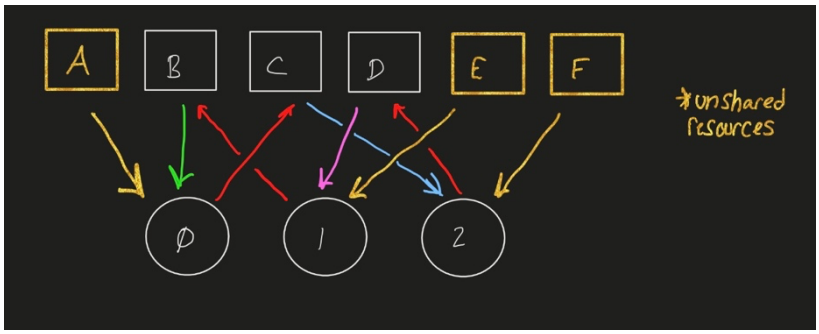
George Martin
COMP 3500 – HW2

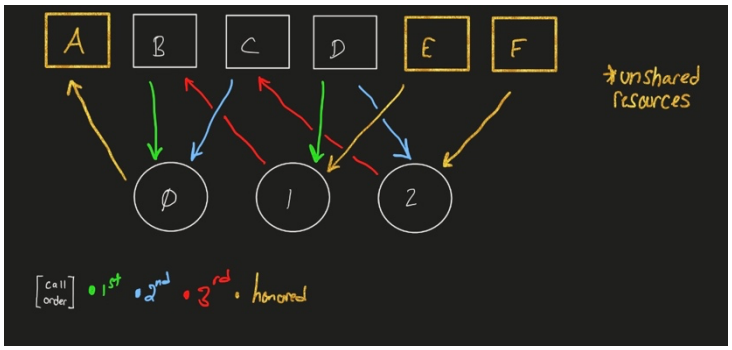
1a. A deadlock exists in this graph where:
P(0) – acquires A and B
P(1) – acquires D and E
P(2) – acquires C and F

In this case, P(0) would be waiting on resource C which is held by P(2). P(2) would be waiting on resource D which is held by P(1), and P(1) would be waiting on resource B which is held by P(0). Here we have formed a cycle and are now in a hold&wait deadlock position. Note: A, E, and F are not shared, but are required for the processes to complete their critical section.



1b. By changing the order, we can mitigate a chance of a deadlock during a concurrent run.

P(0)	P(1)	P(2)
(B)	(D)	(F)
(C)	(E)	(D)
(A)	(B)	(C)



By changing the order to allow the non-shared resources to be pulled at different intervals in each processes, we have eliminated the chance of a deadloc. In any order, the following should work, with the main focus being on the order of P(0) and P(2). If any process ran through its entire sequence first, the issue would be resolved, but if they executed sequentially or in a random order, we would want to ensure one process would be focused on competing for two shared resources while the other two only compete for one initially. The way P(0) has been ordered, it would get resource B regardless of the outcome, even if P(1) ran through it's entire code it eliminate a deadlock for P(0). P(0) would now only need to be concerned with resource (C) since resource (A) is unshared. With P(2) calling for resource (C) as its final get() call, (C) would either go to P(0) first or immediately after P(2). Either way, this prevents a deadlock. P(1) could get (D) or be initially blocked, but if it were blocked there wouldn't be a deadlock. If it got (D), then (E) being unshared is always honored anyway. It would only have to wait on (B), which would either be immediately honored or used by P(0) which we have already proven would not be deadlocked with P(2). In this way, no cycle exists that would create a deadlock.

George Martin
COMP 3500 – HW2

2. Yes, the concurrent running of these processes could result in one or more being blocked forever. This would happen if both ran their initial wait functions concurrently:

foo: semWait(S)
bar: semWait(R)
foo: semWait(R) - blocked
bar: semWait (S) - blocked

In this situation, x would never be modified since there is now the smallest instance of a hold&wait cycle with no preemption or mutual exclusion. Signal would never be called freeing the resouces.

3. Deadlock prevention, avoidance, and detection are all methods that software engineers can handle potential deadlock scenarios. In deadlock prevention, a static approach, the software engineer ensures that a function does not allow competing processes to indefinitely hold a resource. In this sense the software engineer must ensure that at least one of the cases does not hold: mutual exclusion, hold & wait, no preemption, or circular wait. However, deadlock prevention, though ideal is not always practical.

Deadlock avoidance, another static approach, handles deadlocks in a more practical sense. Instead of focusing on process conditions it's primary focus is on system states. In deadlock avoidance, the goal is to keep the system in a "safe state" as often as possible, limiting the risk of running in an "unsafe state" which has the potential to drop into a "deadlock" state.

Finally deadlock detection (a dynamic approach) throws caution to the wind and goes for gold (joking). In deadlock detection the system monitors processes and resource instances. If it detects a deadlock situation it can approach it in two categories. In the single instance of each resource type it would fall into a detect cycle in wait, while in a situation with multiple instances of resource types, it would apply a safety algorithm. In either case, to recover from this deadlock the system will attempt to abort one or all deadlock processes, and/or use resource preemption or a rollback.

4. We can prove a system of 4 resources and three processes is deadlock free by use of proof by contradiction. If we assume the system is in a deadlock, then we should be able to find a case where resources are each being held in a way that would not allow any process to complete. If we assume each process picks up a resource, there would be 1 resource left over, with all processes currently waiting for 1 more resource to complete and free up their held resources. The moment a process acquires the 4^{th} resource, at least one process is able to run, ultimately freeing up [2] resources and allowing the other two processes to run. In this case, a deadlock situation is impossible, and we have proven the system to be deadlock free. ■ <-