For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices ***Allocation*** and ***Need*** as vectors and refer to them as ***Allocation****i* and ***Need****i* **.** The vector ***Allocation****i* specifies the resources currently allocated to process *Pi*; the vector ***Need****i* specifies the additional resources that process *Pi* may still request to complete its task.

6.5.3.1. Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize

Work = ***Available*** and ***Finish***[*i*] = ***false*** for $i = 0, 1, ..., n - 1$.

2. Find an index *i* such that both

a. ***Finish***[*i*] == ***false***

b. ***Need****i* ≤ ***Work***

If no such *i* exists, go to step 4.

3. ***Work*** = ***Work*** + ***Allocation****i*

Finish[*i*] = ***true***

Go to step 2.

4. If ***Finish***[*i*] == ***true*** for all *i*, then the system is in a safe state.

This algorithm may require an order of $m \times n2$ operations to determine whether a state is safe.

6.5.3.2. Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let ***Request****i* be the request vector for process *Pi* .

If ***Request****i* [*j*] == *k*, then

process *Pi* wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pi*, the following actions are taken:

1. If ***Request****i* ≤***Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

Available = ***Available***–***Request****i* ;

Allocation*i* = ***Allocation****i* + ***Request****i* ;

Need*i* = ***Need****i* –***Request****i* ;

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for ***Request****i* **,** and the old resource-allocation state is restored.

6.5.3.3. An Illustrative Example

To illustrate the use of the banker's algorithm, consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B*, and *C*. Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances. Suppose that, at time *T*0, the following snapshot of the system has been taken:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

The content of the matrix ***Need*** is defined to be ***Max*** − ***Allocation*** and is as follows:

	Need
	A B C
*P*0	7 4 3
*P*1	1 2 2
*P*2	6 0 0
*P*3	0 1 1
*P*4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence <*P*1,*P*3, *P*4, *P*2, *P*0> satisfies the safety criteria. Suppose now that process *P*1 requests one additional instance of resource type *A* and two instances of resource type *C*, so *Request*1 = (1,0,2). To decide whether this request can be immediately granted, we first check that *Request*1 ≤ ***Available***—that is, that (1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	***Need***	***Available***
	A B C	*A B C*	*A B C*
*P*0	0 1 0	7 4 3	2 3 0
*P*1	3 0 2	0 2 0	
*P*2	3 0 2	6 0 0	
*P*3	2 1 1	0 1 1	
*P*4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies the safety requirement. Hence, we can immediately grant the request of process *P*1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by *P*4 cannot be granted, since the resources are not available.

Furthermore, a request for (0,2,0) by $P0$ cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm.

6.6. Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

• An algorithm that examines the state of the system to determine whether

a deadlock has occurred

• An algorithm to recover from the deadlock

In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

6.6.1. Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge $Pi \rightarrow Pj$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $Pi \rightarrow Rq$ and $Rq \rightarrow Pj$ for some resource Rq . In Figure 6.7, we present a resource-allocation graph and the corresponding wait-for graph. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to ***maintain*** the wait-for graph and periodically ***invoke an algorithm*** that searches for a