Inheritance Locks

The protocol’s idea is strongly influenced by Chapter 7 -Deadlocks- of the “Operating System Concepts” book by Abraham Silberschatz, Peter B. Galvin and Greg Gagne. Moreover, I shall point out three more important papers for the protocol:

1. Holt 1972 Some Deadlock Properties of Computer Systems, Computing Surveys, Volume 4, Number 3, pages 179-196

2. Coffman et al. 1971 System Deadlocks, ACM Computing Surveys, Volume 3, Issue 2, Pages: 67-78

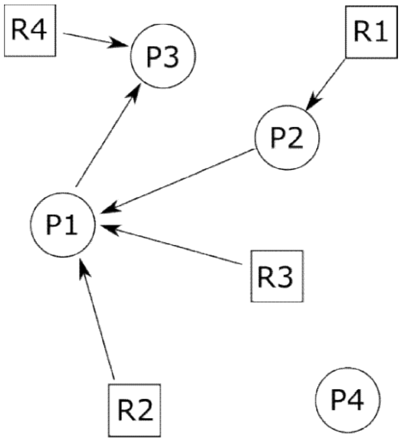
3. Dijkstra 1965 Cooperating Sequential Processes, Technical Report, Technological University, Eindhoven

These papers essentially introduced the resource-allocation graph, the 4 essential properties of a deadlock and the dinning philosopher problem respectively.

The authors introduce in “Operating System Concepts” the following protocol: “If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resource the process is currently holding are pre-empted. [...] The pre-empted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. “. Applying this protocol to the dinning philosopher problem, however, shows that it suffers from starvation. Namely, as long as one of the neighbours is eating a philosopher has to wait and therefore has to give up all the forks of her possession. Hence, the other neighbour might start eating and so on and so forth.

I would like to redefine this protocol such that: A waiting philosopher only gives up her fork if she is waiting (directly or indirectly) for the requester to give up a resource -i.e. a fork-.

Let me illustrate this protocol with an allocation graph (note that each resource has only one instance -as it would be the case with a mutex-):



P2 is currently holding R1 and waiting for one of the resources hold by P1 (R2 or R3). P1 is waiting for resource R4 which is currently held by P3.

P3 can therefore freely use R1, R2, R3 and of course R4. On the other hand, P4 can not do this as none of the processes are waiting for P4 (directly or indirectly). Let’s assume P4 would like to lock R1: In this case, it would have to wait for P2 and hence position itself into the waiting tree for P3.

In fact, it is quite easy to see that we would always have trees and never a cycle in the allocation graph. In the root positions there would be running processes which can use any resources contained within their tree.

There are two clear disadvantages of this protocol though: Whenever we try to acquire a lock, we risk losing integrity of any critical region we might be in. For example: We might acquire a lock for a database table and read from it. Next we try to acquire a second lock for a different table (we might therefore give up our first lock temporarily). Once we are holding the second lock, it would be a mistake to assume that the read data from the first table is still the same! The second disadvantage: We are required to lock and unlock resources in a strict Last-In-Firs-Out order (a stack). Let's consider the given example without this limitation: Imagine P3 starts using R2 and then gives up R4 (before releasing R2). This could lead to starvation of P1 (and P2) as R4 could easily be taken by another process by the time P3 has finished with R2.

During my project I would thoroughly investigate these kinds of limitations.