International Research Journal of Computer Science (IRJCS)
Issue 02, Volume 3 (February 2016)
ISSN: 2393-9842
www.irjcs.com

Thread Organization and Intra-Thread Datasharing (T.O.I.T.D)

Fenil R. Doshi
Computer Science and Engineering
SRM University Kattankulathur

Rohan H. Pooniwala
Computer Science and Engineering
SRM University Kattankulathur

G. Vijayalakshmi
Computer Science and Engineering
SRM University Kattankulathur

Abstract*—Multithreading is a major part of parallel programming used almost everywhere but it needs to be implemented effectively to make it useful. There are many ways to implement multithreading in any language. In this paper we show how multithreading can be implemented easily with the use of basic concepts like static variables, etc and then we show how different threads can communicate and interact with each other using the same concepts.Method shown in this paper is similar to how the OS handles multiple processes. We will demonstrate the selective activation of Threads and also demonstrating the execution of the threads sequentially. We also discuss intra-thread communication using static variables. Thus the difficult concept of synchronization and the "join()" method is both replicated and simplified using the basic methods. Finally we present a data structure to depict multiprocessing in a small application using the above discussed concepts.

Keywords*— List of important variables: no_of_threads, thread_turn, thread_no, running, and container; Runnable; Static variables; Downcasting.

I. INTRODUCTION

Multithreading is a major part of parallel programming. It simulates multiprocessing on kernel level by executing commands one after other from different set of codes. It is often used to make a program faster by sharing the resources thus reducing the overhead on both time and the processing power (As shown in [2] and [3]). Above all this it is more important to keep the processor busy and the processes more responsive. By busy we mean that if there is some activity that demands some user action, then this activity should not hinder with some other background processes, thus saving a lot of time.

There are many ways to implement multithreading in a program. The method we used in this paper is similar to how the OS keep all the multi-processes in order. The algorithm used is not exact clone of the algorithm used in the OS but it works in similar manner. The algorithm used in OS cannot be used here directly because of its complexity in code (Given in detail in [1] and [7]).

We have assigned a specific unique number to every thread running by the main thread. And then we use this number to run, pause or kill the thread (Refer Fig. 3 and Fig. 4). Further, this method can be implemented to kill thread at runtime or make a step by step sequential execution of several threads. The main features of programming language used for this are (more information in [4]):

- *OOP*
- *Inheritance*
- *Static Variables*

Threads should also be able to interact with each other by sending and receiving data. Method shown in the paper (Intra-thread communication) simplifies this task by using the same above concepts that is static variables, etc (which supposedly should be implemented in its generic form).

We specify the specification of this idea. We specifically used Java to demonstrate the implementation although the algorithm specified in this paper can be applied in any language which supports the above features.

II. BASIC SETUP (THREAD ORGANIZATION)

In the basic setup we have one major class – the superclass which will be implementing/extending the Thread classes (mainly the Thread class or the Runnable interface in Java) (Refer Fig. 1). This superclass can be made abstract to denote that it cannot be instantiated and its sole purpose is to act like a bypass factory for all the other subclasses which are going to denote various processes performing different tasks.

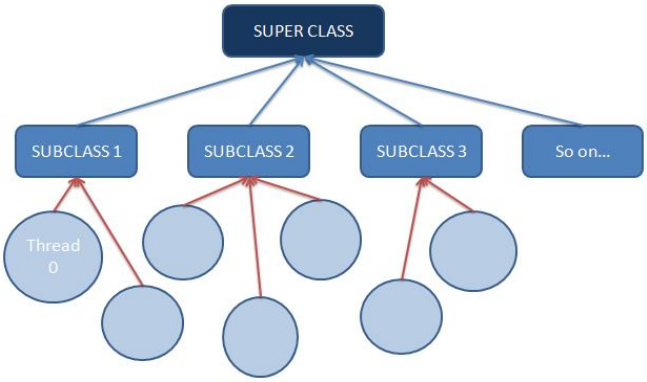

Fig. 1: The general layout of the setup.

Now we put in values inside this superclass which will be extended in all the subclasses. The superclass will consist of (Refer Fig. 2):

1. *A static integer depicting no of threads of the superclass: "no_of_threads"*
2. *Static Array with Dynamic length (basically an ArrayList in Java): this will hold the data whether a particular thread is running or paused or killed: "running"*
3. *The Thread number of that particular thread: "thread_no"*
   (Just remember that all static members of a class are common not only to the objects of that class but also to all the objects to all of its subclasses)

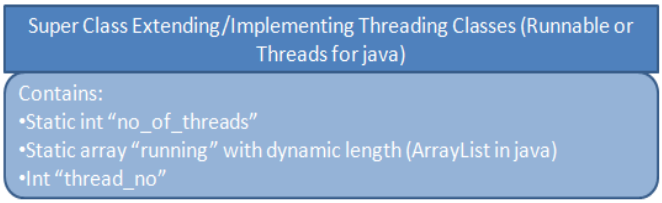

Fig. 2: The super class along with its contents.

III. How Thread Number Is Assigned

When the subclasses are instantiated (either referenced by the subclass itself or even the superclass) the thread object will be immediately created. Now there will be a constructor that will assign that thread a unique ID which is the "thread_no" and will increment itself by 1. Like this the "thread_no" is not only the unique id but also the '(n+1)th' thread created. Further the constructor will also start this thread which will call the run method of the subclass whose object the thread it is. Every subclass will have its own separate run method. Apart from this the Constructor will also indirectly add this thread object to the "running" array (dynamic array). By indirectly we mean that the thread object is not itself added to the array but the index position of the new added element in the array list is same as the "thread_no" of that thread. The element that is added is an integer which is by default 0. 0 signifies paused thread (although the thread will be started). It is important to add the Thread to the "running" array before starting it as a precautionary approach. The thread should be started at the last line inside the constructor.

IV. How Thread is Activated/Paused/Killed

Now once we have all the started threads and at the default mode. We can now manage the Threads and decide if we want to run them in a sequential order (of the sequence already established) or based on the users decision and just let random active threads run in the background.

When the thread is active it will perform a particular task which is not exactly the entire code in the run method of the specific thread but that task is a part of code embedded inside the run method and when paused, it will not perform that task. Once killed, we cannot activate the thread again. (Refer Fig. 3)

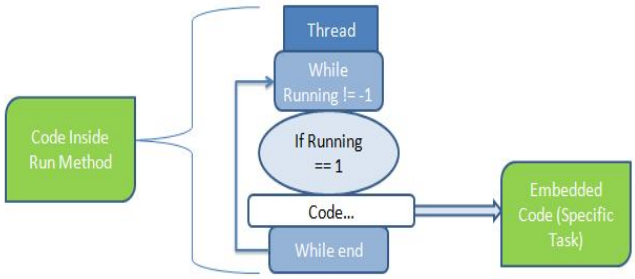

Fig. 3: Structure of run method of the thread.

A. User's Decision

The user will decide which thread he wants to run and keep active and which thread he wants to pause or kill during runtime.

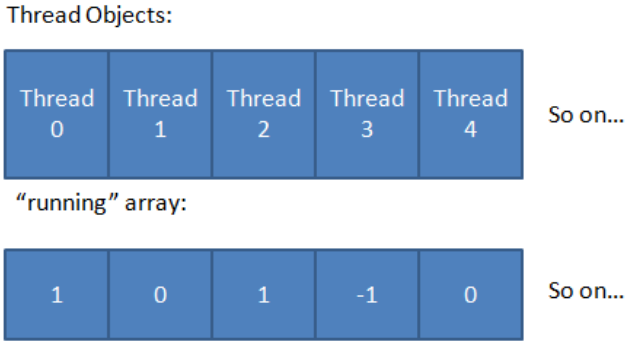

Fig. 4: The arrangement of the "running" array with offset positions to access every element.

The user will explicitly call upon methods that will activate (set the corresponding element at the index number same as thread number of the "*running*" array to 1) or pause (set the corresponding element at the index number same as thread number of the "*running*" array to 0) or kill (set the corresponding element at the index number same as thread number of the "running" array to -1).

Value	Significance
0	Paused
1	Active
-1	Kill Status

Fig. 5: The possible statuses of every thread.

The activated threads will run the embedded task in the run method of the thread object.

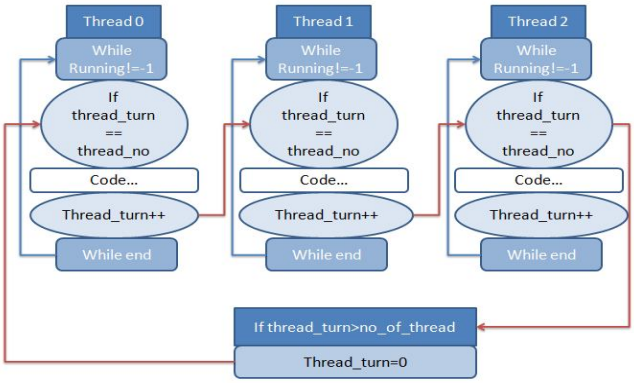

Fig. 6: Implementation of sequential flow with the code block representing the specific task.

B. Implementing Sequential Flow Introduction

In this, we start a task which is next in the queue from currently running task. For this, we create a new static int "*thread_turn*" which signify which thread is supposed to run. When it is time for next thread to start, the current thread changes the "*thread_turn*" to the thread number of the next thread and then finishes the remaining work (the work can be made to run in the background while the next intended thread is performing the required task). This is useful when a work is to be done by executing some tasks one after other and some tasks at the same time. The sequential running can be randomized based on the code or the users input or can be an automated sequence by changing the thread_turn in a regular pattern.

1) Algorithm:

- *First, we create a new variable called "thread_turn". This variable signifies the thread which is supposed to be run next.*
- *Thread_turn is set to the thread_no of the first thread.*
- *In all thread, an if-else statement is created inside the repeating loop (for all activated or paused threads) which checks whether it's the turn of the thread or not.*
- *The code to be run (the specific task) is written in the if-block. At the end of the code, thread_turn is set to next thread's thread_no.*
- *In last thread, the thread_turn is again set to first thread (for a cyclic run).*
- *Thus sequential flow is obtained.*

2) Working:

First of all the thread should either be in an activated or a paused state to enter inside the True repeated loop (if it is in kill state, the flow will stop and thread number will not be changed). Now once inside the loop, when thread_turn is set to first thread, the if-condition in the first thread is set true, and thus task is performed. At same time, other threads are not completely inactive. A while-loop is running in them checking if they are alive (active or paused) or killed and the if-condition is checking if it is the turn of the desired thread. Therefore, when thread_turn is changed to the next thread, next thread starts. And the thread will perform the required task.

V. BASIC SETUP (INTRA-THREAD DATASHARING)

Now assuming that the Thread Organization model is already implemented, we go forward on understanding how will we make these threads communicate.

For this purpose, along with all the contents already discussed that the superclass contains , we also create another static variable object which is container type and generic in nature(an arraylist in java or a list in python) , thus allowing the threads to communicate in form of any datatype available in the language that you have chosen.

Let's say this generic object is an array called "container".Every subclass will have its own way of dealing with the communicated element that it will be receiving.

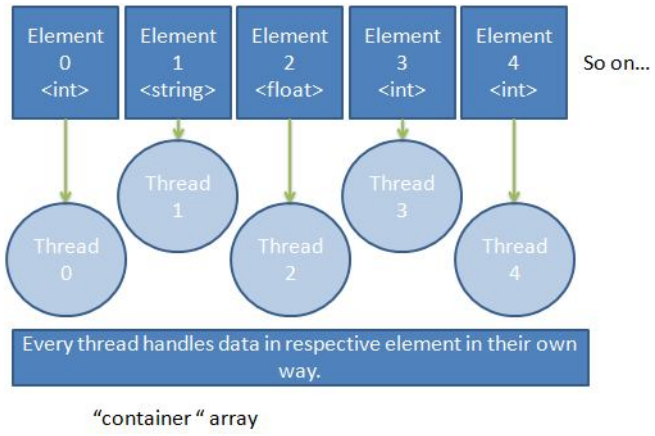

Fig. 7:The "container" array through which data exchange takes place.

VI. How Data is shared between Threads

This section is going to help us understand how data is going to be shared between multiple threads at runtime. When a thread is running and performing a particular task, let's say it wants to pass some information to some other thread which is not running at that very second. So what we can do is simply passing on the message (which can be a string, integer, float, etc or any other datatype) to that element in the "container" array whose index number is same as the thread number of the desired thread to whom the message is being sent. Since the container object is of generic type, it can store data of any type and later be downcasted and operated upon depending on which type the original data was. Thus the sending of the data packet is done. For other approaches, refer [6].

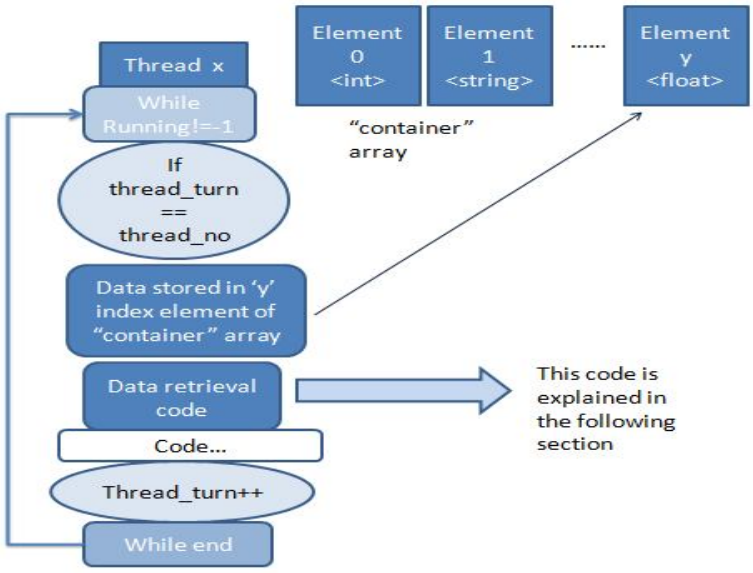

Fig. 8: Storing data from thread to "container" array.

Now once the data is sent .The data has to be received by the desired thread. This can be done in the following two ways:

A. Receiving when the turn to perform the task comes

In this case the thread which has to receive the message, will receive the message when their turn to perform the task comes. That is, they will receive the message once they are inside the if- condition. Inside the if-condition there will be the data retrieving code which will read the data from the element of the "container" object whose index number is same as that of the thread number. After receiving the data packet from that element, two things need to be done:

1) *The content inside the element from which the data is read has to be erased and reset to their default value. This step is just taken for safety measures in case the element is read again.*
2) *The data which is read has to go for the type checking tests and then has to be downcasted to its particular type. Now the data is readily accessible and can be operated upon. (in Python the type checking tests are done using the type() factory function and in Java we use getClass().getName() method).*

The diagram shown in Fig. 8 demonstrates how the thread will receive data when its turn to perform the task comes.

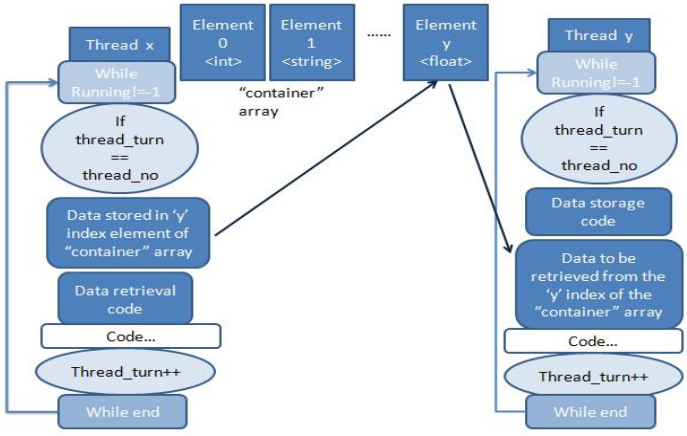

Fig. 9: Data retrieval when turn of the thread comes.

B. Receiving in the background

In this case, the thread which has to receive the message will receive it even when its turn to perform the task has not been arrived. The retrieval code is outside the if-condition suite. The method of retrieving the data packet still remains the same as described in the above section, the only thing changing is the location of the code.

The major advantage is that here we are tapping on the full potential of parallel computing by not only allowing the threads to run in a parallel manner but by letting every particular threads do two tasks simultaneously . This will reduce the factor of time to a great extent in case of large number of threads.

Also create the data retrieval code above the if-condition in the while loop to avoid the situation where the thread has to process on the data but the data is not available yet.

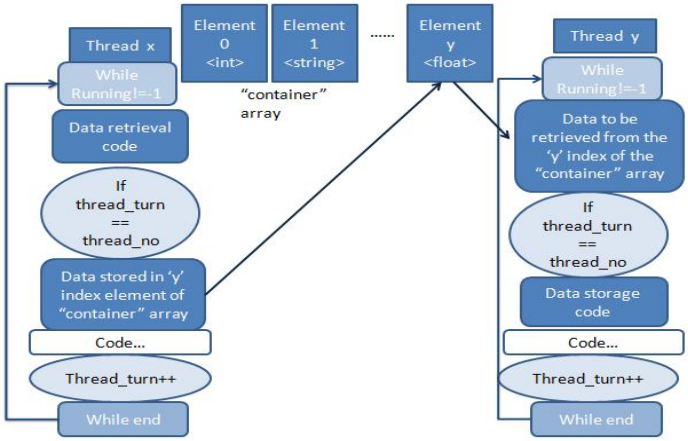

Fig. 10: Data retrieval in the background.

After receiving the data packet from that element the same two steps as described in the previous section should be executed.

VII. CONCLUSIONS

We finally conclude that the above said approach will reduce both, the overhead on time and complexity of the program. The above task can be achieved with objects too, but every time the object is used, the virtual machine will need to search or load the class and then find the object in the heap. However here, searching through the running threads is way easy for the machine. Including this, one big advantage is that we are able to tap on the maximum potential of the processor in achieving a particular task.

Apart from this there can be one enhancement in the future on this model. When a thread is killed it will stop performing any task but the element with the corresponding index number will remain seated in the "running" array, thus wasting memory. This memory leak can be avoided by changing the way every thread is assigned a "thread_no" variable. Before assigning it using the static variable "no_of_threads" we first check the "running" array and search for any element whose corresponding thread is killed. If we get any such element , we assign the index number of that element to the "thread_no" of the thread , without making any change to the static variable "no_of_threads".

ACKNOWLEDGEMENT

This research paper is made possible through the help and support from everyone, including: parents, teachers, family, and friends.

Especially, please allow us to dedicate our acknowledgment of gratitude toward the following significant advisors and contributors. First and foremost, we would like to thank our ***professor Vijayalakshmi*** for her most support and encouragement. She kindly read our paper and offered invaluable detailed advices on all technical flaws which were avoided with the right guidance. Finally, we sincerely thank our parents, family, and friends, who provide the advice and financial support. The product of this research paper would not be possible without all of them.

REFERENCES

[1] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, *Operating System Concepts*, Ninth Edition , Chapter 4 (Multithreading in operating systems).

[2] *Introduction to Parallel Computing*. [Online] Website: https://computing.llnl.gov/tutorials/parallel_comp/

[3] Joe Johnson, and G. VijayaLakshmi, *Comparison Study of Parallel Computing with Alu and GPU (cuda),* International Journal of Science and Research(IJSR),India Online ISSN: 2319-7064.
[4] *Object-Oriented Programming Concepts.* [Online] Website: https://docs.oracle.com/javase/tutorial/java/concepts/
[5] Massimiliano Meneghin, Davide Pasetto, Hubertus Franke, Fabrizio Petrini, and Jimi Xenidis, *Performance evaluation of inter-thread communication mechanisms on multicore/multithreaded architectures.* [Online] Available: http://researcher.watson.ibm.com/researcher/files/ie-pasetto_davide/PerfLocksQueues.pdf
[6] Research Gate Website. [Online]. Available: http://www.researchgate.net/post/What_is_the_simplest_possible_way_for_threads_to_communicate
[7] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean, *Using Continuations to Implement Thread Management and Communication in Operating Systems,* [Online] Available: http://www.cse.iitd.ac.in/~sbansal/os/bib/draves91continuations.pdf

