and task attributes are set to their correct values. The task is then inserted into the tasks blocking queue. The future is also inserted into the future list queue. A signal to indicate that a new task has been inserted in the queue is sent on a condition variable (`empty`). The signal is sent to relieve threads that were blocked because the queue was empty.

The `invokeAll` and `invokeAny` methods allow for the submission of multiple tasks to the executor framework. Reading in the tasks is done in correspondence with the `AbstractExecutorService` (see Section 7.3), reading in the first message from the receive message queue to determine the message and then reading in the two messages for each task in the data message queue. After each task has been read in, a message with the task's future's ID is sent back to the client. A group is created to group tasks that have been submitted together. Each group has a unique group id (`groupID`). The `groupID`, along with the group type (`groupType`) is stored in the future's attributes. The `groupType` is used to identify if the group was submitted using `invokeAll` or `invokeAny`.

The `invokeAll` method returns all the tasks' futures after they have all completed execution. The JCP library currently returns the futures before they complete execution.

This issue is fixed by storing the futures in an array. After all the tasks have been read in and queued for execution, each future is waited for individually until it has completed execution. When all the tasks have completed execution, a confirmation message is sent back to the client process.

The `invokeAny` method returns results for one of the group's completed tasks. The `ExecutorService` class has a method `getGroup` for obtaining a completed task's result from a group. The `getGroup` method returns a future that it has found to have completed from the supplied group.

The `getGroup` method checks each future, starting from the head. If the future's `groupID` matches the supplied `groupID`, the future is evaluated to test if its results are ready. If the future's results are ready, the future is returned; otherwise, the next future matching the `groupID` is checked. If the end is reached before a completed future is found, the check is started from the front again. To allow the futures to be updated and save CPU resources before starting from the front again the thread waits on a `taskChange` condition variable up until one of the future's status changes.

7.4.3 Executor Service Shutdown

There are two methods to shutdown the executor service, `shutdown` and `shutdownNow`. The `shutdown` method waits for already submitted tasks to complete execution before shutting the executor service down. The `shutdownNow` method, on the other hand, shuts the service down immediately and returns the tasks that were still waiting to be executed. The `shutdown` method first sends a confirmation message to the client that it has received the message. The `ExecutorServiceProc` then wait for the tasks that are already submitted to the executor service to finish execution. The `ExecutorServiceProc` then cleans up and exits once the tasks are done executing.

The `shutdownNow` method will stop the executor service and return all the tasks still in the queue waiting to be executed. A message containing the number of tasks is sent back to the client process through the `commandType` field. Sending the tasks involves sending each task back individually. For each task, a message is sent containing the length of the task, followed by a message in the data queue carrying the task itself. This is similar to the way a task is sent to a `JVMWorker` process. After all the tasks have been sent to the client, a final message containing a `NOCOMMAND` as the `mesg_details` is sent to signal that there are no more tasks left. The `ExecutorServiceProc` then cleans up and exits.

Cleaning up involves killing all the `JVMWorker` processes still running in the system and closing the message queues. Killing the `JVMWorker` processes is done by calling the `kill` function with `JVMWorker` process id, and *SIGTERM* as the arguments. Afterward, the process sleeps for a second; then it proceeds to remove the message queues. The one-second sleep ensures that the client process has already read the confirmation message and stopped reading from the message queue.

7.4.4 Task Allocation

The `getCommand` and `getCommandIfAvailable` are actions used by the `JVMWorker` processes to obtain a task from the `ExecutorServiceProc`. These actions retrieve a task from the queue using the `nextTask` method of the `ExecutorService`. The task is then assigned to the process; if the process is no longer alive, the task is inserted back at the front of the queue as previously explained in killing of excess `JVMWorker` processes. The task, along with the task's ID, is sent to the `JVMWorker` process. In some instances, there will be no task available to send back. In such scenarios, the `getCommand` method will block until the task is available. The `getCommandIfAvailable` method, on the other hand, will return immediately, notifying the `JVMWorker` process of the failure.

7.4.5 Result Storing

The `JVMWorker` process uses `SUBMITRESULTS` action to submit the results of execution back to the `ExecutorService` class. The result as well as the success of the execution are obtained from the message queues. The success of the execution is stored in the `commandType` field of the message. The success state may comprise of two possible values: *EXCEPTION* or *SUCCESS*. These values are then stored in the relevant future (`result` and `state`). A confirmation message is then sent back to the `JVMWorker` process.

7.4.6 Get Result

These methods are for returning the results of execution back to the client. Four methods need to be implemented by the `ExecutorServiceProc`, `get`, `isDone`, `cancel`, and `isCancelled`. Each message received for these methods will contain the task's ID, which will be used to obtain the task or future to work with.

The `get` method first looks for the future to work with using the supplied task ID and the `ExecutorService`'s `get` method. The method will return the future containing the results. The future's status, which will be how the task execution completed (*SUCCESSFUL*, *EXCEPTION*, and *CANCELLED*), is stored in the message `commandType` field to be returned to the client. The message and the results are then conventionally returned to the client.

As the method may block if the results are not ready, there is an overload method that takes in a timeout and waits until the method times-out or the results are returned. In this method, the results are waited for on a `pthread_cond_timedwait` function, using the `taskChange` condition variable.

The `ExecutorService` `get` method looks for the future to work with using a while loop, starting from the queue's head and with each iteration moving onto the next element until the future being looked for is found. The future is recognized by comparing the value received from the message's `taskID` field to the future's ID. After the correct future is found, the task's status is checked to establish if it has completed execution or not. If the task has completed execution, the future is returned. Otherwise, the task is waited for using a condition variable until it completes execution, or the timeout elapses.

The `isDone` and `isCancelled` methods are similar in how they work. The `Executor-Service`'s `isDone` or `isCancelled` method is called respectively, and the results are stored in the `message_details` field. Both methods look for the future in a similar manner to that of the `get` method and then return the future's `isDone` or `isCancelled` field.

The `cancel` method calls the `ExecutorService`'s `cancel` method. The method takes in the ID of the task needing to be canceled. The method in turn, will attempt to cancel the task before it starts execution. The `cancel` method will then first attempt to find the task in the list of tasks waiting for execution; if the task is found, it is removed from the list, and its future is canceled by setting it's status field to *CANCELLED*. In addition the future's `isCancelled` and `isDone` fields are set to true, and its `status` field is set to `CANCELLED`. If the task is successfully canceled, true is returned; otherwise, false is returned.

Depending on the value that is returned by the `ExecutorService`'s `cancel` method, we will know whether the task is successfully canceled or not. If the task has not been successfully canceled and `mayInterruptIfRunning` is set, which is obtained from the `commandType` field sent in the message, we need to check the running processes. This is done by calling a `JVMPool` method `cancelTask`.

The cancelTask method goes through all the JVMWorker processes to see if they are running the task to cancel. To test if a JVMWorker process is running the task, the task's ID is looked for in the jvms array. If a process is found to be running the task, the process is terminated (to stop the tasks execution) and restarted, and true is returned; otherwise, false is returned.

If the task is canceled while running, the future is then also canceled, as discussed before. The method's success state is then returned to the client through a message, the message_details containing the value SUCCESS if the task is successfully canceled; otherwise, FAIL is placed in the message_details.

7.5 JVMWorker Process

The JVMWorker process is started by the ExecutorServiceProc (see Section 7.4) to execute the submitted tasks. Therefore, the JVMWorker process needs to obtain tasks from the ExecutorServiceProc, execute the tasks, and send the results back to the ExecutorServiceProc.

When the JVMWorker process starts it requires three long values to be passed in as command line arguments. The JVMWorker process uses these values to connect to the message queues (send, receive, and data message queue) for communication with the ExecutorServiceProc.

When the JVMWorker process is up and running, it uses the message queue to obtain a task to execute. A message is sent using the send message queue with GETCOMMAND set in the message_details field and the process id set as in the type field of the message. The JVMWorker then waits for a response from the ExecutorServiceProc on the receive message queue using its process id as the message type to read from the queue. The response message will have the type field set to the JVMWorker process' process id, the ID of the task the JVMWorker will be executing, and the length (size) of the serialised task. The JVMWorker process then reads in the task from the data message queue using the task length, and it's process id as the message type. The returned String is then de-serialized into an executable task. The task obtained is then run by the JVMWorker process.

Two values need to be returned back from the JNI method, the String form of the task and the task's ID. Returning two values proves to be an issue as there is no way for a method to return two different values. Combining the values into a String value and

delimiting the String using a known delimiter will not work as we do not know which characters will be used in the Serialized task. The approach taken to solve this issue is to have the JNI method return a new object of the `Task` class. The object will contain an integer field to store the task's ID, as well as a String field to store the command. The object is initialised in the JNI method and returned. The values of the object are assigned at initialisation time.

Another issue comes up when de-serializing, do we de-serialize it to a `Callable` task or a `Runnable` task? We do not know as the task can be of any type. This issue is solved by creating another class called `Command`. The class will store the task, as well as what type of task it is. Before the task is sent to the `ExecutorServiceProc` by the `AbstractExecutorService`, the `AbstractExecutorService` class will store the task in the `Command` class as well as the type of task it is. The `Command` class is then the class to be serialized and not the `Runnable` or `Callable` task. The `JVMWorker` process then de-serializes the String received from the `ExecutorServiceProc` into a `Command` object. The `JVMWorker` process then obtains the task type and task from the `Command` object.

Depending on the task type, the `JVMWorker` process calls different methods (`run`, or `call`). The relevant method is called in a `try-catch` statement; the catch statement catches all types of exceptions in-case the task throws an exception during execution. The reason for this is because there is no way of telling which exceptions may be thrown by the task, and if an exception is thrown and not caught, the `JVMWorker` process will crash and terminate and need to be restarted by the `ExecutorServiceProc`. Having the process restart when an exception is thrown may be inefficient as restarting a process is costly. Therefore the restarting of processes has been minimized. Currently, the only known way a `JVMWorker` process may die is if the Java method `System.exit` is called while executing the task. If an exception is thrown, it needs to be stored back in the future and thrown when a Java client attempts to get the result. To throw the exception back to the Java client, the class-name and message of the exception are stored in a delimited String and returned as results. The client will get the message and retrieve the exception class, and message using the delimiter. The chosen delimiter is the ',' character, as this value cannot be in a class-name, we do not have to worry about the message containing the delimiter, and not knowing where to split the String due to multiple delimiters. Even if the delimiter is in the message String it will not be an issue because the first occurrence is always the correct delimiter.

If the task is a `Callable` task, the call method is invoked, and the result is serialized and sent back to the `ExecutorServiceProc`. If the task type is a `Runnable`, the run method

is called, and a NULL value is sent as the result.

The results are sent through the message queues with the `message_details` set to `SUBMIT-RESULT` and the `commandType` set to the execution's success state. If an exception was thrown during the execution of the task, the success state is set to `EXCEPTION`, and the exception class and the message are sent as the result.

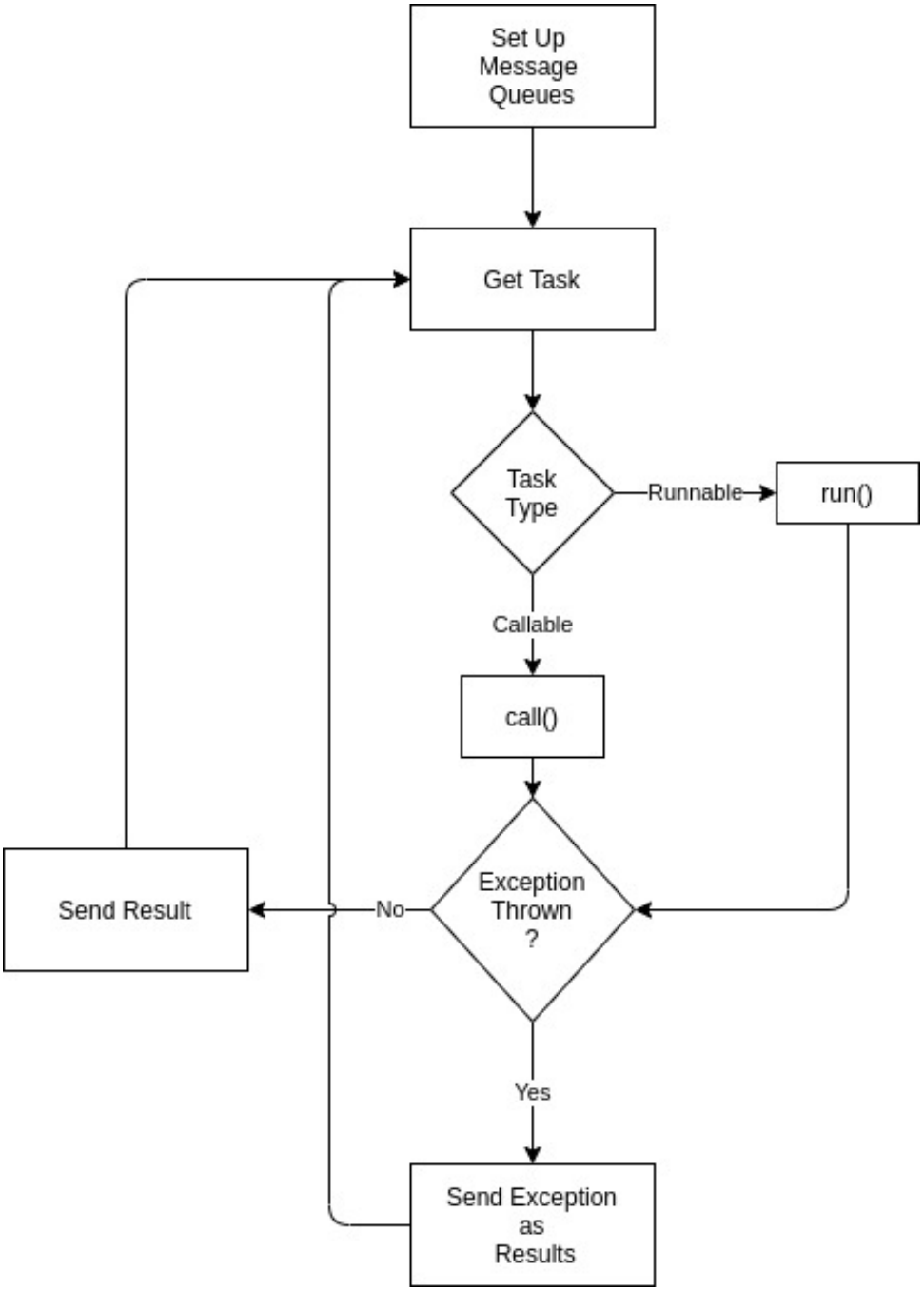


Figure 7.3: Simplified flow diagram to show execution of `JVMWorker` process

7.6 FutureJCP

`FutureJCP` is the implementing class of the `Future` interface. All the methods declared in the `Future` interface are implemented in this class. A `FutureJCP` is typically created by the `AbstractExecutorService` when submitting a task for execution. It is created using the three message queue keys and the task's ID returned by the `ExecutorServiceProc` after submitting a task.

`FutureJCP` communicates with the `ExecutorServiceProc` through the message queues. `FutureJCP` passes the method type and the task's ID through the message queues. The possible method types include `get`, `cancel`, `isDone`, and `isCancelled`.

The `get` method is called with `GET` as the value stored in the `message_details`. The method then waits for a message containing the result in a String. The message will also contain the status of the execution (`SUCCESS`, `CANCELLED`, and `EXCEPTION`). If the execution status is `SUCCESS`, the result is passed back to the Java side, and the String is de-serialized back to the required data type. If the status is `CANCELLED` a `CancellationException` is thrown by the JNI method. If the status is `EXCEPTION`, it means an exception was thrown while executing the task. The exception is then retrieved from the result and thrown.

`cancel`, `isDone`, and `isCancelled` are performed similarly. The value of the relevant method type is placed in the `message_details` field, and the message is passed to the `ExecutorServiceProc`. A response message is then received from the message queue, which will contain the result of the method. No data is received back from the `ExecutorServiceProc` for these methods.

7.7 Executor Implementations

As we have previously discussed, there is no class that allows the client to create an Executor service as the current `AbstractExecutorService` class is an abstract class. This means that a class to implement the abstract class is needed in order for a client program to use the executor service. The two implementation classes that have been developed in the JCP library are the `ProcessPoolExecutor` class and the `ForkJoinPool` class.

7.7.1 ProcessPoolExecutor

This class is a simple implementation of the executor service. There are no additional methods defined by the `ProcessPoolExecutor` class as the `AbstractExecutorService` class has no required methods to implement. The only methods provided in the `ProcessPoolExecutor` class are static factory methods, and constructor methods(which use the parent class constructors).

The executor provides three static factory methods to build a new executor service using default values (DEFAULT_KEEPALIVE = 60). These methods are `newFixedProcessPool`, `newSingleProcessExecutor`, and `newCachedProcessPool`.

An executor service created using the `newFixedProcessPool` method always contains the same number of processes in it, regardless to the workload. This is done by setting `corePoolSize`, and `maximumPoolSize` to the same value. The number of processes to have in the pool is passed in by the client class as an argument to the method.

The `newSingleProcessExecutor` method creates a pool with a single `JVMWorker` process in it. This is done by setting both the `corePoolSize`, and `maximumPoolSize` parameter to one.

The `newCachedProcessPool` method creates a pool that starts with a single process. The number of processes in the pool gradually increases as the workload increases up to a maximum of `MAX_CAP` (32767) processes. In this type of executor service, the processes do not get killed off even when the workload decreases. This executor service is created by setting `corePoolSize` to one, `maximumPoolSize` to `MAX_CAP`, and `keepIdleProc` to true.

7.8 ForkJoinPool

`ForkJoinPool` extends the `AbstractExecutorService` class. `ForkJoinPool` is intended for problems that are efficiently solved by employing the divide-and-conquer pattern. In these problems a task is recursively broken down into multiple sub-tasks until each task may be efficiently solved sequentially. This implies that a task needs to be able to split into multiple sub-tasks, and resubmit its sub-tasks back to the executor service executing it and obtain the sub-tasks results when they are done. The QuickSort algorithm is an example of a problem that employs such a pattern.

7.8.1 ForkJoinPool

The `ForkJoinPool` is more complicated than the previously discussed `ProcessPoolExecutor` class. The class implements the default constructors of the parent class, a single constructor of its own, and a static factory method. The constructor implemented by the `ForkJoinPool` takes in a parallelism integer. The integer is used to define the `corePoolSize`, and `maximumPoolSize` is set to `MAX_CAP`. This allows the pool size to increase as more tasks are recursively submitted into the pool up until `MAX_CAP` is reached. At `MAX_CAP` creating additional processes would probably be more detrimental to the overall performance of the system compared to running the tasks in the processes currently in the pool.

The class also contains a static factory method to create a new pool (`createDefaultPool`). The `corePoolSize` in this constructor is set to the number of available processors at run time. The number of available processors in the system is obtained from the `Runtime.getRuntime().availableProcessors()` method. The maximum size of the pool is then set to `MAX_CAP`. This method resembles the `commonPool` method of the `java.util.concurrent.ForkJoinPool` class, however these two methods differ from each other, which is why they have different names. The difference between these methods is that the `createDefaultPool` is affected by the `shutdown`, and `shutdownNow` methods, while the `commonPool` method is not. The reason for this is because the JCP pool may need to continue running even after the process that created the pool terminates. With multithreaded `java.util.concurrent.ForkJoinPool` the executor service will terminate with the thread, however with multiprocesses there may still be other processes in the system needing to use the executor service.

The class provides extra functionality to the executor service by supplying an extra method. The method is an `invoke` method that submits a `ForkJoinTask` to the executor service for execution. The method accepts a `ForkJoinTask`, and the class of the return type, similar to that of `submit` method in the `AbstractExecutorService` in Section 7.3. The method then submits the task using the `AbstractExecutorService`'s `submit` method. The future of the `ForkJoinTask` is then set to the future received from submitting the task through the `ForkJoinTask`'s `setFuture` method. The reason for implementing the `invoke` method is that, unlike a standard `Callable` task, a `ForkJoinTask` keeps its future with it; therefore, the task's future needs to be set after submission. This makes the standard `submit` method not fully suitable to be used with a `ForkJoinTask`.

7.8.2 ForkJoinTask

`ForkJoinTask` is an abstract class implementing the `Callable` and `Future` interfaces. The class implements all the `Future` methods; an `invoke` method, a `join` method, and a constructor method. The `call` method of the `Callable` interface is not implemented; this method will need to be implemented by an extending class.

The class keeps an internal `Future` within it. All the implemented `Future` methods call the internal `Future`'s methods. The `Future` needs to be provided when the task is submitted for execution through the `setFuture` method. The `ForkJoinPool` and `ForkJoinTask` classes already do this automatically if the task is submitted through their `invoke` methods.

The `invoke` method in the class is used to submit a second `ForkJoinTask` (usually a subtask) for execution to the executor service through the `ForkJoinPool`'s `invoke` method. The method simply calls the pool's `invoke` method. Since the `ForkJoinTask` does not have a reference to the `ForkJoinPool` executing it, a reference to the `ForkJoinPool` needs to be created. A `ForkJoinPool` is created using the `ForkJoinTask` keys.

The `join` method is the most complicated method of the `ForkJoinTask` class. The method is used to obtain the results of a `ForkJoinTask`, and if the results are not yet ready the method attempts to obtain and run another task from the executor service. The method first checks if the results are ready through the `isDone` method. If the results are ready, the result is obtained through the `get` method and returned. If the results are not yet ready, the process attempts to obtain another task from the executor service (using the `getAndExecuteCommandIfAvailable` method) and execute it. When the execution of the second task is complete, the initial task is tested again to see if it is complete; if not, the process starts again. Getting a task may fail if there are no tasks needing to be executed in the executor service, if this occurs it means the task is currently getting executed by another process in the system. The process, therefore, waits by sleeping for 100 milliseconds to save CPU resources before attempting to check again. Listing 7.6 below shows the code of the `join` method.

The constructor of the class simply accepts the message queue keys and stores them internally. These keys are needed when the `invoke` method of the class is called. This is because before the `ForkJoinPool`'s `invoke` method is able to be called, the `ForkJoinPool` needs to be obtained. Obtaining the pool then uses the keys stored. It is not possible to expect the `Future` to be supplied during initialization of the task as the `Future` is only

obtained after the task has been submitted. Therefore the `Future` has to be set at a later stage.

```
1   public T join() throws CancellationException, ClassNotFoundException,
        ExecutionException, SystemException, IOException {
2    while (!isDone()) {
3     boolean commandAvailable = ForkJoinJVM.getAndExecuteCommandIfAvailable(
          sndKey, rcvKey, dataKey);
4     if (!commandAvailable)
5      try {
6       Thread.sleep(100);
7      } catch (InterruptedException e) {
8       e.printStackTrace();
9      }
10   }
11   return get();
12     }
```

Listing 7.6: ForkJoinTask's join Method

7.8.3 ForkJoinJVM

The `ForkJoinJVM` class works in a similar way as that of the `JVMWorker` discussed in Section 7.5. `ForkJoinJVM` is intended to be used by `ForkJoinTasks` waiting for their results to be ready. The class contains a single public static method accessible to other classes `getAndExecuteCommandIfAvailable`. The method attempts to obtain a command from the `ExecutorServiceProc` (see Section 7.4) and execute it. In contrast to the `getCommand` from the `JVMWorker` class, the `getCommandIfAvailable` method allows the calling process to not block and return immediately if there is no command available for it to execute. If a command is obtained, the command is executed in a similar way to that of the `JVMWorker`, and the method returns true. Otherwise, nothing is done, and the method returns false.

7.9 Summary

In this chapter the design and implementation of the executors framework is discussed. The discussion begins with the type of tasks that are executable by the framework, which includes `Runnable` and `Callable` tasks. The difference between the two types of tasks

is also analyzed and discussed in this chapter. The implementation of the core functionality of the framework (implemented through the `AbstractExecutorService` class) is discussed in detail. The `ExecutorServiceProc` which manages the tasks that have been submitted for execution is discussed, as well as how it integrates with the `AbstractExecutorService`, the `FutureJCP`, and the `JVMWorker` processes which are the processes that execute the submitted tasks. The implementing classes of the `AbstractExecutorService` (`ProcessPoolExecutor`, and `ForkJoin Pool`) are discussed last, and the functionality they add to the `AbstractExecutorService`.

Chapter 8

System Testing

This chapter discusses how the classes of the JCP library are tested. The tests for these classes aim to assess if the JCP classes perform as expected. The testing framework used to perform these tests is the JUnit 4 framework. The components of the JCP library to be tested are the shared variables, atomic variables, data structures, and synchronisers. Testing the executors framework is difficult using unit tests. Therefore the executors framework has been tested in conjunction with the system benchmarks in Section 9.3.

8.1 Shared Variables and Atomic variables

A test class is written for each of the shared variables available in the library, although they are very much similar. Since the tests are very similar, only one of the shared variables tests will be discussed. The class to be discussed is the `JCPBoolean` class.

The initialisation methods (`new JCPBoolean()`) and `destroy` methods are tested first in a single test method called `set_Up_Connection`. The constructor to get access to an existing JCP variable is also tested in the `set_Up_Connection` test method. Testing the `destroy` method involves calling any arbitrary method from the JCP variable after the `destroy` method has been called. A `SystemException` is then expected to be thrown, with a message of "Invalid argument", when calling the method. The type of exception as well as the message of the exception are inspected to ensure the expected exception is thrown by the library. If the exception is not thrown, or the exception thrown is not the expected exception, it means that the variable was not successfully destroyed by the `destroy` method therefore the test fails.

The other two methods that are tested are the `get` and `set` methods. The `get` method is tested first to retrieve the value that is set when the variable is initialised. The `get` method is called after the value is initialised. The value returned by the `get` method is tested to see if it is the expected value set during initialisation through the `assertEquals` method. The test will fail if the expected value and the value returned by the `get` method are not equal.

After the `get` method has been tested and verified it is working correctly, the `set` method is tested. In the test of the `set` method the value set during initialisation is changed through the `set` method. The value stored is then retrieved using the `get` method. The value returned by the `get` method is then tested to see if it is equal to the value the `set` method has changed it to. If the value retrieved by the `get` method is equal to the value set by the `set` method the test passes, otherwise the test fails as the `set` method has not been able to change the value (or changed it to an incorrect value) of the shared variable. We also know that the method with the bug is the `set` method as the `get` method has already passed.

`JCPString` does not have a `set` method. To modify a `JCPString` variable, the variable needs to be destroyed and recreated. A `String_Modification_Test` method is written to test if the `JCPString` class behaves as expected if the `JCPString` is attempted to be modified by recreating it without destroying it first. The `JCPString` class should throw an exception when `JCPString` is recreated without destroying it first. The test method tests if the exception is thrown and if the type of exception thrown is the expected exception. The message of the exception is also tested if it is the expected message of the exception.

The Atomic classes go a step further and define atomic functions not available in the shared variable classes. This means tests also need to be written for these methods. As the atomic classes use the underlying gcc functions, it is assumed that the gcc functions used are concurrency-safe, and no concurrency issues may arise if they are used correctly. Therefore the tests only test if the relevant methods use the gcc functions correctly and not necessarily if they are concurrency-safe.

There are four types of methods that the atomic classes introduces, these methods are modify and get (`addAndGet`, `incrementAndGet`, and `decrementAndGet`), get and modify (`getAndAdd`, `getAndIncrement`, and `getAndDecrement`), exchange values (`getAndSet`), and compare and set (`compareAndSet`).

The modify and get tests involve initialising the variable to a known value, calling the relevant method and testing the value returned by the method to see if it is equal to the

value expected to be returned by the method. If the returned value is as expected the test passes as it means the variable has been successfully updated, otherwise the test fails as the update has not been successful.

The get and modify tests first call the relevant method after initialising the variable. The value returned by the method is then tested to ensure that it is equal to the value the variable was initialised to. If the values are different the test fails as the method gets the incorrect value. A further test is performed to see if the value has been updated to the expected value. This is done by calling the `get` method and then comparing the value returned by the `get` method to the expected value.

With the exchange values tests the value x is set at initialisation. The `getAndSet` method is then called with a value y which is a different value to the value the variable was initialised to. The result of the method are then tested to see if they are equal to value x. The test then gets the value stored in the variable using the `get` method. The value is expected to be equal to y which was set by the `getAndSet` method.

The `compareAndSetTest` method first initialises the variable to value x. The `compareAndSet` method is then called with x as the expected value and y as the new value to set. The results of the method are tested if it is equal to true, as the method is expected to succeed. The value stored in the variable is then tested to see if it is equal to the newly updated value (y) using the `get` method. The test then calls the `compareAndSet` method using value x as the expected value, and an arbitrary value (v) as the value to set the variable to. The operation at this point is expected to fail as the value that is stored in the shared variable is y and not the supplied x. The test method then tests if the call to the `compareAndSet` method did not update the value as the operation failed. The test is performed by calling the `get` method on the variable and the results are tested to see if they are equal to x. If the results are not equal to x then the test failed as the value had been modified by the `compareAndSet` even though the value passed in as the expected value was not the value stored in the variable.

8.2 Synchronisation

The synchronisation classes are tested differently to the shared variables and atomic classes. The synchronisation classes testing involves creating multiple processes that will perform the type of synchronisation being tested. The behaviour of the processes is then analysed to see if it is as expected.

8.2.1 Lock, ReadWriteLock, and Mutex

Testing the `lock` and `mutex` classes involves testing the lock functions (`lock`, and `acquireKey`) and the try lock functions (`tryLock`, and `tryAcquireKey`). Both the tests create a lock/mutex simply referred to as the lock from here on, and a `JCPLong` shared variable (see Chapter 4.1). The tests then start ten client processes that will be updating the shared variable concurrently. The clients will be using the lock to protect the shared variable against race condition. When the processes are finished the test method will check that the value of the shared variable is correct, then destroy the lock.

The client processes obtain access to the lock and the shared variable using a key. After the client process has obtained the lock the process executes a for loop of a thousand iterations. With each iteration the process locks the lock, gets the current value of the shared variable, increments the value, sets the shared variable to the updated value, and unlocks the lock.

The `tryLock` method may fail if the lock is held by another process. If the `tryLock` method fails it is attempted again without going into the next iteration until it succeeds.

The results are then checked by the main process to see if no race conditions occurred. The test is done by checking the value of the shared variable and comparing it to the expected value which is the number of client processes multiplied by the number of iterations per process.

These tests create contention for the lock as it is run with ten processes competing for the same lock. There is also a high chance of a race condition as the variable is updated multiple times (thousand times) by multiple processes (ten processes). The test was ran five times without the locking mechanism in place, and each time the tests failed, as there was no protection against race conditions. However, with the lock implemented the tests always passed. This would mean that the lock is able to successfully provide mutual exclusion.

With the testing of the `ReadWriteLock` another group of client processes are added to the mix. Now there are two groups of client processes: writers and readers. The writers perform in the same manner as the client group of the `lock`: acquire the write lock of the `ReadWriteLock`, update the value of the shared variable and release the write lock, repeat the process a thousand times. The readers acquire the read lock, read the value, and release the lock, an infinite number of times. What this does is allow the writers to write to the shared variable while the readers are also attempting to read from the shared

variable. The main process starts these two group of processes and waits for the writers to terminate. The reader processes are then terminated by the main process using the `destroy` method. The value of the shared variable is checked to see if it is correct. If the value is not correct then the test fails as a race condition has occurred.

8.2.2 Condition Variables

The `Condition` class is tested in a similar fashion to the locks. A `Lock` instance is created by the main process as well as the condition variables. The client process accesses the lock and condition variable through defined keys. A `JCPLong` shared variable is used in the test. The shared variable will be incremented by the client process till it reaches a thousand. The main process will wait on the condition variable until the client process is finished incrementing the shared variable. When the client is finished it will send a signal on the condition variable waking up the main process. The main process will then wake up and test if the shared variable has been updated to the correct value. We will know the class is correct if the main process eventually wakes up and the value is what is expected. If the main process wakes up before the signal is sent, the test fails as the condition variable is not performing as expected. On the other hand, if the main process does not eventually wake up the `Condition` class contains a bug. The bug may be with the `signal` method or the `await` method.

The `awaitNanos` method is tested by waiting on the condition variable for a thousand nanoseconds. The result of the wait is tested, which is the remaining wait time. If the result is less than or equal to zero, it means the wait timed out therefore the client process is not finished. If the client process is not finished it means the shared variable is less than thousand, otherwise if it is finished it should be thousand. Both cases are tested by the test method.

8.2.3 Cyclic Barrier

Testing the `CyclicBarrier` class involves creating five test methods (`setUpTest`, `await-Test`, `awaitTimeoutTest`, `resetTest`, and `getNumWaitingTest`).

The `setUpTest` test method tests the creation and destruction of a `CyclicBarrier`. The test is performed by first creating a new `CyclicBarrier`. The CyclicBarrier is then

destroyed using the `destroy` method of the `CyclicBarrier`. To ensure the `CyclicBarrier` is successfully destroyed, the `await` method is called on it, which should throw a `SystemException`.

With the rest of the tests ten client processes are started. The client processes simply obtain a reference to the `CyclicBarrier` using a predefined key. After obtaining a reference to the `CyclicBarrier` a client process calls the `await` method, and terminates.

The `awaitTest` and `awaitTimeoutTest` test methods create a `CyclicBarrier` with eleven parties (for the main process and the ten client processes). The client processes are then started and the main process then calls the `await` method on the `CyclicBarrier`. When the `await` method returns it means all the client processes have successfully called `await`. The client processes will terminate at this point and the main process will continue with the test. If any exception is thrown at this point the test fails. The client processes are started up again and the `await` method on the `CyclicBarrier` is called again. This is done to test if the `CyclicBarrier` had been successfully tripped.

The `awaitTimeoutTest` needs to test the method `await` method with a timeout supplied. In order to perform this test, the main process first calls the `await` method with no client processes running therefore the CyclicBarrier will never be tripped, causing it to inherently timeout. After the CyclicBarrier has timed out, the CyclicBarrier is reset. The processes are then started and the normal execution flow of the method is tested.

The `reset` and `getNumberWaiting` methods are trivial to test. In the `reset` method, the CyclicBarrier is reset by the main process while the client processes are waiting on it. To ensure the client processes are ready and waiting the main process sleeps for three seconds before resetting the barrier. The waiting client processes will throw a `BrokenBarrierException` at this point. The exception is printed on the command line for visual inspection. After resetting the CyclicBarrier the main process starts ten additional client processes. These new client processes and the main process will wait on the CyclicBarrier again. This is to ensure that the CyclicBarrier had been reset to the correct state and may be used correctly from here on. The `getNumbersWaiting` method is tested by calling the `getNumbersWaiting` method before the client processes are started which should return a zero.

8.2.4 Semaphore

Testing the `Semaphore` class has proven to not be a simple task using JUnit tests. The issues arise because we would like to allow a limited number of processes into the critical section, but not necessarily only one process. To test the `Semaphore` class a single `Semaphore`, and ten client processes are created. The number of processes that may enter the critical section (acquire a `Semaphore`) at any given time is set to two so the ten processes will be competing for the two permits of the `Semaphore`. In the critical section each process will acquire the `Semaphore`, increment a shared `AtomicInteger`[1] value, test if the value is less than two, go to sleep for a hundred milliseconds, decrement the value, and then release the `Semaphore`. If more than two processes enter the critical section (this means the `Semaphore` contains a bug) the value of the `AtomicInteger` will be greater than two, when this happens the code throws an exception to indicate that the test has failed. To increase the possibilities of the tests failing the critical section code is executed a hundred times by each of the client processes.

8.3 Data Structures

The concurrent data structures consist of four main categories: array, `BlockingQueue`, `ConcurrentHashMap`, and lists. Barring `ConcurrentHashMap` each category has multiple data structures to natively support Java primitive data types: char, double, integer, long, string, and generic types. `BlockingQueue` also includes two queue types (i.e. list and array types) for each of the class types. Each of these classes has been tested with JUnit tests, testing each method of the class in the process.

8.3.1 Array

For each of the array test classes there are five test methods that need to be implemented, `set_Up_and_Destroy_Test`, `containsTest`, `getAndSetTest`, `indexOfTest`, and `lastIndexOfTest`.

The `set_Up_and_Destroy_Test` needs to test for the creation of the array and the termination of the array when the library is done with it. The method creates an array using

[1] An `AtomicInteger` shared variable is used because the two concurrent processes could still cause a race condition if the ordinary shared variables are used.

the required keys, and the length. The method then creates a second reference to the array using the keys. After the method has verified that the two initialisation methods are correct the method destroys the array using the second array reference. Afterwards using the first reference to the array the array is tested to see if the array has been successfully destroyed. The test is performed by calling any method on the array and a `SystemException` should be received with "Invalid argument" as the message, otherwise if the method is successful then the array has not been destroyed properly. Using the two different references to work with the array ensures that both the initialisation methods of the array work as expected. This also ensures that arrays initialised using the same keys are linked together. Performing the test in this manner tests three different methods and if they function properly.

The `getAndSetTest` performs a test on the `get` and `set` methods of the array. A value is first `set` into one of the indexes of the array and the value is then retrieved. A test on the returned results is performed to see if the value inserted is the value returned. If the value matches then both methods work correctly, otherwise one of the methods is malfunctioning. In addition, the `IndexOutOfBoundsException` is tested. The methods are called with an invalid index and the exception is expected to be thrown.

Testing the `indexOf`, `lastIndexOf`, and `contains` method involves setting a value at one of the indexes of the array. For `indexOf` and `lastIndexOf` methods the value is set twice at different indexes and the relevant method is called with the value. The index returned by the method is then checked to see if it matches the expected index value. The methods are also called with a value that does not appear in the array therefore expecting negative one as the result. The `containsTest` simply calls the `contains` method after the value has been set, and checks the results against an expected value.

8.3.2 Lists

Lists also include the same five tests as the array tests, and which are implemented in a similar fashion. As the list class includes other extra methods compared to the array class, the testing of these methods also needs to be implemented. These methods include `remove`, `addLast`, `addFirst`, `add`, `clear`, `getFirst`, `getLast`, `removeFirst`, `removeFirstOccurrence`, `removeLastOccurrence`, and `removeLast`. Some of these tests are similar to each other, therefore to avoid repetition they are discussed together.

The remove methods (`remove`, `removeLast`, and `removeFirst`) are tested by inserting values into the array. For `remove` a value is then removed at an index, other methods

will simply remove the value at the beginning, or end of the list. To test whether the value has been successfully removed the `indexOf` method is called in the list, the return value of the method should be negative one if the value has been successfully removed. The `removeFirst`, and `removeLast` methods test the success of the method by calling `getFirst`, and `getLast` respectively. The result should be the value which was second in the list or second to last depending on the method.

For the add methods (`add`, `addLast`, and `addFirst`) values are inserted into the list. The values are then tested to determine if they were added correctly and in the right position. The test is done by calling the `get` method on an index with a known value, the returned value is then compared to the expected result. The `add` method also tests `IndexOutOfBoundsException` . The test is performed by calling the `add` method with an index that will throw the exception. If the exception is not thrown, there is an error with the method, otherwise the method correctly throws an `IndexOutOfBoundsException`.

The remove occurrence (`removeFirstOccurrence`, and `removeLastOccurrence`) methods remove the first or last item found matching the supplied value. In order to test the method multiple values are inserted into the list, with multiple positions containing the same value. For example, there will be a list containing the value x at two different positions in the list, maybe at index zero and index two. When the method `removeFirstOccurrence` is called with x as the value to remove the method should remove the x at index zero and leave the x at index two, which will become index one. In contrast `removeLastOccurrence` will remove the x at index two, leaving the x at index zero. The methods then need to be tested to see if they perform as expected and leave the array as expected. The test is done by calling the `indexOf`, and `lastIndexOf` method with the removed value (x in this example). The result should return the index of the value the method did not remove. In addition to removing the value, the method returns the success of the functions. When the method is called the returned value is checked to see if it is correct. A value not in the list is also attempted to be removed using these functions and the result is tested to see if it is negative.

Testing the `clear` method is simple. The test simply adds values to the list and then clears the list. To test if the list has been successfully cleared a value at index one is attempted to be added of which an `IndexOutOfBoundsException` should be thrown as the list is empty.

8.3.3 Blocking Queue

As with arrays and lists, the tests of the different types are very similar therefore only the integer test will be discussed. First the `BlockingQueue` initialisation and `destroy` methods are tested. The test is performed in a similar way to that of the lists and arrays. The other methods tested are the `add`, `offer`, `put`, `offer` with a timeout supplied, `take`, `poll`, `remove`, `poll` with a timeout supplied, `element`, and `peek` test.

Testing the `add` and `put` method involves creating a `BlockingQueue` with N elements. N elements are then added to the queue using the `add`/`put` method. There should be no failure with these adds. For the `addTest` method another element is added to the queue, at this point the queue should throw an `IllegalStateException` which is tested by the test. If the exception is not thrown the test fails. With the `put` method no extra element is added as the test process will block forever (deadlock), as there is no process to remove elements from the queue, and unblock the test process.

`offer` differs from the `put` and `add` methods as the `offer` method will return immediately and not throw an exception even if it was not successful while trying to add an element to the queue. The test is also performed with a queue containing N elements. N elements are added to the queue, and the return value is tested using `assertEquals` method. The return value should equal true. Another element is then attempted to be added to the list using `offer`. As the queue is full, the return value should be false. The return value is tested to see if it is correct. The test of `offer` with timeout is the same as `offer` except a timeout is supplied.

The `take` and `remove` methods differ from each other because `remove` throws a `NoSuchElementException` if the queue is empty while the `take` method will block and waits till there is an element in the list. In the test a queue with a length of one is created. An element is added (using the `add` method) and removed from the queue (using `remove`, or `take` method). This is repeated a hundred times. The returned value of the method is tested to check if it is equal to the added element. For the `remove` method test, a remove is attempted to be performed on the empty list, for which a `NoSuchElementException` is expected to be thrown by the method. For the same reason as the `put` method no element is attempted to be removed using the `take` method on an empty queue.

The `poll` and `poll` with timeout methods attempt to remove an element returning null if there is no element. These methods are only supported for String and generic queues. The `BlockingQueue` tests for the unsupported queue types only checks to see if the method

throws an `UnsupportedOperationException`. With String and generic queues a queue of length hundred is created. In a for loop (repeating 150 times) elements are added and the `poll` method is called twice. The value returned from the first `poll` call should equal the value added to the queue, and the value returned from the second call should be null. The loop is repeated 150 times to ensure the elements are getting removed. If the elements are not getting removed the `BlockingQueue` will become full at the 100^{th} iteration, and additional adds to the `BlockingQueue` will throw a `SystemException`.

The `element` and `peek` methods attempt to get the first element without removing it from the queue. These methods differ in that `element` throws a `NoSuchElementException` if the queue is empty and `peek` returns null. As with `poll`, `peek` is only supported by the String and generic arrays. In the tests a queue of a hundred elements is created. In a for loop (of a hundred iterations) the test inserts into the queue by calling the `add` method, and examines the element at the head of the queue through the `peek`, and `element` methods. The value returned by the methods is tested if it is equal to the value first inserted in the queue. In order to ensure methods are not removing the elements, another add is performed. At this point an exception should be thrown as the queue is full. In the tests the `peek`, and `element` methods are called before any element is added to the queue and the value returned is tested to check that it is null, and a `NoSuchElementException` is thrown by the `element` method.

8.3.4 Concurrent Hash Map

With the exception of setting up and destroying of the `ConcurrentHashMap`, there are three methods that need to be tested: the `put`, `get`, and `remove` methods. Setting up and destroying the `ConcurrentHashMap` is performed in the same way as the other data-structures. The `ConcurrentHashMap` is tested using a `ConcurrentHashMap` that uses the Integer type for the key and the String type for the values.

When testing the `put` method the value "Hello World!!" is inserted into the `Concurrent-HashMap` using the `put` method. Zero is set as the key.

With the `get` method test, the `get` method is called directly after creating the `Concurrent-HashMap`. The `get` method should return a null no matter what key is supplied as there are no values in the `ConcurrentHashMap` as of yet. A value is then inserted into the array using the `put` method. The value is then returned from the `ConcurrentHashMap` using the `get` method. The returned value is then analysed to see if it is equal to the inserted

value. The `get` method is then called one more time to ensure the value still remains in the `ConcurrentHashMap`.

Testing the `remove` method is identical to `get` method test, except the the `remove` method is called instead of the `get` method. The tests also differ in that as the value gets removed after calling the `remove` method, so calling the method again should return null.

8.4 Summary

In this chapter the testing of the JCP atomic variables, shared variables, synchronisation, and data structure classes is discussed. The classes are tested using JUnit. All of the tests passed and none of the tests failed. With the tests passing it means the classes have been implemented correctly and may be used for benchmarking the performance of the JCP library which is discussed in the following chapter.

Part III

Findings & Results

Chapter 9

Results and Discussion

In this chapter, the library's performance is measured against other parallel computing methods currently available to Java programmers. The alternative parallel computing method under consideration that the JCP library will be benchmarked against is the `java.util.concurrent` thread library. First, the synchronization classes' performance is analyzed, then the performance of the data structure classes is analyzed, and finally, the executor service's performance. The performance of both libraries will be analyzed and compared against each other. The aim of the project, however was not to improve on the performance of the threading library, but rather to extend the functionality of Java with improved support for multiprocessing concurrency. These benchmarks are performed to realise the performance difference between the two libraries. This chapter also discusses the code portability between the two libraries, as well as the difficulty of using the library.

All the benchmarks have been performed on the computer described in Chapter 3. To increase the accuracy of the benchmark results each benchmark is run with no other user applications running on the system.

9.1 Synchronisation

With synchronisation there are plenty of classes that could be used to benchmark and test the libraries' performance. The possible test classes include `Condition` class, `CountDownLatch` class, `CyclicBarrier` class, `Lock` class, `Mutex` class, `ReadWriteLock` class, and the `Semaphore` class. The class that has been chosen to be benchmarked is the `Lock` class

as arguably, it is the most versatile synchronization method and will be the most used synchronization method in the library.

For the locking benchmarks, the same number of threads or processes are created depending on the test. A lock is also created which is of the `java.util.concurrent.locks.ReentrantLock` or `za.co.jcp.synchronisers.lock.Lock` package. The processes and threads go into a for loop (for ten thousand iterations), simply locking and unlocking the lock in each iteration. This means each process or thread will lock and unlock the lock ten thousand times. The lock contention will depend on the number of processes and threads used in the test. This value changes with each test, starting from one going to ten, and increasing in values of tens thereafter that until it reaches hundred.

The benchmark will only measure the critical section's timing and not the whole program (creating the processes/threads, locks, and destroying them). This is because we are mainly concerned with how long it will take a process or a thread to lock the relevant lock and how much the lock's performance will deteriorate as lock contention increases.

In order to ensure that only the timing of the critical section is tested, a `CyclicBarrier` is used. A `CyclicBarrier` is used to ensure the timer is not started until all the processes or threads are ready to start executing the critical section. The `CyclicBarrier` will be tripped when all the processes and threads are ready and the main process or thread will start the timer and wait for all the processes or threads on the `CyclicBarrier`. When a process or thread is finished running its critical section code it will call the `await` method of the `CyclicBarrier`. When all the threads or processes are finished executing their critical section the `CyclicBarrier` will be tripped and the main process or thread will stop the timer. The timings obtained will be the time it takes for the processes or threads to execute their critical section.

When performing the benchmarks, the timings proved to be extremely volatile and inconsistent, as can be seen in Table 9.1 when considering and comparing the shortest times and longest times. This may be the result of what the operating system scheduler is scheduling at that given time. A solution implemented for this problem is to repeat the benchmarks to get an average time estimate. Therefore each benchmark has been repeated ten times, and the average of the time is recorded as well as the largest and smallest times for the benchmark. To make comparisons the average time is the time that will be used as it gives the most accurate time to work with.

	JCP Time (ms)			Multi-Threading Time(ms)		
Concurrency	Shortest	Longest	Average	Shortest	Longest	Average
1	45	48	46	0	5	1
10	1460	1502	1485	3	27	6
20	2723	3371	3214	5	34	12
30	4166	5234	4984	11	39	18
40	6076	7167	6976	18	53	24
50	9076	9166	9100	25	64	30
60	10960	11016	10989	30	75	37
70	12755	13213	13103	36	76	43
80	14667	15093	14937	43	93	51
90	16412	17052	16826	48	98	55
100	17878	19120	18850	52	105	62

Table 9.1: Timings of Lock Benchamrks.

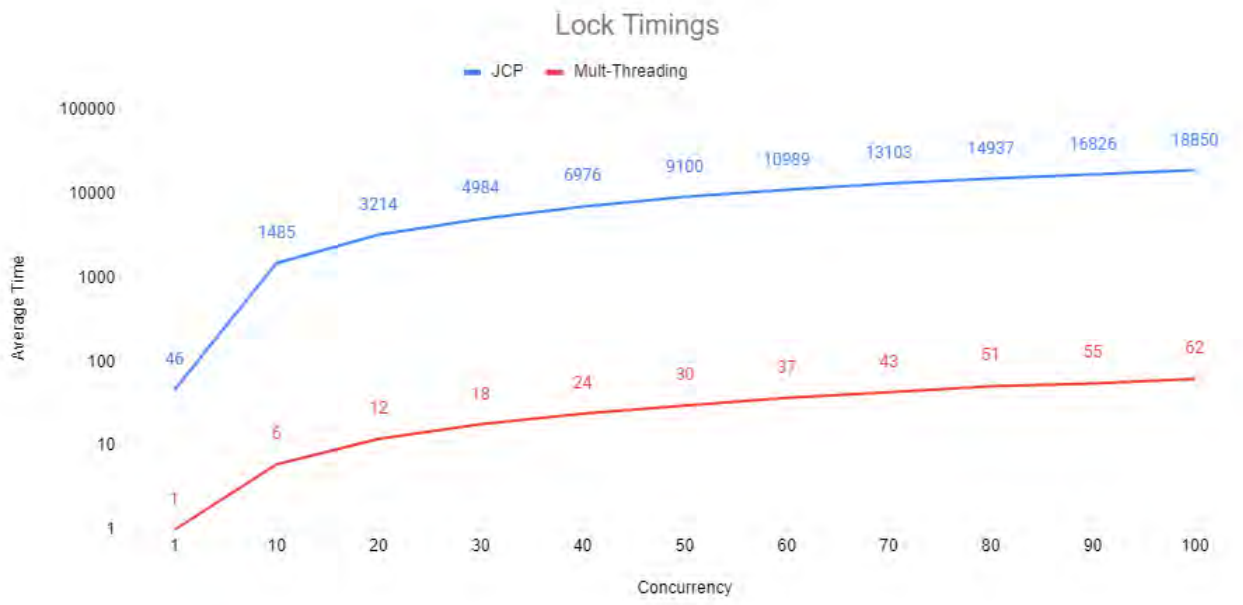


Figure 9.1: JCP Lock Timings vs Thread Lock Timings

When analyzing the timing difference between the two lock implementations, the timing difference between the two locks is significantly large therefore a logarithmic graph is used as opposed to a linear graph (shown in Figure 9.1). From Figure 9.1, and Table 9.1 we can see that the two locks' timings are vastly different, with the JCP `Lock` being significantly slower than the multithreaded `Lock`, as expected. From the graph we can see that both locks handle lock contention in a similar manner. When the number of

processes or threads using the lock increases, the timings of the two lock implementations increases in a similar manner. Looking at how the JCP lock is implemented, the reason for such a major difference in time would primarily be due to the time it takes to access the shared memory the lock is implemented on.

9.2 Data Structures

To benchmark the performance of the data structures, the producer-consumer problem is used. An integer `BlockingQueue` is used in the problem, `BlockingQueueInteger` for the JCP tests and, `java.util.concurrent.ArrayBlockingQueue` for the multi-threading tests. The size of the queues is set to 100 000. The benchmark is run multiple times with each benchmark, N producers and N consumers are created. The producers execute a for-loop (100 000 times). With each iteration of the loop the producers insert the loop counter into the `BlockingQueue`. On the other hand, the consumers also execute a for loop (100 000 times) with each iteration taking a value from the blocking queue and printing their loop counter along with the value obtained separated by a colon. When running the benchmarks the output is piped to a local file.

The initialization and timing of the benchmark are performed similarly to that of the lock, using barriers. The benchmark only times the critical section, inserts into the blocking queue. The timing results of the benchmarks are shown in Table 9.2 and Figure 9.2. As shown in the table, the number of producers and consumers used starts from one and increments in values of one until the benchmark is using ten producers and ten consumers.

The results obtained at low concurrency levels are the expected results, with a multi-threading library executing faster than the JCP library. However, at three producers and three consumers, the timings are not what is expected, and the JCP library starts to perform better than the multithreaded library. This trend continues up to the maximum concurrency the libraries are benchmarked with. Although the two libraries' margin is not big, considering that the threads share their address space and the JCP library has to use a dedicated process and message queues for the `BlockingQueue`, it is a surprise to have the JCP library performing better than the threads library.

	JCP Time (ms)			Multi-Threading Time (ms)		
Concurrency	Shortest	Longest	Average	Shortest	Longest	Average
1	1137	1413	1230	325	431	343
2	1585	2080	1811	1424	1619	1501
3	2087	2481	2363	2343	2837	2500
4	2633	3057	2901	3237	3820	3386
5	3151	3741	3544	4199	5356	4564
6	3877	4426	4220	5371	6270	5611
7	4757	5502	5094	6176	7037	6487
8	5896	6682	6219	7140	7893	7488
9	6407	7685	7319	7975	9112	8359
10	8018	8709	8418	9092	9927	9356

Table 9.2:
Timings of Producer Consumer Benchmark.

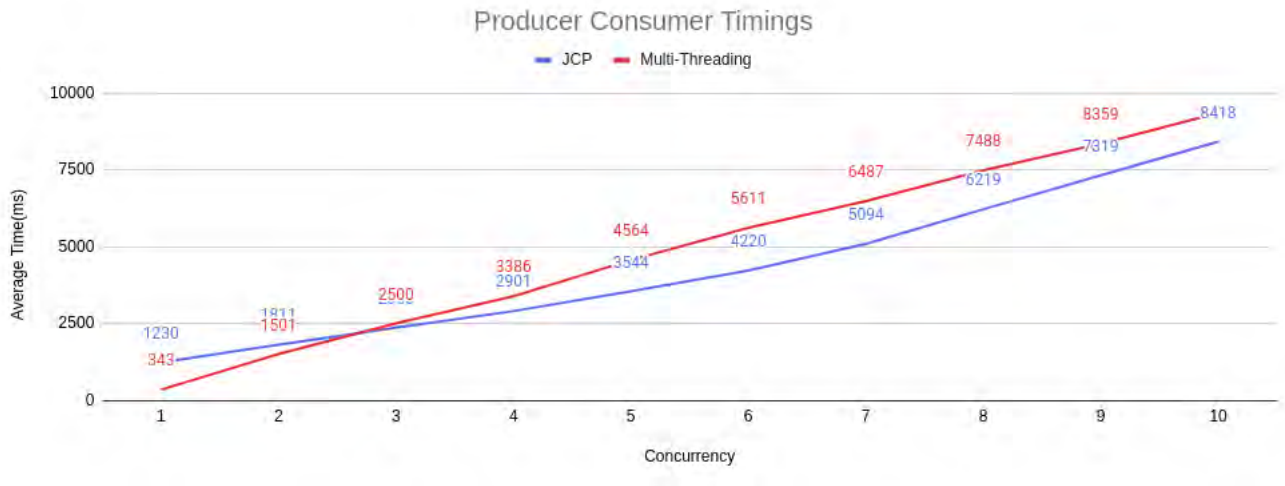


Figure 9.2: Producer-Consumer Timings

The timings of the producer-consumer benchmark were not consistent with what was expected to happen. Therefore analysis was performed on the benchmark to see as to why the JCP library performed better than the multithreaded library at concurrency greater than three. This time the output of the benchmark was no longer getting piped to a file. This gave us the similar results to the previous benchmark (i.e. JCP library still performed better than the multithreaded library for more than three producers/consumers).

Since there was no difference in the results of the two benchmarks, and we still did not know why the JCP library performed better than the multithreaded library the benchmark was performed again. This time the consumers no longer printed the value obtained from the `BlockingQueue`, therefore omitting IO from the benchmarks and purely benchmarking the performance of the `BlockingQueue`. The result of this benchmark are shown in Table 9.3. From these results we can see that the multithreaded library performs better than the JCP library, which is what was expected. Therefore the reason the JCP library performed better than the multithreaded library was due to the printing of the results obtained from the `BlockingQueue`. It seems that the Java libraries may have some thread-level synchronisation for IO operations, which impacts on the performance of this benchmark.

	JCP Time (ms)			Multi-Threading Time (ms)		
Concurrency	Shortest	Longest	Average	Shortest	Longest	Average
1	536	855	660	18	65	28
2	616	917	822	25	114	39
3	845	952	886	41	148	60
4	853	1035	893	67	191	94
5	1003	1159	1080	83	209	114
6	1282	1448	1345	91	213	123
7	1522	1756	1594	97	222	137
8	1748	1873	1802	121	270	182
9	1894	2169	2008	186	295	204
10	2118	2311	2193	189	346	234

Table 9.3:
Timings of Producer Consumer Benchmark (No IO).

9.3 Executor Service

The executor service is benchmarked using a Fibonacci program. There are three types of Fibonacci benchmarks used. Each of the benchmarks measures the performance of the executor frameworks against different workloads.

In the first benchmark, the time it takes to calculate a Fibonacci sequence is measured. With each test there are ten processes or threads calculating the *Nth* number of the

Fibonacci sequence (Fib N). To increase the workload of each test the calculation of each Fibonacci sequence is calculated ten times by the processes or threads. This will result in ten tasks submitted into the executor service for each benchmark, so each process or thread will typically be calculating its own Fib N (assuming the workload is balanced evenly among the threads or processes). For example Fib 10 is calculated ten times by ten processes or threads, Fib 20 is calculated ten times by ten processes or threads, and so forth. N starts from ten and increments in values of ten until N is equal to fifty. N is obtained from the command line arguments. The static functions `newFixedProcessPool`, and `newFixedThreadPool` are used to start the JCP executor service, and multi-threading executor service respectively. An array list of futures is then created to store the futures. The tasks are then submitted in a for loop for execution, and the futures returned are stored in the array created. To obtain the results, the program goes into a for loop and obtains the results in their order of submission.

The time it takes to execute the whole program is measured in milliseconds, as well as the time it takes to execute the critical section. The critical section time is the time from the submission of the tasks to the time it takes to get the results back. The results of the benchmarks are shown in Table 9.4. From the table, we can easily identify that the timings between the two libraries follow a similar pattern. The timings are relatively low for tests where N is low ($N = 10$ to $N = 40$); however, they increase significantly for large values ($N = 50$). The table also contains a timing ratio column to compare the critical section timings of the two libraries. From the ratios we can see that at low values, the multi-threaded library is significantly faster than the sequential library (4.69 times faster). However at relatively high values the two libraries critical section timing are relatively the same, with the JCP library 1.02 times slower than the multithreaded library at $N = 50$.

	JCP Timings (ms)		Multi-Threading Timings (ms)		Timing ratio
N	Critical Section	Whole Program	Critical Section	Whole Program	JCP : multithread
10	183	193	39	42	4.69 : 1
20	153	163	43	47	3.56 : 1
30	178	189	56	58	3.18 : 1
40	946	956	796	799	1.19 : 1
50	96183	96194	94066	94073	1.02 : 1

Table 9.4: Timings of Fibonacci Benchmarks

The second benchmark measures the amount of time it takes to calculate a large Fibonacci ($N = 50$) sequence multiple times. In the benchmarks, the number of times the sequence

is calculated is modified. First, the sequence is calculated once, then ten times, after that incrementing in values of ten up to fifty iterations. The timings of the benchmark are shown in Table 9.5 and Figure 9.3. When analyzing the figure, we notice that both libraries' timings are similar, with the notable difference coming at large values ($N = 50$). The second thing to notice from the figure is that the rate of increase in time is relatively low from one to ten, unlike with the rest of the program. This is not a surprise as the benchmarks use ten processes or threads. This means that Fibonacci one only utilizes one process or thread and leaves the other processes/threads idle; i.e., the system only starts to become fully utilized at ten iterations; therefore, the timings of one iteration are not vastly different from the timings of ten iterations.

	JCP		Threads	
iterations	Critical Section(ms)	Whole Program(ms)	Critical Section(ms)	Whole Program(ms)
1	82848	82863	82504	82510
10	94124	94135	93776	93779
20	190723	190733	186310	186313
30	283464	283477	278992	278992
40	380042	380052	371829	371832
50	483845	483855	464585	464588

Table 9.5: Timings of Fib(50) Benchmarks

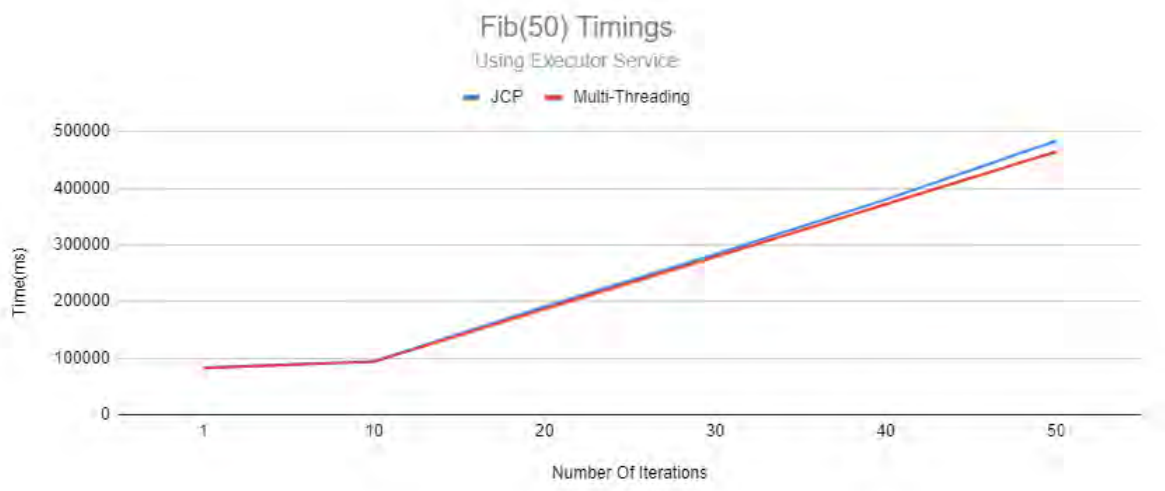


Figure 9.3: Fib(50) Timings

The third benchmark calculates Fibonacci fifty using the `ForkJoinPool`. Both the JCP library and multithreading library are used in this benchmark, with the timings of the two benchmarks coming to 12 441 ms and 11 458 ms, respectively. Again with this benchmark, we realize that although the libraries' timings are similar, the threads library is still more efficient than the JCP library, as expected.

The benchmark's main class is similar to that of the previous benchmark, except only creating a single Fibonacci task. The difference between the tasks comes in the `Fibonacci` class. In the previous benchmark, each Fibonacci sequence was run from start to finish by the same thread/process in a recursive fashion. However, with this benchmark, instead of recursing, another task is created and submitted for execution. This is performed until a set threshold is reached. When the threshold is reached, the rest of the computation is then performed sequentially in a recursive method.

9.4 Library Similarities

One of the design objectives of the JCP library is to make the usability of the JCP library as simple as possible. The approach taken to do this is to closely follow the existing multi-threaded library. As a result the JCP library has implemented the same classes as the multi-threaded library. This will allow programmers who already know how to write concurrent programs using the multi-threaded library to easily use the JCP library. We will now analyse the degree of similarity between the code of the JCP benchmarks and the code of the multi-threading benchmarks. We will also analyse how much code needs to be changed on a program using the multi-threaded library to use the JCP library.

The code we will be analysing is the Fibonacci code used in the benchmarking of the Fork-Join classes. The code of the classes is shown in Appendix A. This code is used because it uses the code of the libraries extensively, although it is still a relatively simple program to analyze. Both benchmarks use two classes, a `FibMain` class, and a `Fibonacci` class. The code of the classes will be compared with each other to see how they differ (i.e. the `FibMain` class of the JCP benchmark will be compared to the `FibMain` class code of the multi-threaded benchmark, and the same for the `Fibonacci` classes).

Comparing the `Fibonacci` classes we realise that both classes implement methods that perform similar functions. First we will compare the constructors of the two classes. The constructors differ in the parameters they accept. Both methods accept the sequence number to calculate (`int x`), and the threshold to stop forking additional tasks and calculate sequentially (`int threshold`). However the JCP benchmark's class takes in three extra parameters for the keys (`sndKey`, `rcvKey`, and `dataKey`). The method also differs in that the JCP benchmark's class is required to call the parent class' constructor (using `super`) in order to pass the keys. The rest of the code of the methods are identical to each other. The two methods also differ in that JCP benchmark's class constructor throws exceptions (`SystemException`, `ClassNotFoundException`, `IOException`).

The second methods implemented by the `Fibonacci` classes we will be analysing are the `compute` method, and the `call` method. These methods, although their method signatures differ (in the name of the method), perform the same function, and the code of the two methods is similar. The difference of the code comes when creating new tasks to fork. Due to the JCP benchmark's class constructor requiring extra parameters, the parameters need to be supplied when creating new tasks. The second difference in the code is with the method used to submit the task for execution to the executor. These method also differ in the exceptions thrown. The multi-threaded benchmark's class throws no exception while the JCP benchmark's class throws an `Exception`.

The last method that is implemented by both classes is the `seqFib`. These methods have the same method signature. In addition these methods use the exact same code, there is no difference in the code of the two methods. With this method we may say that the code is the same and there will be no need to change the code when porting an application from one library to the other.

The `FibMain` class is the second class implemented by both versions of the benchmark. The classes both implement a single method (`main`). The difference in the two classes are the initialisation, and passing of the keys to the `Fibonacci`, and `ForkJoinPool` constructors. The second difference is the passing of the return types' class to the invoke method.

From the analysis of these classes we can realise that the code of the classes is very similar, and if a program using the multi-threaded library needed to change to use the JCP library there would only be minor changes that would need to be done to the code of the applications. The constructor of the class would need to be changed to accept three long values for the keys, and call the constructor of the parent class. With the `compute` method the name of the method would also need to be changed to `call`, and the method used to fork new tasks would need to be changed. In addition the two methods will need to handle exceptions that may be thrown, whether by throwing the exception or handling the exception inside the method.

9.5 Conclusion

In this chapter the performance of the JCP library and the multi-threaded library were analyzed and compared. In addition the similarity of the two libraries is looked at, as well as how the two libraries differ in usage.

When analyzing the performance of the libraries, first, the performance of the synchronization methods is analyzed. The synchronization method that is analyzed was the lock. In this regard, the multi-threading library was considerably more efficient than the multi-processing library, as expected. When analyzing the performance of the data structures, the `BlockingQueue` was the class chosen for analysis. In this group, the results were surprising as the multi-threading library is more efficient when using low numbers of threads/processes. However, as the number of processes/threads used increases, the JCP library starts to become more efficient. The inefficiency of the multithreaded library for this benchmark seems to be related to the behaviour of IO operations in multithreaded Java applications. For the executor service, the threads library is marginally more efficient than the JCP library for both the `ProcessPoolExecutor` and the `ForkJoinPool` classes.

With these benchmarks and results, we realize that barring the synchronization methods, the two libraries' performance is similar when faced with similar workloads. The multi-threading library performs more efficiently than the JCP in most situations, as we would expect. At the same time, the JCP library is also marginally more efficient in some situations, which was due to the IO involved printing the results. Also to note with these two libraries, the more work done by the benchmarks, the more the timing difference between them decreases. This is evident in the timings of the Fibonacci benchmarks: for low-valued Fibonacci sequences the timing ratio between the two libraries is large, but at higher sequences the ratio between the timing results for the two libraries is small.

From these two classes analyzed in Section 9.4, we can realise that the usage of the multi-threaded library and the JCP library is very similar, and a programmer who is already proficient with the multi-threading library may easily adapt and use the JCP library with little further knowledge required. The porting of code from one library to the other is also relatively simple, with a few minor, simple changes to the code. When porting from multi-threading to the JCP library, the main changes required will be the handling of exceptions, and supplying keys to the library to use for the message queues and shared memory. This similarity between the two libraries makes the JCP library a library that will be easy to use for concurrent Java programmers.

Chapter 10

Conclusion

Throughout history, there has been a need for faster computers. The need for faster computers has resulted in the number of cores in a CPU to increase in recent years. Intel's entry level processors (Intel Celeron, and Intel Pentium) produced today (2020) have at least two cores per processor [27]. However, programs do not automatically run faster on these extra cores; programs need to be programmed to utilize the extra cores. To do this, programs need to run their code concurrently through multiprocessing or multithreading. However, writing concurrent programs is a cumbersome task with complexities not inherent in sequential programming becoming apparent in concurrent programming (race conditions, deadlock, starvation, work balance etc. are some of the concurrent complexities). Arguably these complexities make concurrent programming more difficult than sequential programming.

The Java programming language provides extensive support for concurrent programming through threads. The support Java provides for concurrent threads is embedded in the language itself. Java also provide threads support through Java class libraries, primarily the `java.util.concurrent` thread library. Although Java has rich support for concurrent threads, it has limited support for concurrent processes. This makes Java limiting as process concurrency provides some advantages over thread concurrency.

10.1 Research Summary

This research's intention was to develop a library that will provide support for concurrent processes. The support provided by the library should match the current support provided

by the current `java.util.concurrent` (multithreading) library. The library that has been built to perform these functions is the `za.co.jcp` (JCP) library.

The structure of the JCP library closely follows that of the multithreading library. The JCP library offers the same classes with the same methods as the multithreading library. This allows for the libraries to be interchangeable by concurrent applications. This will allow for applications already using the multithreading library to easily change to the JCP library if they need to use concurrent processes rather than concurrent threads. The similarity and difference between the two libraries is outlined in Section 9.4, which also highlights the required code changes when porting from one library to the other. The similarity between the two libraries will also allow a programmer who is already familiar with the multithreading library to be able to use the JCP library without any additional knowledge required. These advantages will make the usability of the library simple and easy for programmers.

10.2 Project Objectives Achievement

The objective of this project were to develop a system that natively supports the usage of multiple processes in a Java application. The detailed, concise objective of the project is discussed in Section 1.2. From the project's objective, we may discuss the success of the project.

1. **Support for process shared variables and atomic variables:**
   boolean, int, double, long, char, and String process shared variables for multiprocess concurrent programs have been implemented and tested in the library (Section 4.1). These variables are able to be shared and used by multiple processes running in the same system. Process shared atomic variables have also been implemented for Java applications to use (Section 4.2). With the process shared variables and atomic variables implemented, it means that this project's sub-objective has been met.

2. **Support for process synchronisation:**
   Multiple synchronization methods have been developed and tested to support process synchronization (Chapter 5). The synchronisation methods in the library have been implemented through the `Lock`, `Mutex`, `ReadWriteLock`, `Semaphore`, `Condition`, `CountDownLatch`, and `CyclicBarrier` classes. These mechanisms allow processes to synchronize their actions and prevent some of the bugs apparent in

concurrent programming. With these synchronization methods implemented, this project's sub-objective has been met.

3. **Support for process shared data structures:**
   Multiple data structures have been developed and implemented in the library (Chapter 6). These include `Array`, `List`, `ConcurrentHashMap`, and `BlockingQueue` classes. These data structures are able to be shared and used by multiple Java processes running on the system. In addition these data structures are multiprocess safe. With these data structures implemented, this project's sub-objective has been met.

4. **Task scheduling framework supporting scheduling of tasks through processes:**
   The executor framework has been implemented in the library (Chapter 7). The executor framework allows for tasks to be submitted and scheduled for execution in Java processes. The functionality provided by the executors matches the functionality required by the project's sub-objective; this means that the project's sub-objective has been met.

5. **Performance and similarity comparison between the current `java.util.concurrent` package and the new `za.co.jcp` package:**
   Chapter 9 compares the performance of the implemented library using processes against the performance of `java.util.concurrent` using threads. In addition code using the two different libraries is analyzed to identify the similarities and difference between the codes. As the comparison between the two libraries has been performed, this project's sub-objective has been met.

All of the project's sub-objectives have been met. This implies that the project's objective has been met, and the project has been a success and may be used for developing Java multi-process systems. The system's performance benchmarking revealed that the library's performance is not vastly different from the `Java.util.concurrent` library as initially thought it would be. This is encouraging as an application using the JCP library will not be significantly slower than an application using `Java.util.concurrent` library, and as previously explained in Section 1.1 the application would have gained modularity and the advantage of: if one task of the application crashed or unexpectedly dies, the whole system does not have to die, the process may simply be restarted.

10.3 Future Work

For future work, there are improvements that could be made to the library, which would help improve the usability and performance of the library:

- Currently, the library does not support the use of multiple threads with the library. The usage of threads in the library could result in undefined behavior as the library uses the process id to identify and differentiate the different processes using the library. If multiple threads sharing the same process id use the library, the library will not be able to differentiate them, and this could result in errors. The integration of multiple threads into the system could be implemented in future works.
- In the library, only four data structures are natively supported. The library could be improved to support further data structures. This would allow the library to be more versatile and support a wider array of problems. The data structures that could be implemented in the library include binary trees, graphs, and stacks. This would be an extension beyond the scope of this project.
- For future work, the Java compiler could be modified to provide language-level support for multiprocessing. Functions similar to the synchronized block for threads would be beneficial for the library. This would allow for even easier access and usage of the library by programmers.
- The JCP library is currently only supported for Linux operating systems. Other operating systems will not be able to use the library as it uses the IPC provided by Linux. It would be beneficial to have the library supported in other mainstream operating systems (Windows, and macOS).

References

[1] Giuseppe Agapito, Pietro Guzzi, and Mario Cannataro. Parallel and distributed association rule mining in life science: A novel parallel algorithm to mine genomics data. *Information Sciences*, 453, 07 2018.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating systems: Three easy pieces, vol. 151. *Arpaci-Dusseau Books Wisconsin*, 2014.

[3] Ravishekhar Banger and Koushik Bhattacharyya. *OpenCL programming by example*. Packt Publishing Ltd, 2013.

[4] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Rajkumar Buyya, Christian Vecchiola, and S. Thamarai Selvi. *Mastering Cloud Computing: Foundations and Applications Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[6] Stephen Cass. Multicore processors create software headaches. *Technology Review*, 113(3):74–75, 2010.

[7] Leigh Anne Clevenger, Hugh Eng, Kevin Khan, Javid Maghsoudi, and Mantie Reid. Parallel computing hardware and software architectures for high performance computing. *Proceedings of Student-Faculty Research Day,(White Plains, New York), Seidenberg School of CSIS, Pace University*, May 2015.

[8] Göhringer Diana, Perschke Thomas, Hübner Michael, and Juergen Becker. A taxonomy of reconfigurable single-/multiprocessor systems-on-chip. *International Journal of Reconfigurable Computing*, 2009:1–2, January 2009.

[9] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[10] Ian Foster. *Languages for Parallel Processing*, pages 92–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[11] Thomas L Friedman. Moore's law turns 50. *The New York Times*, 13, 2015.

[12] Jeff Friesen. *Java Threads and the Concurrency Utilities*. Apress, USA, 1st edition, 2015.

[13] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, volume 99 of *OSDI '99*, pages 87–100, USA, 1999. USENIX Association.

[14] Jose Garrido. Process synchronization. In *Performance Modeling of Operating Systems Using Object-Oriented Simulation: A Practical Introduction*, pages 111–146. Springer US, Boston, MA, 2000.

[15] Warren W. Gay. *Linux Socket Programming: By Example*. Que Corp., USA, 2000.

[16] Pawel Gepner and Michal Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13, 2006.

[17] Tejashree Ghadge and Prasad Renukdas. Spontaneous detection and termination of livelocks on concurrent programs. *International Journal of Emerging Technologies and Innovative Research*, 1(7):609–611, 2014.

[18] Brian Goetz. Concurrency: Past and present. Available at http://jaoo.dk/dl/qcon-london-2008/slides/BrianGoetz_HistoryOfConcurrency.pdf (2021/06/18), 2006.

[19] Brian Goetz, Tim Peierls, Doug Lea, Joshua Bloch, Joseph Bowbeer, and David Holmes. *Java concurrency in practice*. Addison-Wesley Professional, 2005.

[20] Javier Fernández González. *Mastering Concurrency Programming with Java 8*. Packt Publishing Ltd, 2016.

[21] John Shapley Gray. *Interprocess Communications in Linux*. Prentice Hall Professional Technical Reference, 2003.

[22] Kate Green. The trouble with multi-core computers. Available at https://www.technologyreview.com/s/406760/the-trouble-with-multi-core-computers/ (2019/11/1), 2006.

[23] William Grosso and Robert Eckstein. *Java RMI*. O'Reilly & Associates, Inc., USA, 1st edition, 2001.

[24] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a java-like language. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 347–362, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[26] Roger Henriksson. Resource allocation graphs. Available at https://fileadmin.cs.lth.se/cs/Education/EDA040/resourcegraphs.pdf (2020/10/15).

[27] IntelC. Intel pentium silver and celeron processors. Available at https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/pentium-silver-and-celeron-processors-product-brief.pdf (2021/12/01).

[28] Limi Kalita. Socket programming. *International Journal of Computer Science and Information Technologies*, 5(3):4802–4807, 2014.

[29] Eric Koskinen and Maurice Herlihy. Dreadlocks: Efficient deadlock detection. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 297–303, New York, NY, USA, 2008. Association for Computing Machinery.

[30] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, 6th edition, 2012.

[31] Kamran Latif. Parallelism via concurrency at multiple levels. *CoRR (Computing Research Repository)*, abs/1406.0184, 2014.

[32] Doug Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 12 2005.

[33] Bil Lewis and Daniel J Berg. Pthreads primer: A guide to multithreaded programming. *Sun Microsystems Inc*, 1996.

[34] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1999.

[35] Fred Long, Dhruv Mohindra, Robert Seacord, and David Svoboda. Java concurrency guidelines. Technical report, Air Force Electronic Systems Center Hanscom AFB MA, 2010.

[36] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional, 2011.

[37] Nihar R Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 5(3es):2–es, 1999.

[38] Ami Marowka. A study of the usability of multicore threading tools. *International Journal of Software Engineering and Its Applications*, 4(3), July 2010.

[39] Eric Larsen McCorkle. *Realizing Concurrent Functional Programming Languages*. PhD thesis, Brown University, 2008.

[40] Paul E McKenney, Vaddagiri Srivatsa, and Gautham R Shenoy. Read/write lock with reduced reader lock sampling overhead in absence of writer lock acquisition, April 26 2011. US Patent 7,934,062.

[41] Sudhir K. Meesala, Pabitra Khilar M., and Ajeet K. Shrivastava. Multiple Instruction Multiple Data (MIMD) Implementation on Clusters of Terminals. *International Journal of Science and Research*, 5:1652–1658, 01 2016.

[42] Ivan Merelli, Ettore Mosca, and Luciano Milanesi. *Parallel Computing, Data Parallelism*, pages 1624–1624. Springer New York, New York, NY, 2013.

[43] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM letters on programming languages and systems*, 1(1):74–88, March 1992.

[44] Scott Oaks and Henry Wong. *Java Threads: Understanding and Mastering Concurrent Programming*. O'Reilly Media, Inc., 3rd edition, 2004.

[45] Oracle. Java SE Documentation. Available at https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html (2020/11/10).

[46] Oracle. Java SE Documentation. Available at https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html (2019/11/10).

[47] Oracle. Java documentation: Concurrency. Available at https://docs.oracle.com/javase/tutorial/essential/concurrency/ (2020/02/15), 2017.

[48] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: Preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, pages 67–72, New York, NY, USA, 2006. Association for Computing Machinery.

[49] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[50] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *IARCS (Indian Association for Research in Computing Science) Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, APLAS '08, page 138, Berlin, Heidelberg, 2008. Springer-Verlag.

[51] Portable Operating System Interface (POSIX). pthread_cond_wait(3) - Linux man page. Available at https://linux.die.net/man/3/pthread_cond_wait (2020/02/15).

[52] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomput.*, 7(1–2):9–50, May 1993.

[53] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Publishing Company, Incorporated, 2012.

[54] Anil Sethi and Himanshu Kushwah. Multicore processor technology-advantages and challenges. *International Journal of Research in Engineering and Technology*, 4(09):87–89, 2015.

[55] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts with Java*. Wiley Publishing, 8th edition, 2009.

[56] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[57] Gürkan Solmaz, Rouhollah Rahmatizadeh, and Mohammad Ahmadian. A study on SIMD architecture. Available at http://cs.ucf.edu/~ahmadian/pubs/SIMD.pdf (2020/01/15), 2013.

[58] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan &; Claypool Publishers, 1st edition, 2011.

[59] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, USA, 7th edition, 2011.

[60] Martin Sysel. A comparison of processes and threads creation. In Radek Silhavy, Petr Silhavy, and Zdenka Prokopova, editors, *Software Engineering Perspectives in Intelligent Systems*, pages 990–997, Cham, 2020. Springer International Publishing.

[61] Andrew Stuart Tanenbaum and Robbert Van Renesse. *A critique of the remote procedure call paradigm*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.

[62] Riccardo Terrell. *Concurrency in .NET: Modern Patterns of Concurrent and Parallel Programming*. Manning Publications Co., USA, 1st edition, 2018.

[63] TIOBE. TIOBE index for April 2019. Available at https://www.tiobe.com/tiobe-index/ (2019/04/01), 2019.

[64] Balaji Venu. Multi-core processors - An overview. *ArXiv*, abs/1110.3535, 2011.

[65] Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in ERLANG (2nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[66] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 176–188, New York, NY, USA, 1991. Association for Computing Machinery.

[67] KC Wang. Process Management in Unix/Linux. In *Systems Programming in Unix/Linux*, pages 101–140. Springer International Publishing, 2018.

[68] P.H. Welch. Parallel hardware and parallel software: a reconciliation. In *Proceedings of the ZEUS'95 (Centres for European Supercomputing) & NTUG'95 (Nordic Transputer User Group) Conference, Linkoping*, pages 287–301, 1995.

[69] George Wells. Interprocess communication in java. In *PDPTA*, pages 407–413, 2009.

[70] George C. Wells. Extending Java's communication mechanisms for multicore processors. In *8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)*. Citeseer, 2010.

[71] George C Wells. *Advanced Programming*. Rhodes University, 2011.

[72] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java™ System. In *Proceedings of the 2nd Conference on USENIX (The Advanced Computing Systems Association) Conference on Object-Oriented Technologies (COOTS) - Volume 2*, COOTS'96, page 17, USA, 1996. USENIX Association.

Appendix A

Fork-Join Fibonacci Code

JCP Fibonnaci Class

```
1  public class Fibonacci extends ForkJoinTask<Long> {
2
3      private static final long serialVersionUID = -7250133288355814240L;
4      private int x;
5      private final int threshold;
6
7      public Fibonacci(int x, int threshold, long sndKey, long rcvKey, long
           dataKey) throws SystemException,
8      ClassNotFoundException, IOException {
9          super(sndKey, rcvKey, dataKey);
10         this.x = x;
11         this.threshold = threshold;
12     }
13
14     @Override
15     public Long call() throws Exception, SystemException {
16         if(x<threshold)
17             return seqFib(x);
18         Fibonacci task = new Fibonacci(x-1, threshold, sndKey, rcvKey,
               dataKey);
19         Fibonacci task2 = new Fibonacci(x-2, threshold, sndKey, rcvKey,
               dataKey);
20         invoke(task2, Long.class);
21         Long val = task.call() +  task2.join();
22         return val;
23     }
24
```

```
25      private Long seqFib(int v) {
26          if(v==1)
27              return 1L;
28          if(v==0)
29              return 0L;
30          return seqFib(v-1) + seqFib(v-2);
31      }
32  }
```

Listing A.1: JCP Fibonacci class

Thread Fibonnaci Class

```
1  public class Fibonacci extends RecursiveTask<Long>{
2
3      private static final long serialVersionUID = -6095524456400544738L;
4      private final int x;
5      private final int threshold;
6
7      public Fibonacci(int x, int threshold){
8          this.x = x;
9          this.threshold = threshold;
10     }
11
12     @Override
13     public Long compute() {
14         if(x<threshold)
15             return seqFib(x);
16         Fibonacci task = new Fibonacci(x-1, threshold);
17         Fibonacci task2 = new Fibonacci(x-2, threshold);
18         task2.fork();
19         Long val = task.compute() + task2.join();
20         return val;
21     }
22
23     private Long seqFib(int v) {
24         if(v==1)
25             return 1L;
26         if(v==0)
27             return 0L;
28         return seqFib(v-1) + seqFib(v-2);
29     }
30
31 }
```

Listing A.2: Multi-threading Fibonacci class

JCP FibMain Class

```
1  public class FibMain {
2
3      public static void main(String[] args) {
4          int FIB_NUMBER = Integer.valueOf(args[0]);
5          int THRESHOLD = Integer.valueOf(args[1]);
6          ForkJoinPool executor = null;
7          long sndKey = 1300;
8          long rcvKey = 1301;
9          long dataKey = 1302;
10         Class<Long> classType = Long.class;
11         try {
12             System.out.println("**********JCP***********");
13             long startWhole = System.currentTimeMillis();
14             executor = ForkJoinPool.createDefaultPool(sndKey, rcvKey,
                   dataKey);
15             Fibonacci fib = new Fibonacci(FIB_NUMBER, THRESHOLD, sndKey,
                   rcvKey, dataKey);
16             long startCritical = System.currentTimeMillis();
17             executor.invoke(fib, classType);
18             Long result = fib.get();
19             System.out.println("FIB " + FIB_NUMBER + ": " + result);
20             long endCritical = System.currentTimeMillis();
21             executor.shutdown();
22             long endWhole = System.currentTimeMillis();
23
24             String output = "\n*************" + FIB_NUMBER + "
                   ********************"
25             + "\nCritical Section Time:" + (endCritical - startCritical) +
                   "ms"
26             + "\nWhole program Time: " + (endWhole - startWhole) + "ms";
27             BufferedWriter writer;
28             try {
29                 writer = new BufferedWriter(new FileWriter(
30                     "benchmarks/za/co/jcp/forkfib/JCPResults", true));
31                 writer.write(output);
32                 writer.close();
33             } catch (IOException e) {
34                 e.printStackTrace();
35             }
36         } catch (Exception e) {
37             e.printStackTrace();
38         }
39
```

```
40          }
41  }
```

Listing A.3: JCP FibMain Class

Thread FibMain Class

```
1  public class FibMain {
2
3      public static void main(String[] args) {
4          int FIB_NUMBER = Integer.valueOf(args[0]);
5          int THRESHOLD = Integer.valueOf(args[1]);
6          ForkJoinPool executor = null;
7          try {
8              System.out.println("**********Threads***********");
9              long startWhole = System.currentTimeMillis();
10             executor = ForkJoinPool.commonPool();
11             Fibonacci fib = new Fibonacci(FIB_NUMBER,THRESHOLD);
12             long startCritical = System.currentTimeMillis();
13             executor.invoke(fib);
14             Long result = fib.get();
15             System.out.println("Fib " + FIB_NUMBER + ":" + result);
16             long endCritical = System.currentTimeMillis();
17             executor.shutdown();
18             long endWhole = System.currentTimeMillis();
19
20             String output = "\n*************" + FIB_NUMBER + "
                   **********"
21             + "\nCritical Section Time:" + (endCritical - startCritical) +
                   "ms"
22             + "\nWhole program Time: " + (endWhole - startWhole) + "ms";
23             BufferedWriter writer;
24             try {
25                 writer = new BufferedWriter(new FileWriter(
26                     "benchmarks/za/co/jcp/forkfib/ThreadResults", true));
27                 writer.write(output);
28                 writer.close();
29             } catch (IOException e) {
30                 e.printStackTrace();
31             }
32         } catch (Exception e) {
33             e.printStackTrace();
34         }
35     }
36 }
```

Listing A.4: Multi-threading FibMain