Resource Allocation Graphs

Roger Henriksson
Department of Computer Science
Lund University

The possibility of deadlock is an undesired property (to say the least) of safety-critical real-time systems. Therefore, a method to verify that a system cannot deadlock, or to detect possible deadlock situations, is of tremendous value. One such method is *resource allocation graphs*. As stated in *Operating System Concepts* by Peterson and Silberschatz [PS85], the occurrence of deadlock requires among other things that hold-wait situations exist and a circular chain of such hold-waits must exist. A resource allocation graph attempts to graphically illustrate all the hold-wait situations in a system. In this graph it is possible to search for cases of circular hold-wait.

In their book, Peterson and Silberschatz [PS85] (Section 8.2.2) introduce a method for drawing resource allocation graphs. However, in their version the resource allocation graph shows all the hold-wait-states that exist at any one given point in time. This is a good tool for illustrating a transient state in the system, but in order to use their version of the graphs for detecting any possible deadlock situation we would have to draw a graph for each and every possible combination of execution state for all threads in the system. Then, each and every one of these graphs would have to be analysed for cycles indicating circular wait. This is clearly unpractical.

Instead, we modify the method of drawing resource allocation graphs slightly in order to display a static view of the system showing every conceivable hold-wait state at any time simultaneously. This paper discusses how to do this.

Graph construction

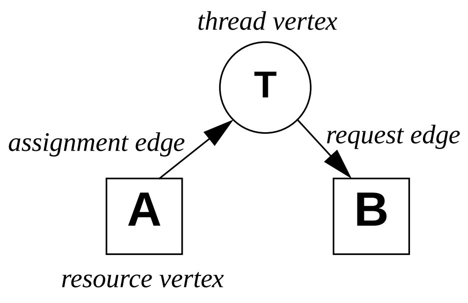
A resource allocation graph is a directed graph consisting of a set of vertices and a set of edges. The vertices come in two flavours, *resource vertices* and *thread vertices*. Each semaphore or other resource used for achieving mutual exclusion corresponds to a resource vertex in the graph. A thread vertex corresponds to *one* possible state for *one* thread in which a hold-wait situation can occur. There are also two types of edges, *request edges* and *assignment edges*. A request edge is a directed edge from one thread vertex to a resource vertex. Edges going in the other direction are denoted assignment edges. A request edge illustrates that the thread in question at a particular point in its execution tries to achieve a lock on the resource the edge leads to (e.g. executing *take()* on a semaphore). An assignment edge indicates that the resource from which the edge originates is held by the thread indicated by the edge destination. Resource vertices are graphically represented by squares labeled with the name of the resource and thread vertices are represented with circles labeled with the name of the thread. Edges are drawn as arrows from one vertex to another.

Since a deadlock is caused by a circular structure of hold-wait states, we only add possible hold-wait situations to the graph. A thread that tries to allocate a resource but does not hold any resources previously cannot cause deadlock. Thread vertices should therefore always have at least one incoming assignment edge and one and only one outgoing request edge. The graph is constructed by walking through the code of the various threads line for line and when a situation is encountered in which the thread attempts to lock a resource while already holding one ore more resource a new thread vertex is added to the resource allocation graph illustrating the possible execution state of the thread.

Below, a short code segment and the corresponding resource allocation graph is shown. The example shows a thread *T* locking two semaphores, *A* and *B*. The thread vertex marked “T” is added to the graph when analysing line 2 in the code. Here, the thread attempts to lock semaphore *B* while already holding semaphore *A*. Note that the *take()* call at line 1 does not affect the resource allocation graph since no resources were held previously by the thread.

T

```
1. A.take();
2. B.take();
3. A.give();
4. B.give();
```



The resulting graph is analysed for the occurrence of circular structures. If such a circular structure is found, it indicates a possible circular wait and thus a situation in which deadlock could occur.

Example

Consider a simple real-time system implemented using two threads, in the following denoted *T1* and *T2*. The two threads use a total of four semaphores, denoted *A*, *B*, *C*, and *D*, in order to achieve mutual exclusion over sensitive pieces of code. The semaphore operations executed by *T1* and *T2* respectively during each invocation is listed below.

T1

```
1. A.take();
2. B.take();
3. C.take();
4. C.give();
5. B.give();
6. A.give();
```

T2

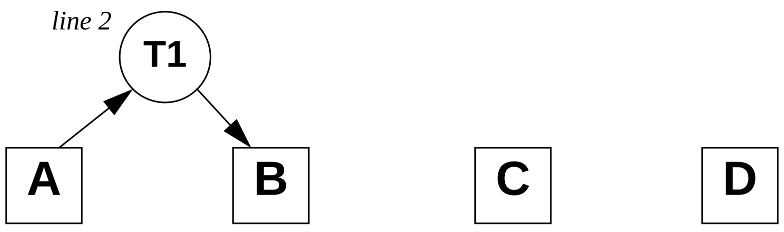
```
1. D.take();
2. C.take();
3. B.take();
4. B.give();
5. C.give();
6. D.give();
```

Can the system possibly deadlock?

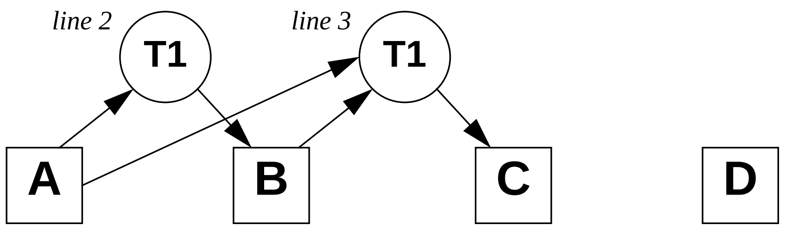
In order to answer the question we have to draw a resource allocation graph for the system. We start by drawing vertices for the resources (in this case semaphores) involved. Here, this means the semaphores *A*, *B*, *C*, and *D*, as depicted below.

We now walk through the code of each thread in order to find, and mark in the graph, each possible "hold-wait" state the threads can enter. We start with *T1*.

T1 might be blocked when it attempts to take the semaphore *A* at line 1, but since *T1* does not previously hold any resource we do not have a hold-wait situation. The thread might have to *wait*, but it does not *hold* any resource. At line 2 on the other hand, *T1* holds *A* and might have to wait for *B* to be released. *T1* can thus be in a hold-wait state at line 2. We update our graph accordingly with a vertex showing the possible hold-wait state. For pedagogical reasons we also augment each vertex in this example with the line number it is associated with in the code.

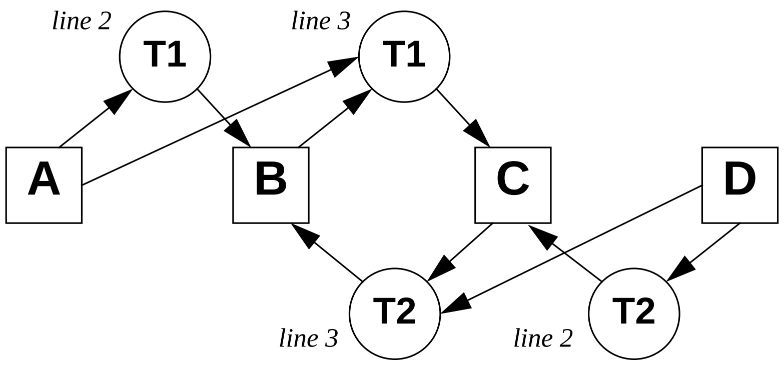


Continuing to line 3 we find that *T1* might be blocked when it attempts to take *C*. At this point, *T1* holds both *A* and *B*. We therefore draw a new vertex illustrating the potential hold-wait state of *T1*. Note that since *T1* holds two resources, there should be two assignment edges to the new vertex, one from *A* and one from *B*, as shown below.



At line 4, T1 releases semaphore C, but we do not introduce any new hold-wait situation since executing *give()* on a semaphore can never block. The same goes for the line 5 and 6.

We now turn our attention to thread *T2* and walk through its code in the same way adding all hold-wait situations we find to the resource allocation graph. After doing this we end up with the following graph:



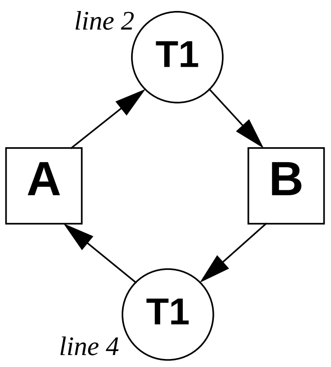
After having finished the graph we search it for cycles. Looking carefully we find one cycle going from *B* to *T1(line 3)* to *C* to *T2(line 3)*. This shows that the real-time system given in the example can indeed deadlock. This occurs if *T1* attempts to take *C* at line 3 in the code at the same time as *T2* attempts to take *B*, also at line 3.

False cycles

It is important to beware of false cycles when analysing a resource allocation graph. Consider the following code and the resulting resource allocation graph:

T1

```
1. A.take();
2. B.take();
3. A.give();
4. A.take();
5. B.give();
6. A.give();
```



Can this system deadlock? Well, obviously not since the system only consists of one thread and it never tries to take an already taken semaphore. Even so, the graph does contain a cycle! How do we explain this?

The key to the answer is that each thread vertex (circle with a thread name in it) in the graph represents *one possible state* the indicated thread can be in. Since a thread can only be in *one and only one* state at any given time, circular wait is not possible in the example above. T1 can be in the upper state in the graph *or* in the lower state, but not simultaneously in both which would be required for deadlock to occur. We should therefore discard any cycle in which every thread vertex does not correspond to a unique thread instance. Another way to put it is that any one thread instance must only occur at most once in the cycle.

Multiple instances of the same thread class

So far, we have considered systems where each thread has a unique implementation. For each thread instance we have walked through the code and marked every potential hold-wait situation in the resource allocation graph. We have learned that a cycle in the graph indicates a potential deadlock, but we have also been warned about false cycles. But let us return to the previous example which produced a false cycle for a moment.

Let us assume that the code given in the example is not the code executed by a single thread instance but rather the code found for example in the *run()* method of a Java Thread class named T. Let us furthermore assume that the main program creates two instances of this class. We say that we have one thread *type* but two thread *instances*.

One way of dealing with this situation would be to treat the two thread instances as two completely separate threads when drawing the allocation graph. This solves the problem, but we risk cluttering the graph with a large number of identical thread vertices.

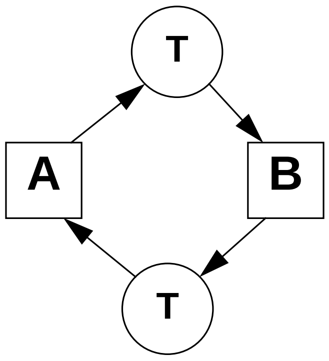
An alternative method is to draw the resource allocation graph as if we only had one thread (of each type), but to take special care when analysing the resulting graph. In our example, we then get the same resource allocation graph as earlier:

T

```
class T extends Thread {
  ...
  void run() {
    A.take();
    B.take();
    A.give();
    A.take();
    B.give();
    A.give();
  }
}
```

Main program

```
Thread t1 = new T();
t1.start();
Thread t2 = new T();
t2.start();
```



In this case we have a true cycle in the graph. This is due to the two instances of T. It is quite conceivable that one instance of the thread is in the state indicated by the upper thread vertex at the same time as the other thread instance is in the state corresponding to the lower thread vertex with deadlock as a result.

The rule when drawing resource allocation graphs in this way is that a cycle indicating deadlock may include duplicate thread vertices, *but only as many as there are thread instances of the corresponding thread type*.

Conclusion

When drawing a resource allocation graph it is thus important to differentiate between thread instances (Java objects) and thread types (Java classes) and treat them accordingly. Do not confuse thread instances with thread types!

Reference

[PS85] J. L. Peterson and A. Silberschatz, "*Operating System Concepts*", second edition, Addison-Wesley, 1985.