Synchronization in Dining Philosophers Problem Using Lock & Release Algorithm

Andysah Putera Utama Siahaan
Universitas Sumatra Utara
Jl. Dr. Mansur No. 9, Medan, Sumatra Utara, Indonesia
andysahputrautamasiahaan@yahoo.com

Abstract — This paper studies how to prevent deadlocks in dining philosophers problem. Deadlocks are undesirable states of concurrent systems, characterized by a set of processes in a circular wait state, in which each process is blocked trying to gain access to a resource held by the next one in the chain. Deadlock prevention is commonly used in distributed real-time and operating systems but, since concurrency is severely limited, an efficient distributed deadlock avoidance schema can have a big practical impact. However, it is commonly accepted that the general solution for distributed deadlock avoidance is impractical, because it requires maintaining global states and global atomicity of actions. The communication costs involved simply outweigh the benefits gained from avoidance over prevention. This paper presents an efficient distributed deadlock avoidance schema using lock and release method that prevents other thread in chain to make race condition.

Keywords — Dining Philosophers Problem, Race Condition, Concurrent, Deadlock, Starvation

I. INTRODUCTION

The term concurrent programming usually refers to the kind of programming required to support simultaneous activities in a software application.

Such a definition is somewhat broad, and indeed concurrent programming can refer to a variety of programming models. For example, multi-processor systems, parallel systems, distributed systems as well as uni-processor systems with simulated concurrency - all these platforms support concurrent programming. In this article, we take particular emphasis on the uni or multi- processor system with actual or simulated concurrency. Many of the issues we handle here are general and can be applied to the other systems as well.

Concurrent programming, as we take focus here, is the set of techniques and tools used in programming concurrent activities that dynamically share information with one another. Note that dynamic information sharing is an important criteria. Otherwise programming two different programs running on two different machines would qualify as concurrent programming under an indiscriminate definition.

II. THEORIES

The programmer new to concurrent programming usually comes across a multitude of terms like threads, semaphores, mutexes, monitors, conditional variables, signals, delayed variables, critical sections, RPCs, rendezvous, remote evaluations, synchronous/asynchronous data transfer, message queues, shared memory, priorities, deadlocks and so on. He/she will also have been introduced to certain “sample” problems like Dining Philosophers and Producer-Consumer problems which usually fail to make much sense. What does it mean if five philosophers get together to eat spaghetti? Well, hardly anything. But the example serves as a classic case common to most concurrent programming issues, if only it is viewed together with practical scenarios. So it is usually best to introduce terms and concepts on a need-by-need basis.

Most of the time in most of the PC-running applications , there are some applications that can benefit from concurrency. Then, there are applications where concurrency is a requirement especially in real-time systems. And finally, there are applications that are inherently distributed in nature, like e-mail for example.

We will touch briefly upon why email is a concurrent system belonging to the last class mentioned above. The e-mail system is well designed and it has interacting components running on multiple machines, dedicated to routing and delivering your mail across the world. It has avoided dead-locks and many other problems. Let’s say that an email you sent to a friend bounced, and by the time it gets back to you, your email account is deleted. What is there that prevents orphaned e-mails from traveling the world back and forth through endless number of servers? What happens if two email delivery components try to send to each other at the same time? As we can see there are many problems due to concurrent operations showing up here.

To take another example, consider a packet broadcast into a network (a general graph). What prevents it from circling round and round? There is logic active in these systems that work only with co-operative agreement and action. And such an agreement put to action is called a protocol. There is concurrent logic at work and they are concurrent systems. Even your network card on the ethernet is part of a concurrent system – because it follows a protocol for avoiding access contention and co-operates (indirectly) with other cards on

your network. Only, the concurrency design was done in the ethernet white papers and the cards are simply following their assigned task.

Again, such systems have to be wary about live-locks. That is, the system is not frozen, but keeps doing something that is useless or wasting resources. Does that happen? Look at the following example.

Many people are familiar with the "finger" program on the Internet. A finger server or daemon in Unix terms supports queries from a client who issues a finger command. Some sites do a "reverse finger" on the client's finger service to find out who is fingering them. Sounds like a reasonable requirement. Now, what would happen if the client's finger service also is the "reverse fingering" type? Unless a special mechanism is in place, these two programs will either dead-lock or repeatedly finger each other and run out of resources and become useless, without extracting any information. That's a case of a live-lock.

We like to classify dead-locks and live-locks and many of the "effects" found in concurrent systems - as emergent phenomena. A phenomena is emergent when you do not explicitly make provisions for it, but it arises from the interactions between the components in your system. You don't code dead-locks. Neither do you code for live-locks or data corruption or system hangs. They come all of their own. What you have to make sure of is that they don't arise in your system. And this means controlling the functionality of your components in certain ways so that the collective effects are minimal. And that is what concurrent programming techniques are mostly about. So basically, many of concurrent programming techniques are ways and means of avoiding destructive emergent phenomena in concurrent systems.

Can we have a constructive emergent phenomena? Yes, you can. But like most emergent phenomenon, it is difficult to design the situation where it can turn up. Remember any times when your design seemed to solve certain problems without you actually putting it in there?. By a set of bells and whistles in place which were meant for other things, the problem got solved by itself. A constructive emergent phenomena that you would like to have is self-organization. More on this later.

So, getting back – Do we need concurrent programming? The answer is yes, if your application has inherent concurrency and requires either manifested concurrency or speed-up. It is yes, if your application has some inherent concurrency and there are substantial design simplifications to be had in using concurrency. But most of the time – it is a plain NO. It can result in messy situations and applications that are exceedingly difficult to debug. Despite what certain Java tutorials might say about threads and concurrency, they are best avoided in vanilla programming tasks.

III. Dining Philosophers Problem

Five philosophers are sitting around a circular table and there is a big bowl of spaghetti at the center of the table. There are five forks placed around the table in between the philosophers. When a philosopher, who is mostly in the thinking business gets hungry, he grabs the two forks to his immediate left and right and dead-set on getting a meal, he gorges on the spaghetti with them. Once he's full, the forks are placed back and he goes into his mental world again. The problem usually omits an important fact that a philosopher never talks to another philosopher. The typically projected scenario is that if all the philosophers grab their fork on their left simultaneously (which can happen by chance – and what can go wrong will go wrong – murphy's law) none of them will be able to grab the fork on their right. And with their one-track mind-set they will forev er keep waiting for the fork on their right to come back on the table.

The basic idea behind the scenario is that if a concurrent activity always does what seems best for itself or what seems to be the right thing for itself in a shared resources scenario, the result can be chaos. Is there a solution to the Dining philosopher problem? Actually the scenario was posed not for a solution but to illustrate a basic problem if the traditional programming approach is applied to concurrent systems. The problem itself crops up in your own concurrent systems and your design decisions should be aware of this and that is what you have to solve. Any set of concurrent programming techniques that you use are expected at the basic level to offer you features that can be used to deal with the Dining Philosophers problem in some way.

An illustration:

Assume that you have the simple task of writing some important information into two files on the disk. But these files are shared by other programs as well. Therefore you use the following strategy to update the files:

```
Lock A
Lock B
Write information to A and B
Release the locks
```

This seemingly straight forward and obvious coding can result in deadlocks if other tasks are also writing to these files. For example, if another task locks B first, then locks A, and if both tasks try to do their job at the same time – dead-lock occurs. My task would lock A, the other task would lock B, then my task would wait indefinitely to lock B while the other task waits indefinitely to lock A. This is a simple scenario, and easy to find out. But you can have a bit more involved case where task A can wait for a lock held by task B which is waiting for a lock held by task C which is waiting for a lock held by task A. A circular wait - a dead-lock results. This is a Dining Philosophers model.

In the above code fragment, one could resort to locking the files one at a time for modification. Then the problem would disappear. But there are times when requirements dictate that you have to lock more than one resource before updating them.

IV. IMPLEMENTATION

A good starting point in designing any event-driven system is to draw sequence diagrams for the main scenarios identified from the problem specification. To draw such diagrams, we need to break up your problem into active objects with the main goal of minimizing the coupling among active ob- jects. You seek a partitioning of the problem that avoids resource sharing and requires minimal commu- nication in terms of number and size of exchanged events.

Dining Philosophers Problem has been specifically conceived to make the philosophers contend for the forks, which are the shared resources in this case. In active object systems, the generic design strategy for handling such shared resources is to encapsulate them inside a dedicated active object and to let that object manage the shared resources for the rest of the system (i.e., instead of sharing the resources directly, the rest of the application shares the dedicated active object). When you apply this strategy to DPP, you will natur- ally arrive at a dedicated active object to manage the forks. This active object has been named "Table".

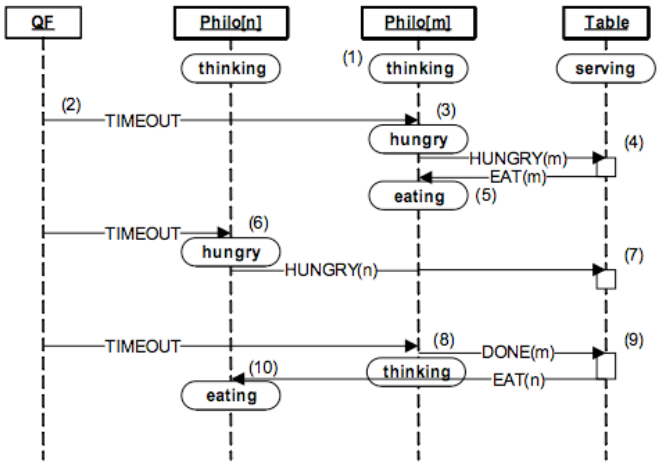

Fig. 1: The sequence diagram of the DPP application.

- Each Philosopher active object starts in the "thinking" state. Upon the entry to this state, the Philosopher arms a one-shot time event to terminate the thinking.
- The QF framework posts the time event (timer) to Philosopher[m].
- Upon receiving the TIMEOUT event, Philosopher[m] transitions to "hungry" state and posts the HUNGRY(m) event to the Table active object. The parameter of the event tells the Table which Philosopher is getting hungry.
- The Table active object finds out that the forks for Philosopher[m] are available and grants it the permission to eat by publishing the EAT(m) event.
- The permission to eat triggers the transition to "eating" in Philosopher[m]. Also, upon the entry to "eating", the Philosopher arms its one-shot time event to terminate the eating.
- The Philosopher[n] receives the TIMEOUT event, and behaves exactly as Philosopher[m], that is, transitions to "hungry" and posts HUNGRY(n) event to the Table active object.
- This time, the Table active object finds out that the forks for Philosopher[n] are not available, and so it does not grant the permission to eat. Philosopher[n] remains in the "hungry" state.
- The QF framework delivers the timeout for terminating the eating arrives to Philosopher[m]. Upon the exit from "eating", Philosopher[m] publishes event DONE(m), to inform the application that it is no longer eating.
- The Table active object accounts for free forks and checks whether any direct neighbors of Philo- sopher[m] are hungry. Table posts event EAT(n) to Philosopher[n].
- The permission to eat triggers the transition to "eating" in Philosopher[n].

At the application level, you can mostly ignore such aspects of active objects as the separate task con- texts, or private event queues, and view them predominantly as state machines. In fact, your main job in developing QP application consists of elaborating the state machines of your active objects.

Figure 2(a) shows the state machines associated with Philosopher active object, which clearly shows the life cycle consisting of states "thinking", "hungry", and "eating". This state machine generates the HUNGRY event on entry to the "hungry" state and the DONE event on exit from the "eating" state be- cause this exactly reflects the semantics of these events. An alternative approach—to generate these events from the corresponding TIMEOUT transitions—would not guarantee the preservation of the se- mantics in potential future modifications of the state machine. This actually is the general guideline in state machine design.

Fig. 2: State machines associated with the Philosopher active object (a), and Table act- ive object (b).

Figure 1(b) shows the state machine associated with the Table active object. This state machine is trivial because Table keeps track of the forks and hungry philosophers by means of extended state variables, rather than by its state machine. The state diagram in Figure 2(b) obviously does not convey how

the Table active object behaves, as the specification of actions is missing. The actions are omitted from the diagram, however, because including them required cutting and pasting most of the Table code into the diagram, which would make the diagram too cluttered. In this case, the diagram simply does not add much value over the code.

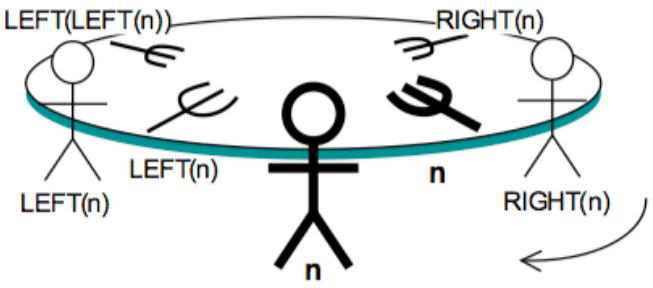

Fig. 3: Numbering of philosophers and forks.

V. Programming Overview

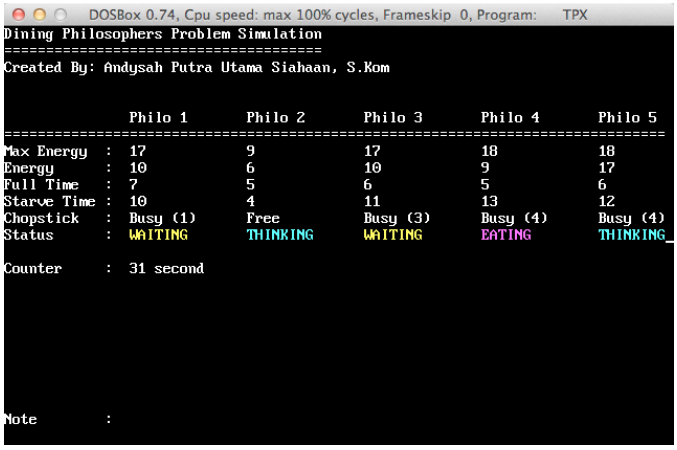

Fig. 4: Normal Running Program

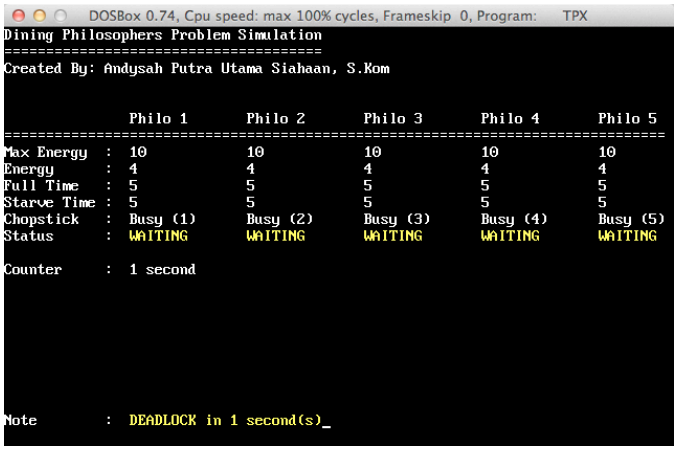

Fig. 5: Deadlock Running Program

VI. Conclusion

We have taken a little tour through the landscape of concurrent programming, and looked at certain issues particular to multi-threaded/multi-processing systems. These are not specific to any particular platform but can be applied to a variety of programming systems out there. There are further connotations for other systems like distributed systems, parallel-processing systems and also real-time systems which will be explored in another forthcoming article.

References

W. S. Davis and T. M. Rajkumar, Operating Systems A Systematic View, 5th Edition, Addison Wesley, 2001.

S. Tanenbaum, Modern Operating Systems, 2nd Edition, Prentice Hall, 2001.

Gary Nutt, Kernel Projects for Linux, Addison Wesley, 2000.

H.M. Deitel, An Introduction to Operating Systems, 2nd Edition, Addison-Wesley, Reading, MA 1990.

Steven V. Earhart (Editor), UNIX Programmer's Manual, Holt, Rinehart, and Winston, New York, NY 1986.