21. Deadlock Management

In this chapter, we discuss more in detail what deadlocks are, how they can occur, and what are possible strategies to prevent the occurrence of deadlocks, or to recover a system after a deadlock has occurred.

21.1 Deadlocks

We have discussed in previous lectures that one of the key aspects of resource sharing is synchronization: different processes should "agree" on which process (or processes) should be allowed inside the critical section. We have noticed that under certain circumstances, processes simply cannot agree on who should go next, leading to starvation. One such case occurred when we considered flawed approaches to achieve mutual exclusion. A second example was in the context of the dining philosophers problem.

Fortunately, we were able to derive a "patch" to the mutual exclusion protocol that prevented deadlocks *by design* when accessing a **single resource**. Even with a correct mutual exclusion protocol, however, deadlocks still occurred in the dining philosophers problem. As it turns out in the latter case, the root of all the evil is that processes can share and synchronize access to **multiple resources**.

In the problem of the dining philosophers, each philosopher/process has *access* to two chopsticks/resources, and needs to synchronize with his two neighbors over the use of such resources. For this particular problem, we were able to identify an elegant solution that leverages some knowledge of the problem setup itself. But what if that knowledge is not available? What if the only thing is given to us is a general statement of the type: "process P and Q intend to use resources R_1 and R_2 at some point and in some way"? Clearly, we need to find a systematic way to solve the deadlock problem.

21.1.1 How Deadlocks Occur

A deadlock can occur when a conflicting pattern in the use of multiple resources arises among competing processes. A deadlock can be defined according to Definition 1.

Definition 1 Permanent blocking of a set of processes that either compete for system resources or communicated with each other.

From the definition above, it follows that a deadlock involves more than one (potentially many) processes, and that a deadlock condition puts the locked processes in a state where no progress is made. Now, modern computing systems are normally structured with thousands of competing processes, and a large number of shared resources. As such, if there was no general solution to the problem of deadlocks, nothing would work. It follows that a general solution *must* exist out there and that we are here to discuss it. Unfortunately, **NO**.

As we will see in these lecture, there exist *some* approaches that attempt to systematically solve deadlocks, but these solutions have drawbacks that make them not very useful in practice. So how computers still work today? The answer is: careful programming. The best weapon to date against deadlocks is **knowing how they can occur**, and prevent them from happening in your code as much as possible.

So, how can a deadlock occur? Since we need multiple processes and multiple resources, let us consider the simplest scenario for a deadlock. Consider two processes, P and Q, and two resources R_1 and R_2. Can this scenario, as is, lead to a deadlock?

A first thing to consider is the type of resources for R_1 and R_2. As previously mentioned, thousands of processes at a time can share a CPU. Yet, this type of sharing does not lead to starvation. The CPU, in fact, is a type of resource that can be used in an interleaved fashion without complications. A little bit of P is done, then a little bit of Q, then some more P, and so on. Boom! This is the same for a disk, for the DRAM subsystem for a network card. Clearly, resources that can be used by multiple processes in an interleaved fashion present no risk of deadlocks.

Other resources, however, require to be **exclusively** used by a single process. In this case, different processes can still share a resource, but they have to serialize in time so that their access and use of the resource is mutually exclusive. As an example of a tangible mutually exclusive device, consider a printer. If we were to allow multiple processes to interleave on a printer, you would probably get some gibberish printout. A similar thing happens on a screen console: if different processes can output simultaneously, the result could be an unintelligible mix of characters. **Shared data structures** are yet another example of mutually exclusive resources. A generalization of a mutually exclusive resource is a resource with limited capacity.

For deadlock to occurs, processes need to share limited-capacity resources. Note that once a process P has acquired access to a limited-capacity resource, say R_1, it makes no sense to preempt P on R_1 because no other process can use R_1 anyway. Instead P needs to be "done" with R_1 before any other process can be allowed to use R_1. Thus, we have constructed 2 conditions for the occurrence of deadlocks:

1. Presence of multiple processes (≥ 2) that share multiple resources (≥ 2);
2. Resources must have limited capacity (e.g. mutually exclusive, where capacity = 1).

These two conditions however are not enough per-se to conclude that there is a deadlock. Hence, they are **necessary** but not **sufficient** conditions.

What else needs to happen for a deadlock to occur? By definition, none of the processes involved in a deadlock are able to make progress. If we only have two processes P and Q, it must mean that P is awaiting for Q to be done with some resource, while *at the same time* Q is awaiting for P to be done with some resource. But how can processes get to this situation?

Let us try to visualize the execution and progress of two processes P and Q that share two mutually exclusive resources R_1 and R_2. Clearly, there is more than one possibility in the *order* at which the two processes can acquire and release the two resources. Let us first consider one particular order, resulting from the (pseudo-)code of P and Q reported in Listing 21.1 and 21.2.

```
/* Process P's pseudo-code */
void P_main(void)
{
  /* Do something */
  acquire(R1);
  /* Do something */
  acquire(R2);
  /* Do something */
  release(R1);
  /* Do something */
  release(R2);
}
```

Listing 21.1: Process P's code.

```
/* Process Q's pseudo-code */
void Q_main(void)
{
  /* Do something */
  acquire(R1);
  /* Do something */
  acquire(R2);
  /* Do something */
  release(R1);
  /* Do something */
  release(R2);
}
```

Listing 21.2: Process Q's code.

From the code, note in particular the order R_1, R_2 at which both processes acquire and release the two resources. We can then use a plot to depict the progress of P and Q toward completion. On our plot, the progress of P is represented on the horizontal (x) axis, while the progress of Q is reported on the vertical (y) axis. At the beginning, both processes are at progress 0, i.e. at the origin of the axes. As P progresses, by say 10 instructions, while Q progresses by 20, we are at the $(10, 20)$ point on the 2D-plane. If only one of the two processes progresses, we move only horizontally (for P) or vertically (for Q).

As described in Listing 21.1 and 21.2, at some point during execution, P and Q will need to acquire or release resources R_1 and R_2. Acquisitions and releases can be reported on the progress plot in the corresponding order. To visualize this way of reasoning about the progress of processes P and Q, refer to Figure 21.1.

Since execution progresses only forward, as time goes by we either move rightward (with progress of P) or upward (with progress of Q) in the plot. Note one important feature: since R_1 is mutually exclusive, and P has acquired (and not released) R_1, then the progress of Q cannot go past the "Acquire R1" statement on the y-axis. Otherwise, it would mean that both P and Q have acquired R_1.

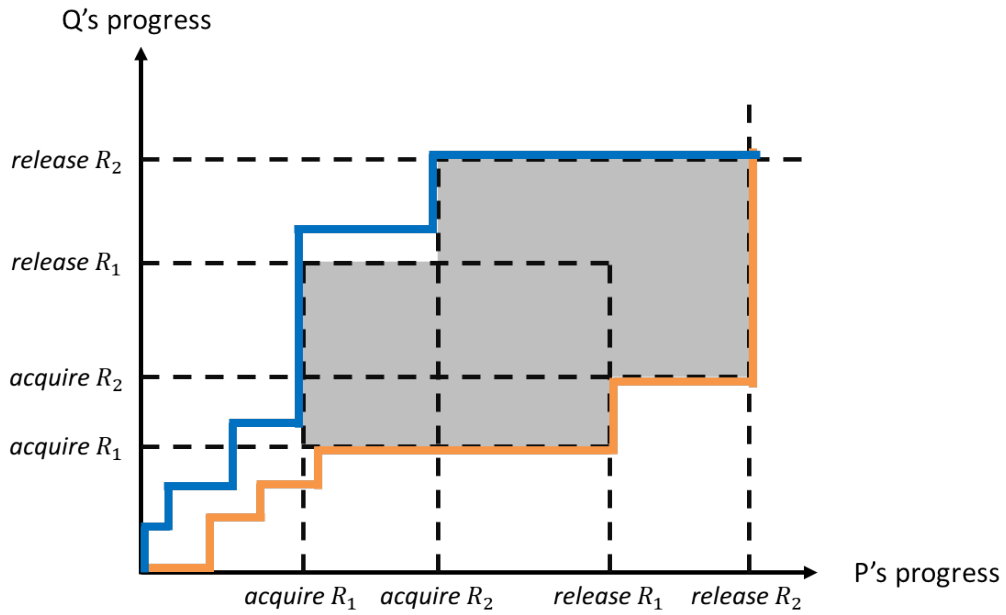

Figure 21.1: Plot depicting progress for process P (x-axis) and Q (y-axis). In this case, no deadlock is possible.

Let us look at the highlighted paths. Path 1 (in orange) corresponds to the sequence: (1) P gets R_1; (2) P gets R_2; (3) P releases R_1; (4) Q gets R_1; (5) P releases R_2; (6) Q gets R_2. Similarly, Path 2 (in blue) corresponds to the sequence: (1) Q gets R_1; (2) Q gets R_2; (3) Q releases R_1; (4) P gets R_1; (5) Q releases R_2; (6) P gets R_2. It is easy to see that there is no feasible path that can go through the area highlighted in gray. In this case, **no deadlock is possible**.

What if we change the order in which P and Q need to use the resources? For instance, consider Listing 21.3 and 21.4. For this case, the progress plot is depicted in Figure 21.2.

```
1  /* Process P's pseudo-code */
2  void P_main(void)
3  {
4    /* Do something */
5    acquire(R1);
6    /* Do something */
7    acquire(R2);
8    /* Do something */
9    release(R1);
10   /* Do something */
11   release(R2);
12 }
```

Listing 21.3: Process P's code.

```
1  /* Process Q's pseudo-code */
2  void Q_main(void)
3  {
4    /* Do something */
5    acquire(R2);
6    /* Do something */
7    acquire(R1);
8    /* Do something */
9    release(R2);
10   /* Do something */
11   release(R1);
12 }
```

Listing 21.4: Process Q's code.

The code has been changed only slightly compared to Listing 21.1 and 21.2. What could possibly go wrong? First, consider execution Path 3 (in green) in Figure 21.2. The path corresponds to the

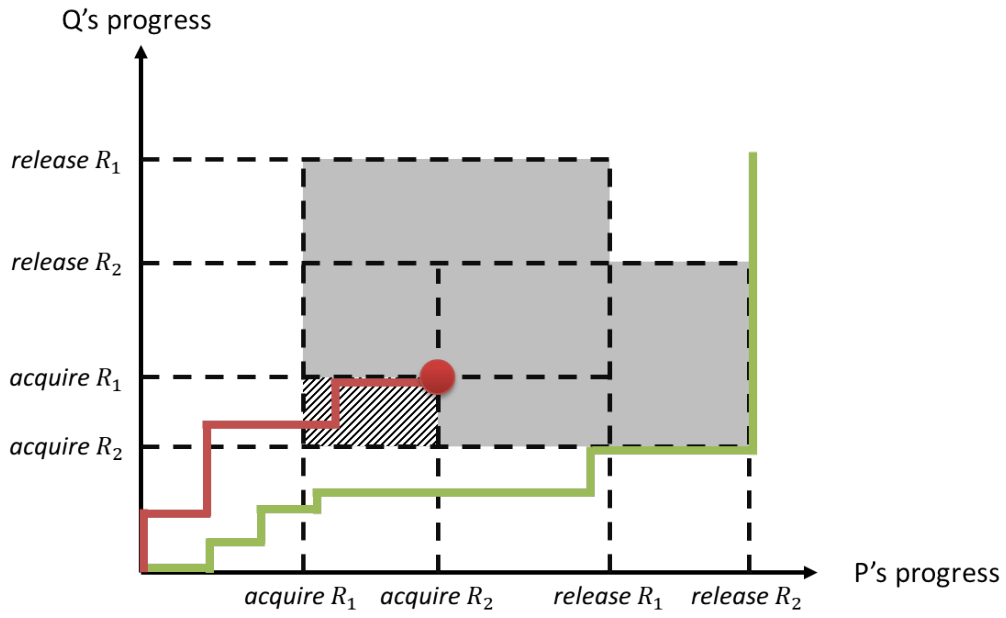

Figure 21.2: Plot depicting progress for process P (x-axis) and Q (y-axis). Deadlock occurs if P acquires R_1 but not R_2 before Q acquires R_2. Similarly, deadlock occurs if Q acquires R_2 but not R_1 before P acquires R_1 (red path).

sequence: corresponds to the sequence: (1) P acquires R_1; (2) P acquires R_2; (3) P releases R_1; (4) P releases R_2; (5) Q acquires R_2; (6) Q acquires R_1. No deadlock!

On the same figure, let us now consider Path 4, highlighted in red. In this case, we have: (1) P gets R_1; and (2) Q gets R_2. At this point, (3) P tries to acquire R_2, which is held by Q; and (4) Q tries to acquire R_1, which is held by P. This is a **deadlock**! In fact, we cannot move neither up, nor to the right. The deadlock is represented on the plot with a solid red dot. Just like in Figure 21.1, the area of the plot highlighted in gray is not feasible. Conversely the are in Figure 21.2 highlighted with a striped pattern is reachable. But here is the catch: if the combined progress of the processes results in a path within this area, the processes are doomed and will end up in a deadlock at the red dot: they have already passed the *point of no return* to avoid the deadlock. So maybe a good strategy to prevent deadlocks from happening is: (1) know where the point of no return is; and (2) prevent processes from passing it. In other words, make sure that the combined progress of the processes does not enter the striped area.

Let us look at another way of representing this situation. Consider a graph in which processes are circular vertices, while resources are squared vertices. Let us say that there exist an edge from a process to a resource if the process is requesting a resource. Similarly, there is an edge from a resource to a process if that process has already acquired the resource. The deadlock situation represented by the red dot in Figure 21.2 is reported in graph form in Figure 21.3. **A deadlock occurs when a loop develops** in the graph.

At the beginning of this section, we discussed two necessary conditions for deadlocks. We can

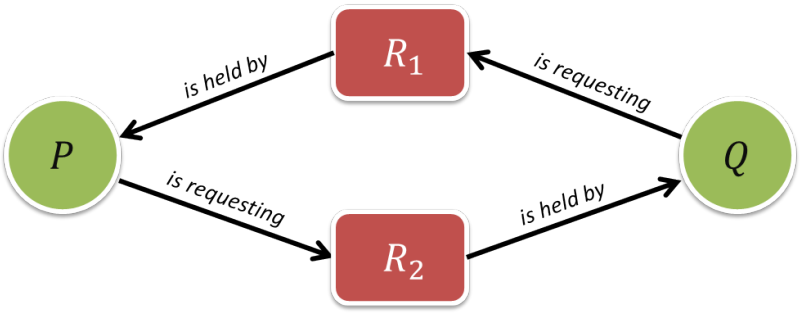

Figure 21.3: Graph of resource-to-process assignments and process-to-resource requests. A deadlock corresponds to a loop in the graph. This deadlock corresponds to the red dot in Figure 21.2.

now complete the list and produce a list of *necessary and sufficient* conditions for the occurrence of deadlocks:

1. Presence of multiple processes (≥ 2) that share multiple resources (≥ 2);
2. Resources must have limited capacity, wtih no preemption allowed once a process has acquired the resource;
3. A process may have acquired resources while awaiting to acquire additional resources (hold & wait);
4. A closed chain of dependency develops in the process-resource graph, meaning that each process in the chain holds at least one resource required by the next process in the chain (circular wait).

Note that condition 4 alone would not necessarily lead to a deadlock if conditions 1-3 did not hold for the considered system.

21.1.2 Deadlock Management

There are effectively three class of approaches to deal with deadlocks: (1) deadlock **prevention**; (2) deadlock **detection and recovery**; and (3) deadlock **avoidance**. Let us take a closer look at these approaches.

Deadlock Prevention

With deadlock prevention, the main idea is to construct a system in a way that deadlocks are not a possibility *by design*. We have constructed a set of necessary and sufficient conditions for deadlocks to happen. It would be enough to make sure that one of these conditions does not hold, and deadlocks would be just a bad, yet forgotten nightmare.

For instance, even if a circular wait develops, we could break the chain. In case of Figure 21.3, we could preempt P on R_1, or Q on R_2. This possibility would mean that Condition 2 above does not hold. Unfortunately, this would expose use to resources that can be corrupted, in an inconsistent state, and/or gibberish (remember the printer?) data being produced.

Another option is to violate Condition 1 by either having less than 2 processes, or by limiting the amount of shared resources between processes. This would mean that processes would have strong limitations when it comes to inter-process communication and parallel progress.

The same reasoning can be applied for all the conditions 1-4. It follows that preventing deadlocks in first place is not a practical solution. Or in other words, it represents a solution that is worse than the problem.

Deadlock Detection & Recovery

Another approach is the following: let processes acquire resources optimistically. Next, check for the occurrence of deadlocks. If a deadlock occurs, identify the processes that are part of the circular waiting chain and abort of the offending processes. This is actually pretty hard to achieve due to two main issues. First, we need to have a way to detect that a group of processes is not making any progress. Second, we need to be able to reconstruct the dependency graph to find the processes involved in the circular wait. There could be thousands of processes in a system, and tens of thousands of shared resources. Moreover, the OS may not be fully aware of the existence of certain resources.

On top of that, even if we were able to identify an offending process in the circular wait, aborting the process would certainly lead to the resource being released. But there would be no guarantees on the resource being in a consistent status afterward. What if we wanted to guarantee resource consistency after an abort/kill? Then each and every process should define a graceful release-upon-kill procedure for each resource and for each point in their execution. This is a lot to ask from programmers.

Deadlock Avoidance

Deadlock avoidance is what we are left with, given that we ruled out as non-practical the other two approaches. Okay then, how do we avoid deadlocks? For instance, we could try **establishing a strict ordering on the way resources can be acquired**.

Consider what happened between Listing 21.1, 21.2, and Listing 21.3, 21.4. The only things that changed was the order in which process Q requested resources R_1 and R_2. With the first ordering, no deadlock was possible (see Figure 21.1). In the second case, deadlock was possible under certain circumstances (see Figure 21.2).

It follows that if we are able to establish a strict ordering in the way resources are requested by all the processes, then we do not have to ever worry about deadlocks. Since we know very little or noting about the specific logic of the processes, our only hope is to enforce some property on the resources themselves. For instance, for R_2, it could be a property of the type: *acquiring R_2 can only be attempted if a process already holds R_1*. In other words, the idea would be to create a global, system-wide ordering in which resources should be acquired.

It follows that:

- For each resource R_i, we assign an system-wide ordering $O(R_i)$;
- If a process requires R_i and R_j such that $O(R_i) < O(R_j)$, then the process will necessarily have to acquire R_i before attempting to acquire R_j.

It is easy to prove that with this approach, it is impossible for multiple processes to enter a circular wait.

The main drawback of this approach however is that processes will be stalled unnecessarily, waiting for resources that they need to acquire, but that they do not really need *at the moment*. It follows that this approach is essentially inefficient, albeit certainly effective.

21.1.3 The Banker Algorithm

Strictly ordering all the limited-capacity shareable resources in the systems leads to large inefficiency because: (i) the order is largely arbitrary since there is no good criteria to perform the ordering; (ii) even if a process needs multiple resources, it does not mean that acquisition requests are badly nested, or not even nested at all; (iii) a knowledge of all the shared resources in the system is required, which may not be always true for highly decentralized logic.

Let us re-think the problem of resource assignment in a different way. Instead of imposing a restriction of what resources a process can request and when, let us impose a restriction on when an acquisition request for a resource is actually granted. In all the previous approaches, once a process reaches the execution point where it requests a resource, the resource is simply granted if no other process is already holding the resource. This leads to problems. Thus, a promising strategy consists in **checking if a resource can be granted even if it is not being currently held by any process in the system**.

Clearly, the performed "check" should prevent processes from entering in a combined progress state where deadlock is unavoidable, i.e. from passing what we called the *point of no return* for deadlocks. In Figure 21.2, the point of no return was represented by the area with striped pattern. Hence, as long as the performed check can steer tasks away from entering any such *area*, no deadlocks are possible. The Banker Algorithm represents a systematic way at performing such a check, for an arbitrary number of processes and resources[1].

Some Notation

Let us first introduce the quantities used by the Banker Algorithm to perform its check. We will use the index i to refer to processes (e.g. P_i), and the index k to refer to resources (e.g. R_k).

- $R(k)$: total amount of resource R_k **present** in the system. The idea is that each resource exists in the system in a certain amount. For instance, $R(k = \text{main memory})$ =4 GB; $R(k = \text{disk space})$ =1 TB; and so on. Mutually exclusive resources have availability $R(k) = 1$.
- $C_i(k)$: total amount of resource R_k that process P_i will ever need during its execution. It can be

[1] If we wanted to represent the combined progress of N processes we should be able to visualize plots in N dimensions. That is not exactly straightforward.

though as the amount of R_k that P_i *claims* it needs to execute. This quantity is simply declared by P_i before starting its execution.
- $A_i(k)$: amount of R_k currently allocated (i.e. granted) to process P_i. Upon P_i's start, $A_i(k) = 0$ for all k. Obviously, it must hold that $A_i(k) \leq C_i(k) \leq R(k)$.
- $V(k)$: total amount of resource R_k **available** at the current time in the system. This quantity can be calculated as: $V(k) = R(k) - (A_1(k) + A_2(k) + \cdots + A_N(k))$.
- $N_i(k)$: amount of R_k that process P_i currently needs to complete its execution. $N_i(k)$ can be computed as follows: $N_i(k) = C_i(k) - A_i(k)$.

After this enjoyable notation digression, let us take a closer look at the Banker's Algorithm.

Safe States

As mentioned earlier, the key idea is to perform a check that granting a resource keeps the system in a state that does not lead to deadlocks. In other words, the algorithm's goal is to keep the system in a *safe* state: outside of any striped area. The point is determining what constitutes a safe state.

A state can be considered safe if and only if there exists a route to escape deadlocks. The escape route can be thought as follows: consider an arbitrary sequence of execution for all the N processes in the system. If for each $i \in \{1, \ldots, N\}$ it is possible to allocate enough resources so that P_i completes (and releases all the previously granted resources), then the system state is considered safe. Clearly, when a safe state is defined in this way, all the processes are able to complete if the system is in a safe state. Consequently, no deadlocks are possible.

The pseudo-code presented in Algorithm 1 presents the main logic to check the safety of a given system state, where a system state is represented by the values of $V(k), A_i(k)$, and $N_i(k)$ for each possible i and k (input). The algorithm first initializes the state of all the processes P_i to `CannotFinish` (Line 2). It also initializes a new quantity $W(k)$ to $V(k)$ for each resource R_k (Line 3). $W(k)$ is used to track the availability of R_k once it has been determined that some process can complete. The loop at lines 4-6 attempt at determining if there exists an order of execution for the processes that allows all of them to complete. In fact, if there is a process that can complete (Line 4), its state is marked as `CanFinish` (Line 5), and its resources are relinquished (Line 6). Finally, if all the processes have been marked as `CanFinish`, the current state is safe and True is returned (lines 7-8). Conversely, the state is not safe and False is returned (lines 9-10).

Resource Granting Logic

Once we know how to check if a given system state is safe, the resource granting logic of the Banker Algorithm is provided in Algorithm 2. The input to the algorithm is a request (query) Q_i made by process P_i for system resources. In Q_i, we indicate with $Q_i(k)$ the amount of R_k requested by P_i. At the beginning, the algorithm checks if the request is consistent with what declared by process P_i (lines 2-3). An error is returned if the request is inconsistent. Next, the algorithm checks if each resource R_k is at all available for allocation of $Q_i(k)$ units. If any of them is not, P_i needs to wait, hence False is returned (lines 4-5). Next, the logic considers as if the request was granted, creating a new temporary system state at lines 6-8. Next, the algorithm checks if the new temporary state is actually safe by invoking the procedure described in Algorithm 1 (Line 9). If the new state is safe,

Algorithm 1: Algorithm to check safety of a given system's state.

Input: $V(k), A_i(k), N_i(k)$ for each i,k
Output: True if current state is safe, False otherwise.

```
1  Function StateIsSafe(V(k), A_i(k), N_i(k) for each i,k)
2      P_i ← CannotFinish for each i;
3      W(k) ← V(k) for each k;
4      while P_i exists such that P_i == CannotFinish and N_i(k) ≤ W(k) for all k do
5          P_i ← CanFinish;
6          W(k) ← W(k) + A_i(k) for all k;
7      if P_i == CanFinish for all i then
8          return True;
9      else
10         return False;
```

request Q_i can be safely satisfied, hence the request is granted to P_i by updating the system's state (lines 11-13) and returning True (Line 14). If the new temporary state was deemed as unsafe, False is returned to indicate that P_i needs to wait (Line 15).

An Example

In order to clarify how the Banker Algorithm works, let us consider a given system. In the system, we there are $N = 4$ processes sharing three resources R_1, R_2, and R_3. The static system parameters ($R(k)$ and $C_i(k)$) are provided in Table 21.1. At the time the system is being consider, moreover, the dynamic system parameters are provided in Table 21.2. As a sanity check on the system's state, note that $R(k) = V(k) + \sum_{i=1}^{4} A_i(k)$ between the two tables.

Suppose that at the considered point in time, P_2 requests $Q_2 = \{Q_2(1) = 1, Q_2(2) = 0, Q_2(3) = 1\}$, or $Q_2 = (1,0,1)$ for short. Now we run the Banker Algorithm to check if the request can be satisfied or not. When we invoke the `BankerHandleRequest` function given in Algorithm 2 with parameters P_2, and Q_2. The following happens:

1. The request passes checks at lines 2-5;
2. Then, a new temporary state is created, with dynamic parameters expressed in Table 21.3;
3. Next, `StateIsSafe(...)` is invoked with the created temporary state;
4. In this case, note that P_2 will be marked as `CanFinish` in Algorithm 1 (Line 4), which triggers a new value of $W = (0+6, 1+1, 1+2) = (6,2,3)$ (Line 6).
5. It now holds that $W(k) \geq N_i(k)$ for each i,k. So P_1, P_3 and P_4 will be also marked as `CanFinish` in the following iterations.
6. Hence, `StateIsSafe(...)` returns True (Line 7-8).
7. Algorithm 2 will then grant Q_2 and update the system state as in Table 21.3.

What if the original request was submitted by P_1 instead of P_2? In other words, what if the Banker Algorithm was invoked with the same state as in Table 21.2 with request $Q_1 = (1,0,1)$? Let us follow again the same steps:

Algorithm 2: Complete Banker Algorithm.

Input: Requesting process P_i; Requested resource amount $Q_i(k)$ for each R_k.
Output: Resource request granted (True), denied (False), or inconsistent (Error).

```
1  Function BankerHandleRequest(P_i, Q_i)
2      if Q_i(k) > N_i(k) for any k then
3          return Error;
4      if Q_i(k) > V(k) for any k then
5          return False;
6      V(k) ← V(k) − Q_i(k) for each k;
7      A_i(k) ← A_i(k) + Q_i(k) for each k;
8      N_i(k) ← N_i(k) − Q_i(k) for each k;
9      safe ← StateIsSafe(V(k), A_i(k), N_i(k) for each i,k);
10     if safe == True then
11         V(k) ← V(k) for each k;
12         A_i(k) ← A_i(k) for each k;
13         N_i(k) ← N_i(k) for each k;
14         return True;
15     return False;
```

		Parameter	Resources		
			R_1	R_2	R_3
		$R(k)$	9	3	6
Processes	P_1	$C_1(k)$	3	2	2
	P_2	$C_2(k)$	6	1	3
	P_3	$C_3(k)$	3	1	4
	P_4	$C_4(k)$	4	2	2

Table 21.1: Static parameters for Banker Algorithm example.

		Parameter	Resources			Parameter	Resources		
			R_1	R_2	R_3		R_1	R_2	R_3
		$V(k)$	1	1	2				
Processes	P_1	$A_1(k)$	1	0	0	$N_1(k)$	2	2	2
	P_2	$A_2(k)$	5	1	1	$N_2(k)$	1	0	2
	P_3	$A_3(k)$	2	1	1	$N_3(k)$	1	0	3
	P_4	$A_4(k)$	0	0	2	$N_4(k)$	4	2	0

Table 21.2: System state (dynamic parameters) for Banker Algorithm example.

		Parameter	Resources			Parameter	Resources		
			R_1	R_2	R_3		R_1	R_2	R_3
		$V(k)$	**0**	1	**1**				
Processes	P_1	$A_1(k)$	1	0	0	$N_1(k)$	2	2	2
	P_2	$A_2(k)$	**6**	1	**2**	$N_2(k)$	**0**	0	**1**
	P_3	$A_3(k)$	2	1	1	$N_3(k)$	1	0	3
	P_4	$A_4(k)$	0	0	2	$N_4(k)$	4	2	0

Table 21.3: Temporary system state to check request $Q_2 = (1,0,1)$ via invocation of `StateIsSafe(...)` for Banker Algorithm example. Updated fields are highlighted in red.

		Parameter	Resources			Parameter	Resources		
			R_1	R_2	R_3		R_1	R_2	R_3
		$V(k)$	**0**	1	**1**				
Processes	P_1	$A_1(k)$	**2**	0	**1**	$N_1(k)$	**1**	2	**1**
	P_2	$A_2(k)$	5	1	1	$N_2(k)$	1	0	2
	P_3	$A_3(k)$	2	1	1	$N_3(k)$	1	0	3
	P_4	$A_4(k)$	0	0	2	$N_4(k)$	4	2	0

Table 21.4: Temporary system state to check request $Q_2 = (1,0,1)$ via invocation of `StateIsSafe(...)` for Banker Algorithm example. Updated fields are highlighted in red.

1. The request passes checks at lines 2-5;
2. A new temporary state is created, with dynamic parameters expressed in Table 21.4;
3. Next, `StateIsSafe(...)` is invoked with the created temporary state;
4. In this case, note that there is no process i such that $V(k) \geq N_i(k)$ for each k. Algorithm 1 will not flag any process as `CanFinish` (Line 4).
5. It follows that `StateIsSafe(...)` returns False (Line 9-10).
6. Algorithm 2 will then deny Q_1 from P_1.

Considerations

It is worth making a few considerations about the Banker Algorithm. First off, note that the algorithm deems as safe a state that will not lead to a deadlock. However, there may exist a state that would not lead to a deadlock that is deemed as unsafe by the algorithm.

For this reason, the Banker Algorithm is still quite conservative, and could perform deadlock avoidance at a high performance (i.e. concurrency) loss. In fact, due to its overly pessimistic determination of what constitutes a safe state, the algorithm may deny requests that would not lead to deadlocks. In order to mitigate this pessimism, a new algorithm that considers the exact order at which resources are requested by all the processes would be necessary.

Finally, an obvious limitation of the Banker Algorithm is that deadlocks are still possible if processes declare inexact values of $C_i(k)$ for some resources, or if there are "hidden" shared resources that are not declared by any process.