(2012)

Using Graph Theory to Analyze and Detect Deadlock*

Zhikai Wang http://www.heteroclinic.net

Montreal, Quebec, Canada e-mail: wangzhikai@yahoo.com *url:*

Abstract: We try to give an algorithm about runtime deadlock detection. We also try to use graph theory to analyze the underlying mechanism.

Keywords and phrases: Runtime deadlock, Algorithm, Graph theory.

Contents

1. Introduction

So far in our coding project, we tried to produce deadlock, detected it and improvised some exception handling. In this article, we use very basic graph theory knowlege to explain our approach. This article is not peer reviewed. The previous coding projects are not peer reviewed either. If you have any comments or want to point out any errors, please contact the author. In the following section, we try to formalize our approach by giving an algorithm. By doing so, we are not limited to the confining box of a particular programming language.

*This latex file is built as per latex template at http://www.e-publications.org. Any dispute upon the usage of the above template, please contact the author.

2. An Algorithm for Runtime Deadlock Detection

First, we set some pre-requisites. A thread can be considered as a process. A resource can be considered as an object for some particular language.
- A thread has a unique ID.
- For all threads, each of them has a data structure of the resources it locks.
- Each resource also has a unique ID.
- A thread can indicate its status, e.g., waiting status. Other threads can read its status. This thread records the resource it is waiting for.
- A thread can lock multiple resources.
- A resource can be locked by only one thread. Each resource has a data structure recording which thread is holding the lock of this resource. It is well-known, or desired if not, that each resource knows the thread tries to enter waiting status for this resource.

Algorithm 1 Runtime Deadlock Detection

input: Thread T_0 and a resource R_0 which T_0 is trying to acquire/lock.
output: N/A. The program has two possible results. T_0 enters into a waiting status. Or the program enters to exception handling to avoid a deadlock.
$i \Leftarrow 0$
while R_i is locked (by other threads) **do**
  Get T_{i+1} from R_i and T_{i+1} is the locking thread of R_i.
  if T_{i+1} is in waiting status **then**
    for each resource R_x locked by T_0 **do**
      if R_x is equal to R_y, (R_y is the resource T_{i+1} is waiting for) **then**
        go to EXCEPTION:
      end if
    end for
    $i \Leftarrow i + 1$
    R_i be the resource T_i is waiting for
  else
    break from **while** R_i is locked
  end if
end while
while R_0 is not free **do**
  T_0 wait for R_0
end while
go to END:
EXCEPTION: some exception handling to avoid deadlock
END:

It will be an ideal situation that all the threads involved won't change their status, or, the operation of the above algorithm be atomic. All threads involved should be in waiting status. It is obvious their status, if changed, will release more resources for the whole system and improve the situation. We also assume there is no signal missing, which in practice can be handled by time-out waiting and poll resource status boolean.
In the following sections, we are going to use very basic graph theory to analyze deadlock and the approach we take in this article and previous coding project.

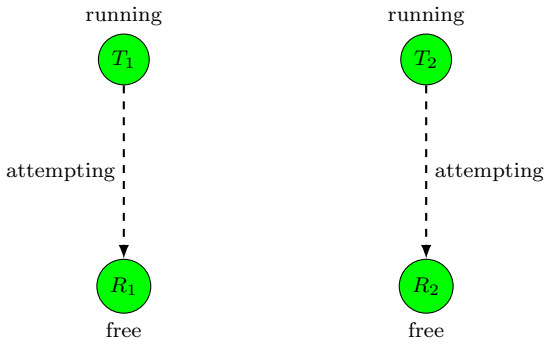

FIG 1. *Threads attempting resources.*

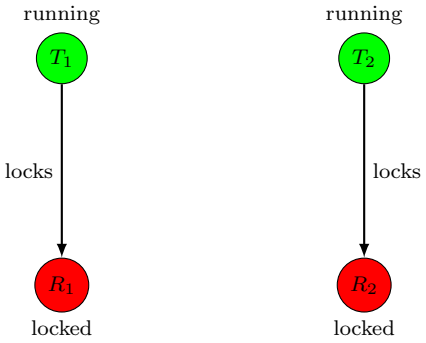

FIG 2. *Resources locked.*

3. Deadlock fomation

We use the Figure.(1) to Figure.(6) to show how deadlock is formed. These diagrams are quite self-explaining. Please stare at these diagrams for some minutes to see if there are some hidden messages. Now we raise some very very basic questions related to graph theory. Please notice we emphasize there is no advanced graph thoery knowledge involved!
First, what type of graph are these graphs?
Second, does graph in Figure.(6) provide enough information about deadlock formation?
Third, what is wrong or in-sufficient, of the graph in Figure.(6) in detecting deadlock?

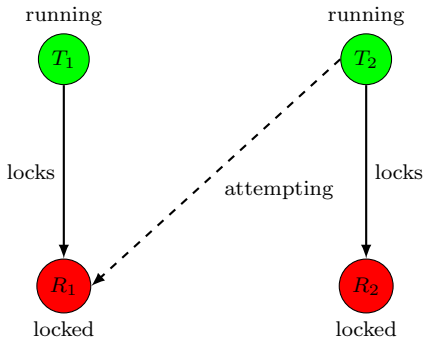

FIG 3. *T_2 attempting R_1.*

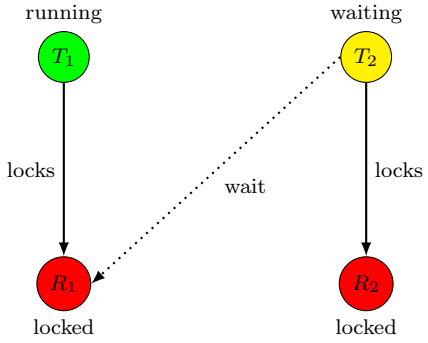

FIG 4. *T_2 waiting (for R_1).*

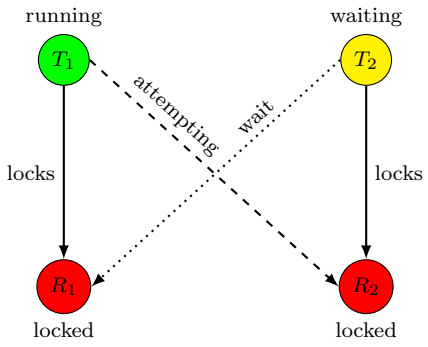

FIG 5. *T_1 attempting R_2.*

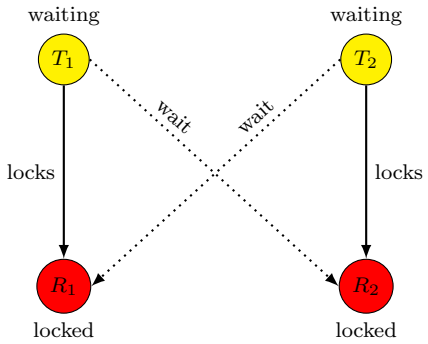

FIG 6. *T_1 waiting (for R_2), deadlock formed.*

4. Deadlock evasion

The graph show in Figure.(6) is a directed graph. It does show the information about the the formation of a deadlock. But for such directed graph, it is not connected. We can find two isolated parts. Simply put, starting from any node, we can not traverse the whole graph. So we understand deadlock for a system happens because we can not capture a whole picture of all the nodes involved. So we have to revise the system's architecture so a topological structure can navigate the confused threads like a lighthouse or the traffic signals. Now, we present the approach we take in our previous coding projects.

For a healthy system without deadlock yet, the hazard is raised when a running thread tries to acquire the resource already locked by another thread. To enter waiting status is still safe, when the threads resources dependency graph does not form a closed chain. Our threads/resources topological structure will provide sufficient information such that before a thread enters into waiting, deadlock can be detected.

Figure.(7) to Figure.(12) show exactly what we did in our previous coding project and what we propose and formalize as an algorithm in this article.

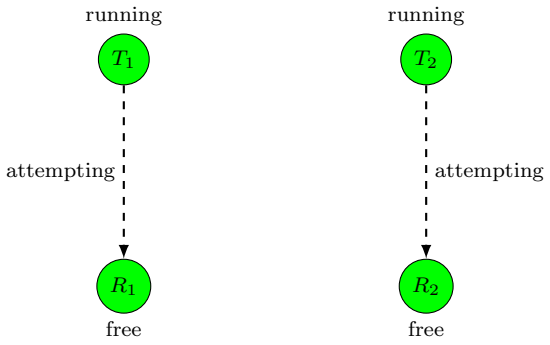

FIG 7. *Threads attempting resources.*

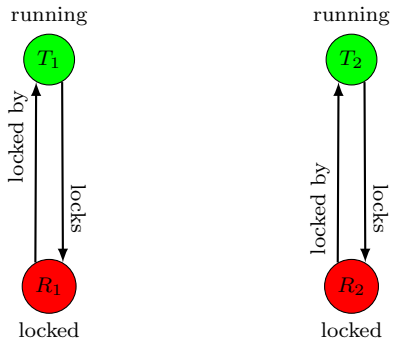

FIG 8. *Resources locked, but we have bi-directional edges.*

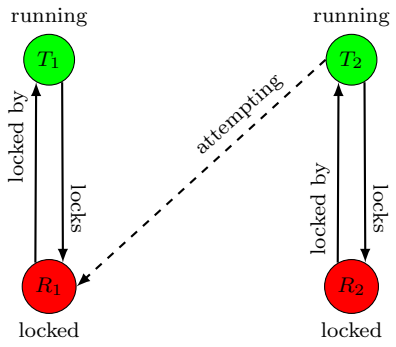

FIG 9. *T_2 attempting R_1.*

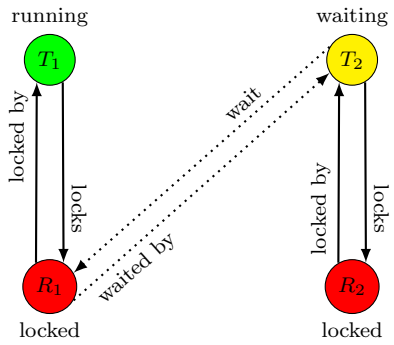

FIG 10. *T_2 waiting (for R_1).*

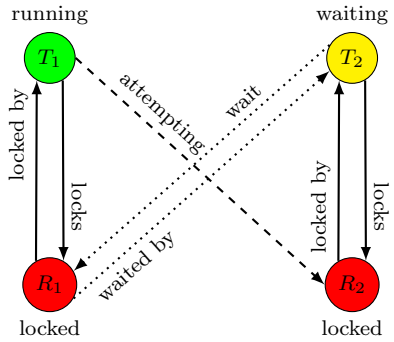

FIG 11. *T_1 attempting R_2, but T_1 has sufficient information about the local ecological chain. Namely, T_1 can traverse the graph with all the nodes involved.*

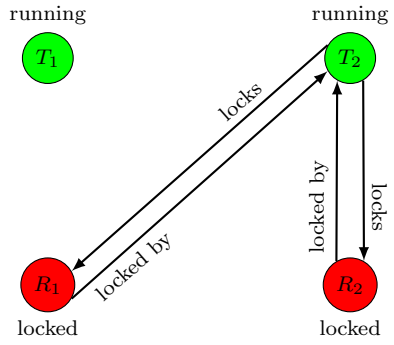

FIG 12. *In our approach, the late thread yields to the early thread. So T_1 releases R_1. T_2 revives. The global system stays healthy.*

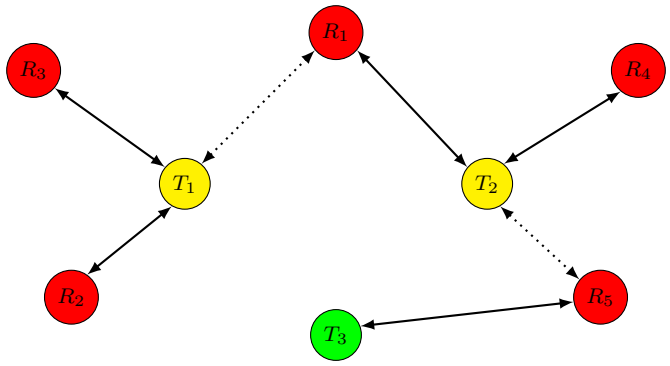

FIG 13. *The initial state of the eco-system (local). T_3 is the hope of the system. When it finishes, it will release critical resource. The system will get a new hope and keep fit.*

5. Discussion

Now we consider a more complex situation. The coding project of this situation can be a good assignment. The dining philosopher problem can also be an example to practice the algorithm we propose.

In the threads/resources lock/wait chain, if the last thread is running, so we consider such a system is health. The last thread is running, then the system has a *hope*. There will be the chance that the running thread finishes and releases some locks so other waiting thread(s) may be revived.
Figure.(13) shows the initial state of the eco-system (local). T_3 is the hope of the system. When it finishes, it will release critical resource. The system will get a new hope and keep fit. In Figure.(14), T_3 is attempting R_3. But according to the established rule, it must yield to avoid deadlock. In Figure.(15), T_3 yielded and T_2 becomes the new hope of the system.

We make a metaphor here. In the highway, the administration may deploy patrols densely so every vehicle is under constant surveillance. This is like running a monitoring thread. In our approach, we train the drivers to follow the well-know traffic rules and also make sure they know how to check signals from others and signal their own intention to the others properly.

References

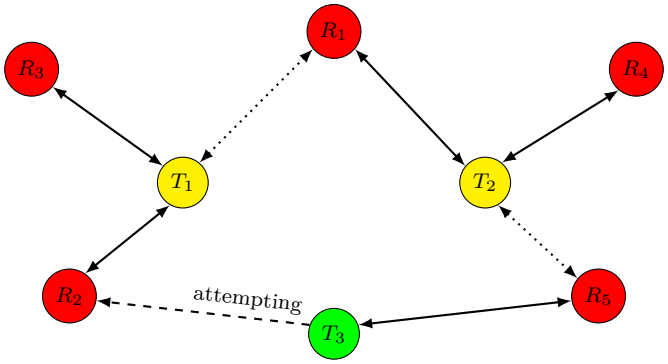

FIG 14. *But now, T_3 is attempting R_3. But according to the established rule, it must yield to avoid deadlock.*

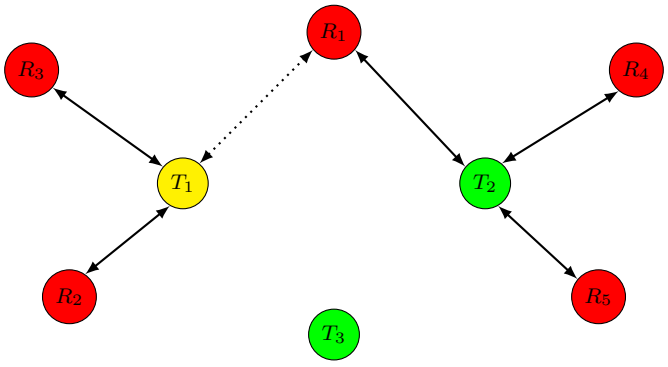

FIG 15. *T_3 yielded and T_2 becomes the new hope of the system.*