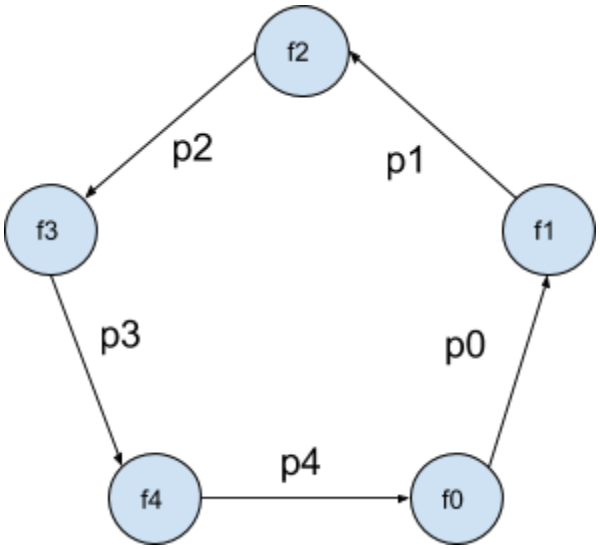
Luke Jaffe
Operating Systems
Due 03/30/16

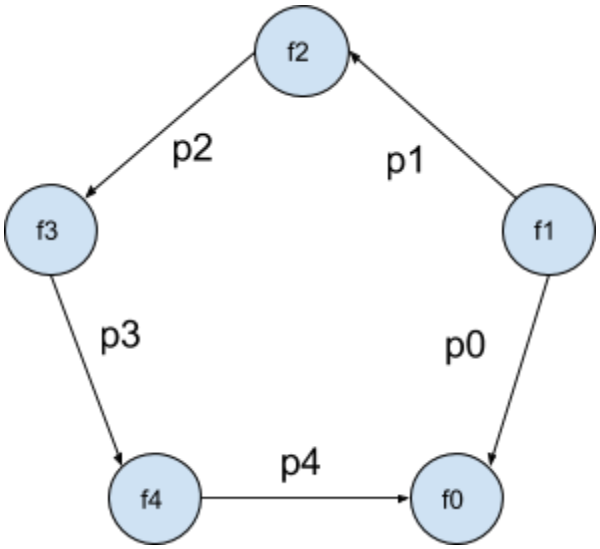
Homework 7

Question 1.

Original resource dependency graph:



This has the potential to cause a deadlock since there is a cycle in the resource dependency graph. To eliminate this cycle, we can reverse the direction of a single arrow. In the code, this can be implemented by having one philosopher pick up his forks in the opposite order of the others. If philosopher zero does so, the new resource dependency graph will be as follows:

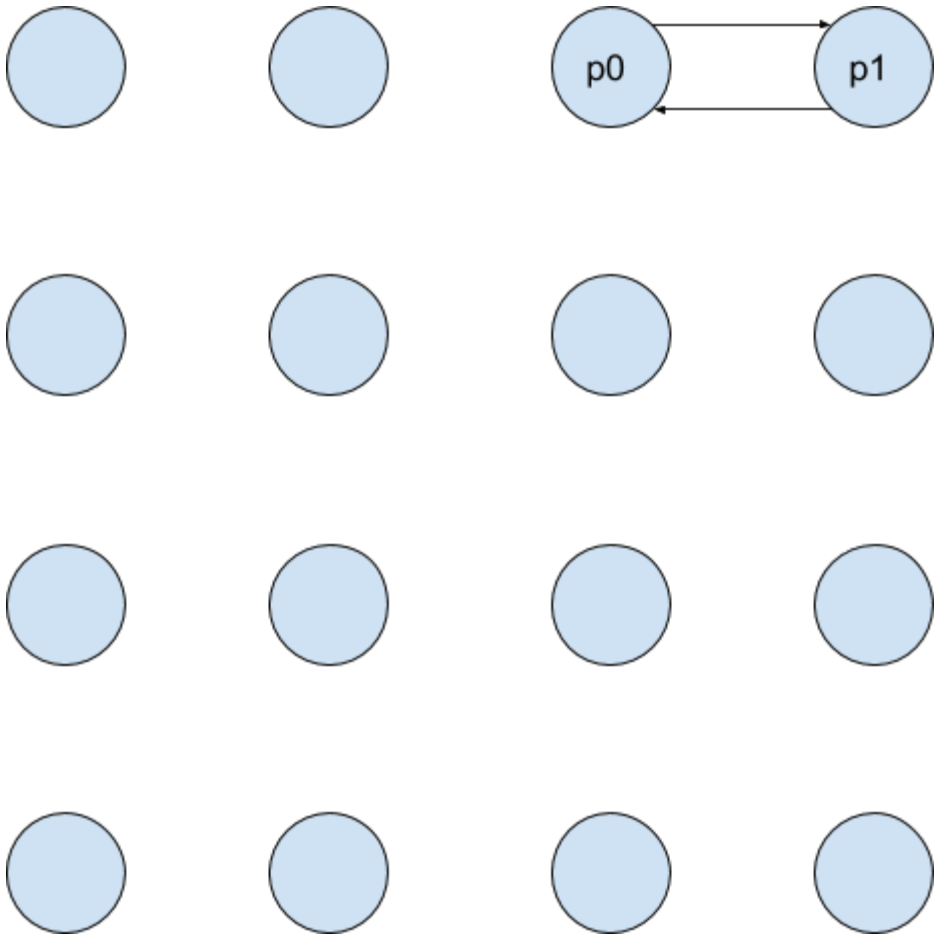


Question 2.

b) A deadlock scenario occurs when when two packets are trying to “trade” nodes. That is, each is trying to occupy the current position of the other. This means that each has got stuck trying to acquire the lock of the new position, when the other holds that lock.

Code output:
packet: 0, pos: (0, 1), next: (0, 2)
packet: 0, pos: (0, 2), next: (0, 3)
packet: 0, pos: (0, 0), next: (1, 0)
packet: 0, pos: (1, 0), next: (2, 0)
packet: 0, pos: (2, 0), next: (3, 0)
packet: 1, pos: (3, 0), next: (2, 0)

c) Nodes in the grid are represented by circles. Current position of a packet is indicated with a label on a node, and an arrow is drawn to show where the packet in that node is attempting to go. The arrow also represents resource dependency. This example is done for the code output above. The cycle is between two nodes, which are each dependent on the other.



d) For the scenario with two in-flight packets, a number of different restrictions could be imposed to prevent a deadlock scenario. One possible restriction would be to have one packet “yield” to the other. Each time one packet is about to calculate a new position, before it does so, it would check the current and next position of the other, and make sure that the two are not about to swap. If they are not, it would continue as normal. If they are about to attempt swapping, it would randomly pick a new valid next position, and go there instead. Then it would continue towards the destination again in its next loop.

This would require some additional elements to be implemented properly. First, the current and next position of the “dominant” packet would need to be shared by both threads, and protected with a mutex. Second, the dominant thread would need to wait for the recessive thread to jump to its random new position before continuing (in the swap scenario). Otherwise, the dominant thread might immediately trigger another swapping scenario (timely context switch) right after the recessive thread has picked its new destination. This problem could be solved with a condition variable, positioned so the dominant thread waits for the recessive thread to make the jump before continuing.