BACHOLER OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2202

JAVA PROGRAMMING

Unit 9

Multithreading

Table of Contents

1. INTRODUCTION

Java is listed as one of the top ten best products of 1995 by the Time Magazine. Java is a highly preferred language for all the enthusiasts of programming, internet search, e-business solutions, application development, and many more. The 'Green Team' responsible for creating such a widely-used programming language involved James Gosling, the father of Java, Michael Sheridon, and Patrick Naughton. They began working on Java development in 1991 for small, embedded system language like set-top boxes, televisions, etc.

The creators kept the following principles in mind while building the Java programming:

- Simple
- Robust
- Platform-independent
- Portable
- Secured
- Speed or High Performance
- Multithreaded
- Dynamic
- Object-Oriented
- Architecture Neutral

The 1st version of Java was then released in 1996, after which many advancements in software technologies and solutions arose, leading to a range of versions that carried different features. These enhanced the value of the above-mentioned principles, and currently, Java is being used in enterprise applications, Windows, and web applications.

STUDY NOTE

C and C++ programming languages are currently able to support multithreading in their latest version: C11 and C++11.

Amongst the various principles on which Java was built, we'll be exploring one: multithreading. When there are multiple requests to execute a particular process or multiple tasks that are to be performed by the said process, multithreading acts as the special aid. The fast-moving world demands faster results to function smoothly, and threading helps in

making that efficiency possible. It allows the execution of multiple concurrent tasks by a single process by distributing the task requirements efficiently to the process.

This process of threading tasks is beneficial in single-processor systems to let the long-running tasks be placed in the background and allow the user input to be processed without interference. So, multiple processing comes into light when it comes to multithreading, and it involves two separate concepts: parallel and concurrent processing.

Parallel multiprocessing makes the system take care of more than one task or 'thread' (an autonomous sequence of instruction. It can run parallel to other threads belonging to the same process). Whereas, Concurrent multiprocessing makes the system handle only one thread at a time. Still, the system moves between multiple threads in a fast-paced manner creating minute yet significant efficiencies.

Despite the difference in conceptual terms, the user is still perceived to be parallel due to the computer system's efficiency in maintaining so. These multiprocessing concepts apply to CPUs where multiple CPUs are added to increase the speed of processing. In Multi-threading, the process adds more threads.

Switching between threads is simpler than switching between entire processes, which collectively speeds up the execution process. Whether it is the GUI of a game or a complex business solution, multithreading speeds up the Java application and allows you to multi-task.

Java evolution also brought many updates in the multithreading aspect: in JDK 1.5, many utilities relating to multiple concurrencies like semaphore, mutex, barrier, latches, etc., were included. We'll look more into Multithreading in further topics

1.1 Learning Objectives

After studying this chapter, you will be able to:

- *Define multithreading and lifecycle of thread along with its types*
- *Explain how to create and start a thread*
- *Understand how a thread works and its Synchronization and define inter-thread communication and how it works*

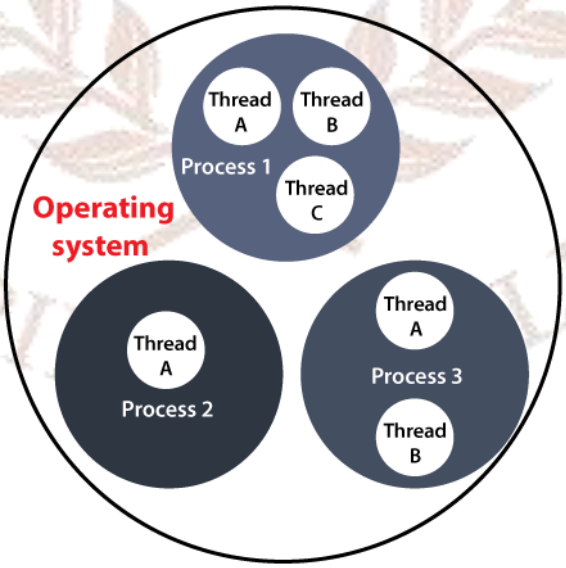
2. THE THREAD CONCEPT

Consider a game where the following tasks need to be performed together simultaneously. For example, calculation of the highest score and updating the leader board are done together.

As in the simple example mentioned above, handling multiple tasks, especially by a computer, was a nightmare in the earlier days, until the thread concept came into existence. This concept helped the system run multiple tasks parallels. It involves using components called threads considered to be the smallest part of the processing unit with their own defined independent path of execution.

Threads are light-weighted processes that do not interfere with or affect the main program and can be executed independently. This helps get more work done (task execution) while enhancing the speed of processing and system efficiency.

This property of the thread is also an advantage to the error corrections and clarifications as their independence makes this step easier and does not hinder the main program's processing. These threads share a memory space and also have their variables, stack, and program counter.



***Source**: javatpoint*

Fig 1: Threads within an OS

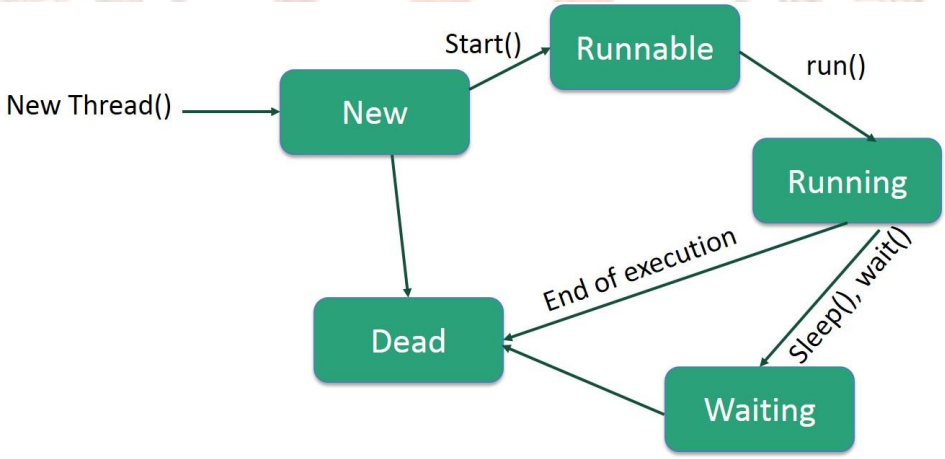
2.1 Life Cycle Of A Thread

Through the execution process of a thread, it is said to exist in various stages:

- **Ready to run stage** - when a thread is ready to be processed by the CPU or, in other words, when it gets CPU time.
- **Running stage** - when the thread is being executed.
- **The suspended stage** - indicate temporary inactivation of the thread's execution due to the CPU tending to another thread. It also sometimes indicates that the thread is in execution.
- **The blocked stage** - when the thread has run out of resources for its execution and is waiting for those resources.
- **Terminated stage** - when the thread execution is halted because of unusual erroneous events, or the thread has entirely executed by the program.

STUDY NOTE

The default priority each thread receives initially is NORM_PRIORITY (constant = 5). But the order of execution of the prioritized threads depends on the platform and is not based on the priority constant values.



***Source**: tutorialspoint*

Fig 2: Threads within an OS

The advancement in electronic technologies like silicon chips etc., pushed the entire world of computers into a new digital era. Therefore, high level and efficient CPU systems have evolved. CPUs are now equipped with multiple cores to reduce the response time. Multiple

thread usage is highly recommended to utilize these multi-core CPUs better to exploit the enormous computing power of Java. This reduces sequential processing and improves multi-taking.

2.2 Types Of Threads

All Java threads are prioritized. The priorities are in the range of:
MIN_PRIORITY (constant =1)
MAX_PRIORITY (constant = 10).

There are two types of threads in Java:

- Daemon thread
- Non-Daemon or User Thread

Daemon Threads are assigned with low priority, and they usually handle the supporting background tasks, whereas user threads have high-priority threads that run in the foreground. User threads or main threads handle more complex and complicated tasks. When a JVM (Java Virtual Machine) is handling these threads, it waits for the active and running user threads to completer their execution (high priority). In contrast, the daemon threads are not waited upon and are forced to terminate once the user threads are executed.

The Java application for concurrent execution creates user threads, while the JVM creates the daemon threads to run background tasks. Therefore, daemon threads are found to be dependent on the running and completion of the user threads.

There are ways to alter the Java code and convert a user thread to a daemon thread using the setDaemon(Boolean)method, where you can set the Boolean value to True or False.

Self-Assessment Questions - 1

1. _____ is the father of Java.
2. Java was released in 1999? [True/ False]
3. _____ principle of Java aids in multitasking.
4. _____ are the smallest units of processing.
5. Threads are heavy and difficult to process. [True/ False]
6. When a thread is inactive it exists in the _____.
   a. Running
   b. Blocked
   c. Suspended
   d. Terminated
7. _____ are the high priority threads. They are also known as _____.
8. _____ method is used to convert a user thread to a daemon thread.

3. CREATING AND STARTING A THREAD

Since we understand what a thread is and how it facilitates multithreading in Java, we move onto creating a thread ourselves. This can be done in two ways:

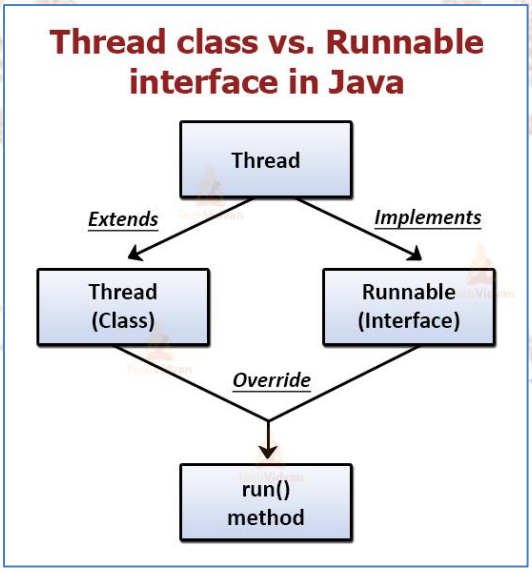
- implementing the **Runnable Interface** (or)
- extending Thread Classes.

What are Thread Classes? (java.lang.Thread)

These provide several methods and constructors that assist in operation performance using threads. Some of the constructors are

- Thread()
- Thread(Runnable target)
- Thread(Runnable, String name)
- Thread(ThreadGroup group, Runnable target, String name)
- Thread(ThreadGroup group, Runnable target, String name, long stackSize)

Using these constructors, the thread class extends the Object class, which implements the Runnable interface, creating the thread.



***Source**: techvidvan*

Fig 3: Creating and starting a thread

3.1 Using Runnable Interface

This interface provides the run() method to perform the thread action. Within this code command, a thread begins its lifetime.

- **Step 1:** Implement the run() method from the Runnable interface. This allows the formation of an entry point for the thread. Here we apply business logic to customize the method as needed for the enterprise solution.

  public void run( )

- **Step 2:** Use the Thread class constructor to instantiate an object

  Thread(Runnable threadObj, String threadName);

  The **threadObj** is the object that will implement the Runnable interface and is named using **threadName**.

- **Step 3:** Now, we move on to the calling of the created object, which is done using the start() method. This piece of code executes a call to the run() by creating a new callstack. When this code is read, the thread shifts from the New to the Runnable stage, thereby calling upon the run()

  void start();

Program: Code to create and start running a new thread using Runnable implementation

```
package unext.java_examples;

public class Main implements Runnable {
         public static void main(String[] args) {
           Main obj = new Main();
           Thread thread = new Thread(obj);
           thread.start();
           System.out.println("This code is outside of the thread");
         }
         public void run() {
           System.out.println("This code is running in a thread");
         }
       }
```

Output:

```
This code is outside of the thread
This code is running in a thread
```

3.2 Extending Thread Class Method

Here there are only 2 simple steps:

- **Step 1:** Override run() method in the Thread class to allow the thread to enter the execution stage.
- **Step 2:** Call its **start()** method

Program: Code to create and start running a new thread using Thread class.

```
package unext.java_examples;

public class Main extends Thread {
        public static void main(String[] args) {
          Main thread = new Main();
          thread.start();
          System.out.println("This code is outside of the thread");
        }
        public void run() {
          System.out.println("This code is running in a thread");
        }
      }
```

Output:

```
This code is outside of the thread
This code is running in a thread
```

Main difference between the 2 methods:

- When the Thread class is extended, unique objects are created for each thread, and further extensions of any other classes are not allowed. This is because Java does not allow more than one class, leading to loss of inheritance benefits.
- When Runnable is implemented, space is created for the instance or object, and this is shared by the threads, allowing the extension of any class that isn't required.

When multithreading used in a code that assigns a counter variable for every thread access, we can observe that only one object or instance is created in the Runnable interface, which

is shredded amongst different threads. Therefore, the counter incrementation is done for each and every point of access.

In the Extend Class method, a new object is created for each thread accessed. Thereby the counter resets and does not increment. This means that different memory is allocated for each object. Due to this outcome, the runnable implementation approach is suggested over the thread class extension. If a class is implementing the runnable interface, then your class can extend another class.

Self-Assessment Questions - 2

9. There are 5 ways to create and run a thread. [True/ False]
10. The components of a thread class called _____.
    a. Objects
    b. units
    c. constructors
    d. none of the above
11. The _____ method is always targeted for overriding.
12. Identify the calling function that calls the execution of run() method?
    a. main()
    b. start()
    c. Thread()
    d. none of the above
13. When Thread class extension method is used, unique _____ are created and they consume _____.
14. Thread Class Extension method is the more preferable method for thread creation and running. [True/False]
15. _____ benefits are lost in thread class extension.
16. A thread can be created by _____.

4. ADVANCED CONCEPTS OF MULTITHREADING

Since we have now covered the basics of Multithreading and a description of the thread concept, we can now delve a bit deeper into the subject. We will now look at the key methods to help navigate the thread while writing the multithreaded application.

Using some of these methods, we try to tackle the problems faced during multithreading via Synchronization and inter-thread communication. These concepts are discussed in detail below.

4.1 Key Methods of A Thread

We mentioned few 'methods' in the previous segment, like run() and start(). What are they? Methods are blocks or segments of code that are executed only when they are called. They are functional in terms of adding data to them; pass values called parameters into these methods.

They are also known as functions (as in C or C++). Calling a method involves the methods' name followed by a parenthesis. Further activation of this method is by calling, after which it either returns a value or void.

There are many such methods involved in threading that are used to call a thread class object. Examples of some of them are:

- **String getName()** – gets the name of the thread that is running in the format of a string of characters
- **void start()** – Initiates a new thread of execution by calling upon a run() method
- **void run()** – Marks the entry point of a thread, thread execution begins here
- **void sleep(int sleeptime)** – suspends the thread for a certain period, shifts the thread into the Suspended stage. The sleep time is mentioned in the arguments
- **void yield()** – Temporary inactivation of a currently running thread and allowing the other threads to get executed
- **void join()** – Queuing up; makes the current thread await its execution till the calling thread is not done
- **Boolean isAlive()** – Checks whether a thread is alive (active) or dead (terminated)

- **getPriority()** – Returns the priority of the thread
- **join()** – halts the current thread unit the previously called thread is terminated.
- **currentThread()** – Returns the reference to the thread that is currently being executed.
- **setPriority(int newPriority)** – Changes the priority of the mentioned or defined thread.
- **interrupted()** – Tests if the current thread has been interrupted.
- **getState()** – Returns the state of the thread.

4.2 Thread Synchronization

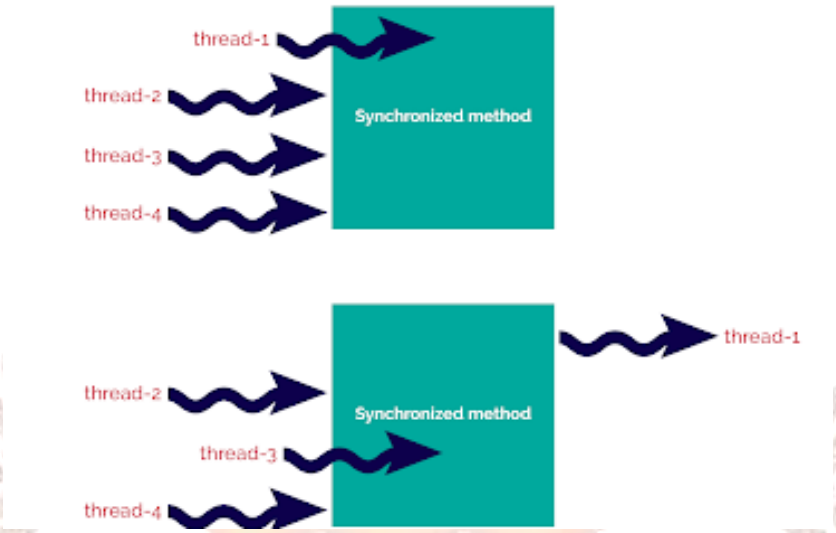
Multithreading is highly useful and makes multi-tasking easier for Java applications, But the more the number of threads, the more complicated the algorithm becomes. When we try to start multiple threads in one program, interference issues arise because the threads try and access a common, shared resource. For example, If the threads are supposed to write data into a file, multiple threads accessing the same file can lead to data override, leading to loss of information. The memory is also being constantly rewritten with multiple access resulting in errors.

Therefore, the developers should ensure these errors don't occur to the best extent possible by controlling how the threads access these shared resources. This control is termed Synchronization. This avoids the problems caused by thread interference and consistency issues.

This synchronization process involves allowing only one thread to access a resource at a particular point in time. And to do so, the concept of monitors was introduced. The objects in Java are all associated individually with a monitor, and only threads can lock or unlock these monitors.

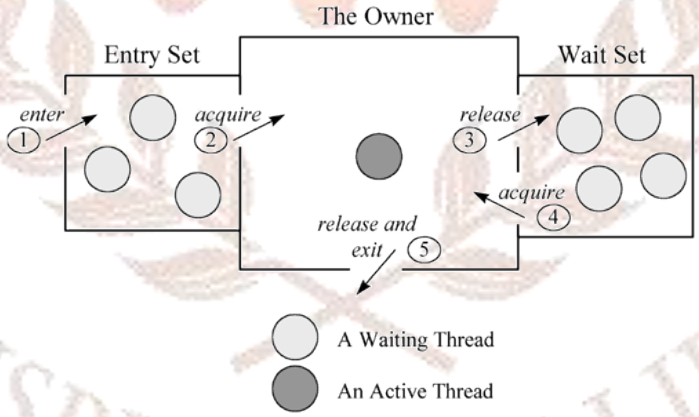
Data can enter these monitors, acquire these monitors (enter a compartment of the monitor), own the monitor (occupies a room of the monitor), releasing the monitor (leaves the compartment it occupied), and exits the monitor. The monitor makes sure that one thread only accesses it using the monitor regions (code bits). Threads enter and exit these monitors

in an organized manner, following all the above steps. If another thread arrives at the monitor entrance, it is made to wait until the currently executed thread is completely done.



***Source**: btechsmartclass*

Fig 5: Synchronization of threads



***Source**: artima.com*

Fig 6: A Java Monitor

These monitors take part in 2 kinds of thread synchronization methods:

- **Mutual exclusion** - This method reduces the interference between multiple threads and allows them to work independently on shared data using object locks. Monitor regions mentioned above are at work here.
- **Cooperation** – Here, the threads are allowed to work together to complete a common task employing wait and notify methods of the class Object. This type of Synchronization is essential when dealing with threads that process the information required by another thread. The tasks associated with these threads are interdependent and aim towards a common goal.

The syntax for the synchronized statement is

synchronized(objectidentifier) {
// Access shared variables and other shared resources
}

The object identifier denotes the object that is locked in the monitor represented by the synchronized statement. The threads can only open these locks and access the resources that are the object.

We can see two sample codes that show us the difference between synchronized and unsynchronized code.

4.2.1 WITHOUT SYNCHRONIZATION

Program: To create two threads, i.e., MyThread1 and MyThread2, to display a set of values without any synchronization between them.

```
package unext.java_examples;

public class Table {
      public void printTable(int n){//synchronized method
                for(int i=1;i<=5;i++){
                  System.out.println(n*i);
                  try{
                   Thread.sleep(400);
                  }catch(Exception e){System.out.println(e);}
                }
      }
}
```

4.2.1.1 Table class

```
package unext.java_examples;

public class MyThread1 extends Thread{
	Table t;
	MyThread1(Table t){
		this.t=t;
	}
	public void run(){
		t.printTable(5);
	}
}
```

4.2.1.2 MyThread1 class

```
package unext.java_examples;

public class MyThread2 extends Thread {
	Table t;
	MyThread2(Table t){
		this.t=t;
	}
	public void run(){
		t.printTable(100);
	}
}
```

4.2.1.3 MyThread2 class

```
package unext.java_examples;

public class TestSynchronization {
	public static void main(String args[]){
		Table obj = new Table();//only one object
		MyThread1 t1=new MyThread1(obj);
		MyThread2 t2=new MyThread2(obj);
		t1.start();
		t2.start();
	}
}
```

4.2.1.4 TestSynchronization class to verify the flow between MyThread1 and MyThread2

Output:

```
5
100
```

```
10
200
300
15
20
400
25
500
```

4.2.2 WITH SYNCHRONIZATION

Program: To create two threads, i.e., MyThread1 and MyThread2, to display a set of values without any synchronization between them.

```
package unext.java_examples;

public class Table {
	synchronized public void printTable(int n){//synchronized method
		  for(int i=1;i<=5;i++){
		    System.out.println(n*i);
		    try{
		     Thread.sleep(400);
		    }catch(Exception e){System.out.println(e);}
		  }
	}
}
```

4.2.2.1 Table class with synchronized printable() method

```
package unext.java_examples;

public class MyThread1 extends Thread{
	Table t;
	MyThread1(Table t){
		this.t=t;
	}
	public void run(){
		t.printTable(5);
	}
}
```

4.2.1.2 MyThread1 class

```
package unext.java_examples;

public class MyThread2 extends Thread {
	Table t;
	MyThread2(Table t){
		this.t=t;
	}
	public void run(){
		t.printTable(100);
	}
}
```

4.2.1.3 MyThread2 class

```
package unext.java_examples;

public class TestSynchronization {
	public static void main(String args[]){
		Table obj = new Table();//only one object
		MyThread1 t1=new MyThread1(obj);
		MyThread2 t2=new MyThread2(obj);
		t1.start();
		t2.start();
	}
}
```

4.2.2.4 TestSynchronization class to check synchronization between MyThread1 and MyThread2

Output:

```
5
10
15
20
25
100
200
300
400
500
```

STUDY NOTE

Object.wait() and Thread.sleep() are entirely two different methods in Java that are used in two different contexts.

- wait() method releases the acquired lock while sleep() method does not release the lock.
- waiting thread can be awakened by invoking notify()/notifyAll() methods while sleeping thread cannot be awakened.

4.3. Inter Thread Communication

We learned about thread creation earlier in this unit, and therefore, we know that thread creation leads to some memory usage. In server, every time a task arises to create and destroy a new thread, it takes a toll on the time and memory occupancy and, eventually, the efficiency of the server. One solution would be to limit the number of threads and the other – recycling!

Thread pools facilitate this recycling by reusing available or previously created threads to execute the tasks at hand. This reduces the problems of resource thrashing and thread cycle overhead. The time gap needed to create a thread for a task is reduced due to prior availability, and therefore, this enhances the system's responsiveness. But this method is not without its risk factors:

- **Deadlocks:** All the executing threads are queued up and made to wait for the results of the blocked threads due to the unavailability of threads for execution.
- **Thread Leakage:** If one thread is removed from a pool for a task execution but not returned to the pool, thread leakage occurs, resulting in reduced pool size. If this continues, the lack of threads for task execution becomes another problem.
- **Resource thrashing:** Possessing a more than an optimal number of threads poses an issue as the complexity rises with context switching between the threads. This results in a starvation problem, and in turn, resource thrashing occurs.

For a processor (N), the maximum size of the thread pool is N or N=1. This accounts for maximum efficiency. When tasks are waiting for input or output, then in those cases, the maximum size of the thread pool becomes

N*(1+W/S)

where,

W = waiting time for a request

S = service time for a request

To overcome these issues caused by thread pools, inter-thread communication is facilitated in Java. Inter-Thread communication efficiently allows more than one thread to communicate by reducing CPU idle time when CPU cycles are not wasted.

An increase in CPU idle time means that the threads await data from another thread's execution. The longer the waiting time, the higher the number of wasted CPU cycles.

Therefore, if the threads involved communicate when their tasks are completed and notify each other in the same regard, then the wastage of CPU cycles can be reduced.

Methods of the object class are also involved in this communication:

- **wait()** – instructs the thread to give up the monitor and sleep. Another thread enters and calls notify.
- **notify()** – instructs a thread to wake up due to a call wait() on the same object
- **notifyAll()** – wakes up all threads that are called wait() on the same object.

All the classes in the thread possess these three methods, and hence, these are called final Objects.

Self-Assessment Questions - 3

17. _____ are blocks or segments of code that are executed only when they are called.
18. Which method suspends the thread for a certain time period, shifts the thread into the Suspended stage?
    a. void start()
    b. void yield()
    c. void sleep()
    d. none of the above
19. Boolean (isAlive) method starts the execution of a thread. [True/ False]
20. Synchronization of threads avoids problems caused by _____ and _____.
21. The objects in Java are all associated individually with a _____, and only _____ can lock or unlock these _____.
22. _____ work by reusing available or previously created threads to execute the tasks at hand.
23. Longer the threads wait for information, higher is the _____ of CPU cycles.

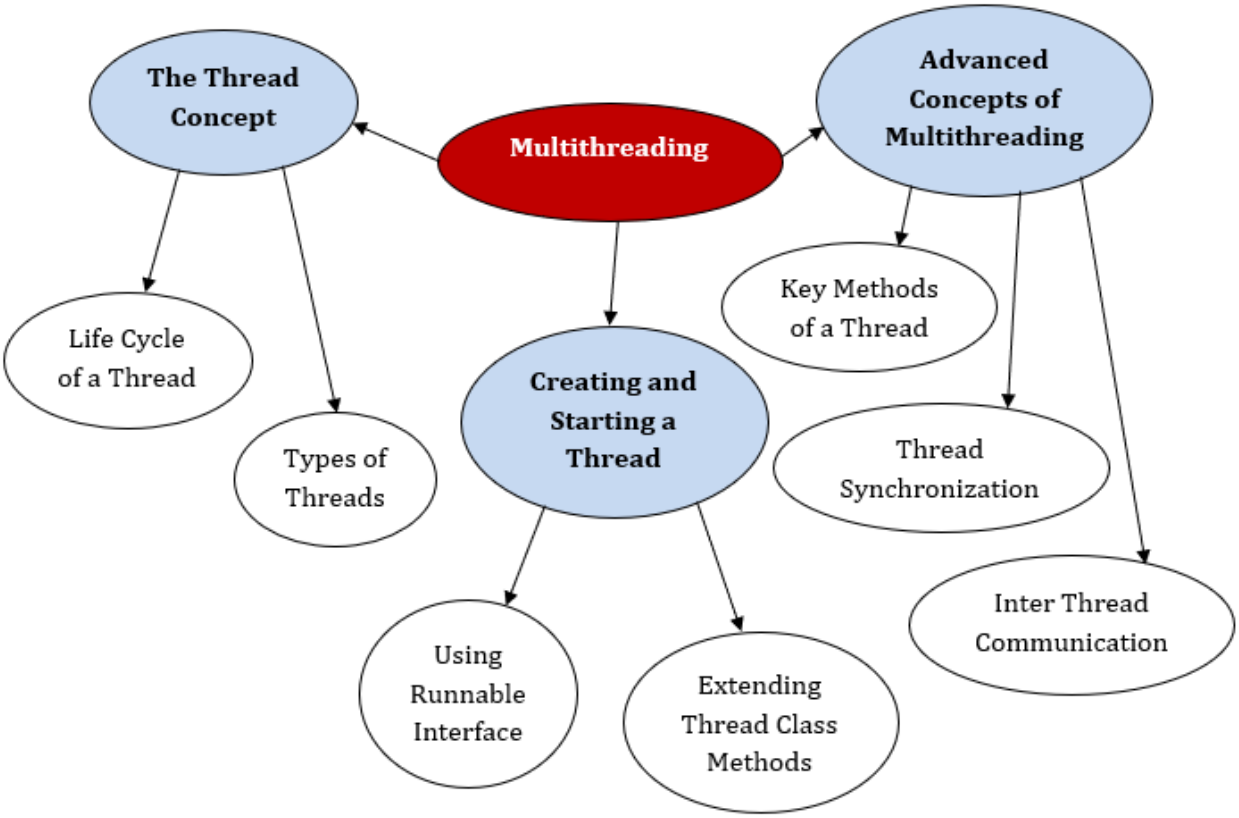


Fig 7: Conceptual Map of Multithreading

5. SUMMARY

- The perks of having a programming language like Java are attributed to its many functionalities.
- The number of principles this language was built upon make for its robust nature.
- One such principle that eases out the workload on the servers and systems is multithreading.
- Multithreading involves the use of threads which are lightweight processes that get executed independently.
- These threads are prioritized and executed in an orderly fashion determined by the platform it is run on.
- The threads help segregate and divide the workload and carry out background operations simultaneously without interrupting the main program.
- They pass through 5 stages of existence, and these stages are: Ready to Run, Running, Suspended, Blocked, Terminated.
- Methods are bits of code or functions that act when called. These mediate the functioning of the tasks efficiently by conveying information about the status of a thread to the other threads in simultaneous action with it.
- They take part in the creation and starting of a thread too.
- Thread creation and starting are done by two methods: either by extension of the thread classes or the more preferred Runnable implementation that allows further activation of classes when using shared resources.
- There are many complications associated with multi0threading despite all the amazing pros, which involve the complexities of the number of threads formed and multiple threads accessing a single shared common resource for task execution.
- These problems face solutions like enhancing Synchronization between the threads employing a monitor-lock-based algorithm where only one thread can enter these monitors at a time or by cooperative actions between the threads.
- And further enhancement of the multithreading process can be done by optimizing inter0thread communication using final object methods like wait(), notify(), and notifyAll(), reducing the CPU cycle wastage.

- Developers must peer into the thread mechanics and respective algorithms to pinpoint more complications and identify their solutions, providing us with more space and time to work on the business logic front to better utilize this multithreading principle.

6. GLOSSARY

Threads: A sequence of instructions that are executed in accordance with the process's context. These are lightweight and can be executed independently.

Multithreading: Allows access to multiple threads via parallel or concurrent processing.

Parallelism: When two threads are being executed simultaneously, it is called parallelism.

Concurrency: When at least two threads are progressing parallels, then it is called concurrency. This exists as a form of virtual parallelism.

Mutual exclusion locks: Objects that lock and unlock monitors and thereby control the access of threads to shared data.

Deadlock: When two threads are waiting for each other to proceed, neither are being executed, leading to no execution.

Main thread: Created by the Java Virtual Machinery and is executed by the main() method.

Monitor: The block that allows one thread to enter into a synchronized statement on the object.

Synchronized statement: blocks of code declared using synchronized(object)

Starvation: Indefinite postponement of execution of low priority threads by higher priority threads.

Waiting: Threads in the queue are waiting to complete another thread's execution and exist in a timed waiting state.

Sleep interval: The period between successive backups. Upon its expiration, the thread is returned to its runnable state.

Blocked: A process is blocked when it is waiting for some event, such as a resource becoming available or completing an I/O operation.

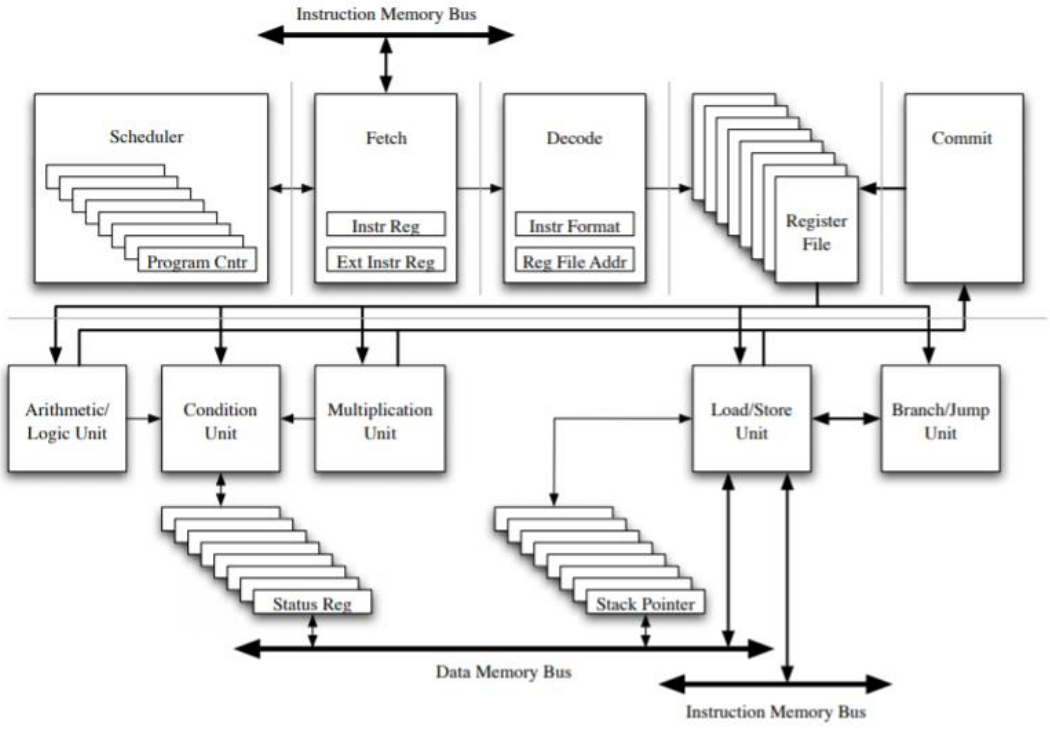
7. CASE STUDY

MULTI THREADING IN EMBEDDED SPACE

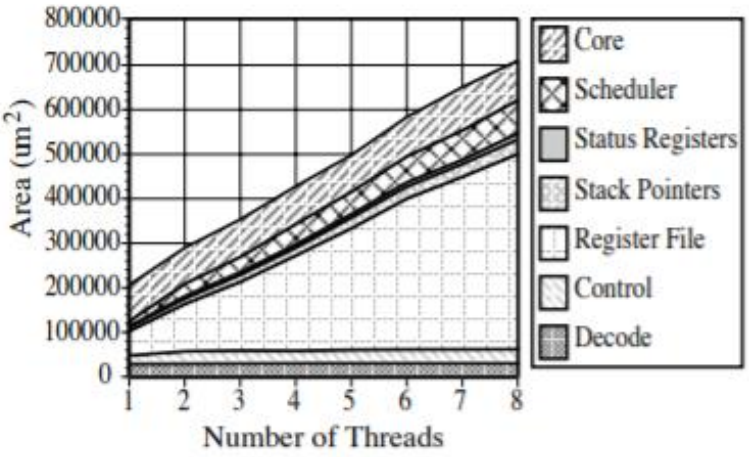
The paper is authored by Greg Hoover, Forrest Brewer and Timothy Sherwood, from the university of California, they aimed to summarise and showcase the advantages of minute, multi-threaded microcontroller design for in increasing the responsiveness of targeted embedded applications.

Generally multi-threading improves resource utilisation, but in embedded spaces it can provide zero-cycle context switching and interrupt service threads.

This paper has studied and quantified the ramifications from a circuit level to that of software layers, which is novel idea.



Multi-threaded, microcontroller - JackKnife core components - overview



Area breakdown by component

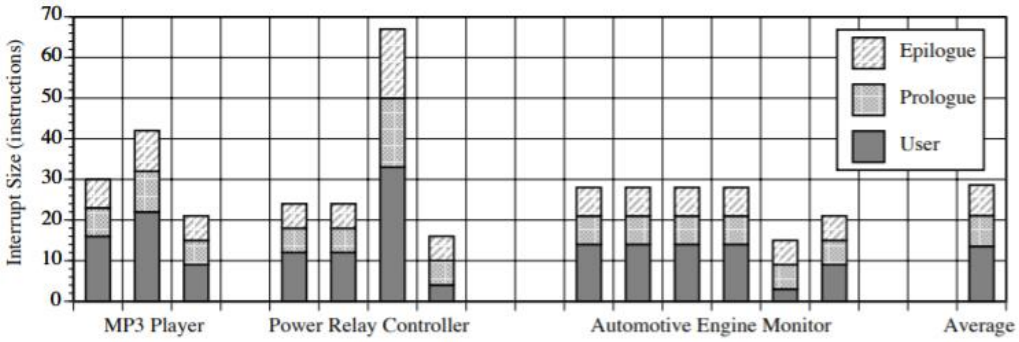
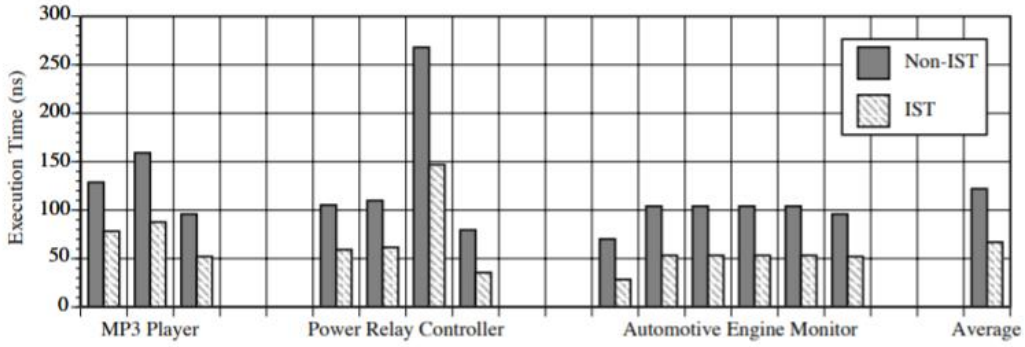
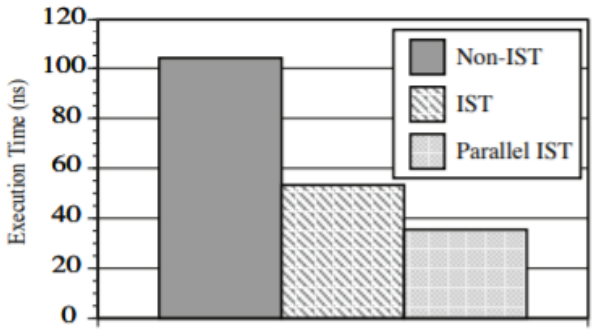


Figure 11: The sizes (instructions) of the interrupt service routines from three different AVR applications, showing the break down of prologue, epilogue, and user instructions.



A comparison of execution time (ns) for running the surveyed interrupt routines in a single threaded implementation (non-IST) versus a multi-threaded implementation employing the use of interrupt service threads.



Comparison of execution time (ns) of a typical interrupt routine when coded for single threaded execution, as an IST, and the average time for running multiple copies of the routine in parallel, with single-cycle response times

Supporting interrupt service threads (IST) with zero-cycle context switching and reduced ISR overhead provides response times on the order of nanoseconds with typically 50% reduction in execution time.

Conclusion: Multithreading in a complex scenario such as a pipelined machine having highly complex synchronization and I/O handling is possible. This design will be a starting point and open up many avenues for integrated solutions.

Source: https://sites.engineering.ucsb.edu/~sherwood/pubs/CASES-embeddedMT.pdf

Discussion Questions:

1. Discuss JackKnife processor architecture.
2. Explain effect of multithreading in JackKnife implementation.

8. TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Q1. What is Multithreading?

Q2. What are threads?

Q3. Explain Synchronization of Threads.

Q4. List the key methods of threads.

Q5. What is inter-thread communication?

LONG ANSWER QUESTIONS

Q1. Elaborate on how you would create and start a thread.

Q2. Compare non-synchronized vs. synchronized threads and their outputs.

Q3. How does Synchronization work? Elaborate on the two types of thread Synchronization?

Q4 What are the challenges with multithreading and how is it addressed. ?

9. ANSWERS

SELF ASSESSMENT QUESTIONS

1. James Gosling
2. False
3. Multithreading
4. Threads
5. False
6. C. suspended
7. Users, Main
8. setDaemon(True)
9. False
10. C. Constructors
11. Run()
12. B. Start()
13. Objects, memory spaces
14. False
15. Inheritance
16. Extending Thread class and implementing Runnable interface
17. Methods
18. Void sleep()
19. False
20. Thread interference and consistency issues
21. Monitors, threads, monitors
22. Thread pools
23. Wastage

TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Answer 1: Multithreading: Allows access to multiple threads via parallel or concurrent processing. It allows the execution of multiple concurrent tasks by a single process by distributing the task requirements efficiently to the process.

Answer 2: Threads are light-weighted processes that do not interfere with or affect the main program and can be executed independently. This helps in getting more work done (task execution) while enhancing the speed of processing and system efficiency. This property of the thread is also an advantage to the error corrections and clarifications as their independence makes this step easier and does not hinder the main program's processing. These threads share a memory space and also have their variables, stack, and program counter.

Answer 3: Process of controlling how the threads access these shared resources is termed Synchronization. The synchronization process involves allowing only one thread to access a resource at a particular point in time. The syntax for the synchronized statement is

```
synchronized(objectidentifier) {
  // Access shared variables and other shared resources
}
```

The object identifier denotes the object that is locked in the monitor represented by the synchronized statement. The threads can only open these locks and access the resources that are the object.

Answer 4: Key Methods:

- String getName()
- void start()
- void run()
- void sleep(int sleeptime)
- void yield()
- void join()
- Boolean(isAlive)

Answer 5: Inter-thread communication is facilitated in Java to overcome these issues caused by thread pools. Inter-Thread communication efficiently allows more than one thread to communicate by reducing CPU idle time when CPU cycles are not wasted. For Long answers, please refer to the material or the references.

LONG ANSWER QUESTIONS

1. Refer Section "Creating and Running threads".
2. Refer Section "Thread Synchronization".
3. Refer Section "Thread Synchronization".
4. Refere Section "Inter thread Communication".

10. SUGGESTED BOOKS AND E-REFERENCES

BOOKS:

- Oaks, Scott and Wong, Henry, (2004), Java Threads, O'Reilly Media.
- Lea, Doug., (1999), Concurrent Programming in Java: Design Principles and Pattern, Addison-Wesley Professional.
- Goetz, Brian., Peierls, Tim, Bloch., Joshua, Bowbeer., Joseph, Holmes., David, Lea, Doug., (2006), Java Concurrency in Practice Addison-Wesley Professional.
- Subramaniam, Venkat., (2011), Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors. Pragmatic Bookshelf.

E-REFERENCES:

- Multithreading in Java Tutorial with Program & Examples viewed on September 28th, 2021, <https://www.guru99.com/multithreading-java.html#3>
- Java Multithreading tutorial, viewed on September 28th, 2021. <https://www.geeksforgeeks.org/java-multithreading-tutorial/>
- Java Multithreading tutorial for beginners, viewed on September 28th, 2021, <https://www.techbeamers.com/java-multithreading-with-examples/>
- Java InterThread Communication viewed on September 28th, 2021, <https://www.studytonight.com/java/interthread-communication.php>

- Difference Between Daemon Threads and User Threads in Java, viewed on September 29th, 2021, <https://www.geeksforgeeks.org/difference-between-daemon-threads-and-user-threads-in-java/>
- Thread Concept in Java, viewed on September 29th, 2021, <https://www.javatpoint.com/thread-concept-in-java>
- The history of java multithreading was viewed on September 29th, 2021, <https://www.programmersought.com/article/35995178094/#:~:text=Java's%20genes%20come%20from%20an,of%20API%20both%20disgust%20programmers>
- Java Thread Tutorial: Creating Threads and Multithreading in Java, viewed on September 29th, 2021, <https://www.edureka.co/blog/java-thread/>
- History of Java viewed on September 29th, 2021, <https://www.javatpoint.com/history-of-java>