16.3 Deadlocks

Deadlock is the situation in which multiple concurrent threads of execution in a system are blocked permanently because of resource requirements that can never be satisfied.

A typical real-time system has multiple types of resources and multiple concurrent threads of execution contending for these resources. Each thread of execution can acquire multiple resources of various types throughout its lifetime. Potential for deadlocks exist in a system in which the underlying RTOS permits resource sharing among multiple threads of execution. Deadlock occurs when the following four conditions are present:

Mutual exclusion-A resource can be accessed by only one task at a time, i.e., exclusive access mode.

No preemption-A non-preemptible resource cannot be forcibly removed from its holding task. A resource becomes available only when its holder voluntarily relinquishes claim to the resource.

Hold and wait-A task holds already-acquired resources, while waiting for additional resources to become available.

Circular wait-A circular chain of two or more tasks exists, in which each task holds one or more resources being requested by a task next in the chain.

Given that each resource is nonpreemptible and supports only exclusive access mode, Figure 16.1 depicts a deadlock situation between two tasks.

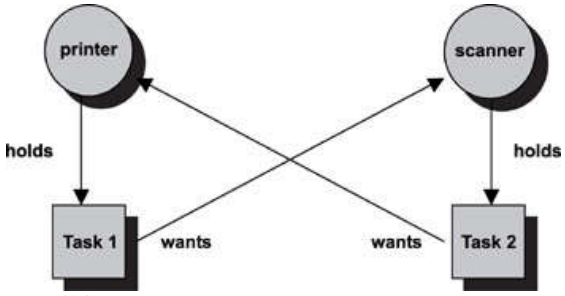

Figure 16.1: Deadlock situation between two tasks.

Figure 16.1 is a *resource graph* . An arrow labeled *holds* going from a resource to a task indicates that the task currently holds (or owns) the resource. An arrow labeled *wants* going from a task to a resource indicates that the task currently needs this resource to resume execution.

In this example, task #1 wants the scanner while holding the printer. Task #1 cannot proceed until both the printer and the scanner are in its possession. Task #2 wants the printer while holding the scanner. Task #2 cannot continue until it has the printer and the scanner. Because neither task #1 nor task #2 is willing to give up what it already has, the two tasks are now deadlocked because neither can continue execution.

Deadlocks can involve more than two tasks.

As shown in Figure 16.2, task T1 currently holds resource R1 (a printer), and T1 wants resource R2 (a scanner). Task T2 holds resource R2 and wants resource R3 (a memory buffer). Similarly, task T3 holds resource R3 and wants resource R1. It is easy to see the cycle, i.e., the circular-wait condition in this system. Tasks T1, T2, and T3, and resources R1, R2, and R3 comprise the *deadlocked set* . Note that in the system in Figure 16.2, one instance per resource type exists, i.e., there is one instance of R1, one instance of R2, and one instance of R3. A later section, 'Multi-Instance Resource Deadlock Detection' on page 266, discusses deadlock situations that involve multiple instances of a resource type.

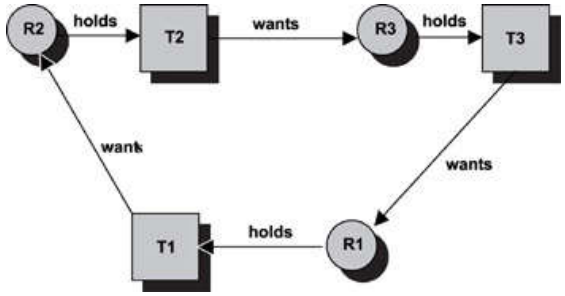

Figure 16.2: Deadlock situation among three tasks.

In this example, each task requires a single instance of a single resource type at any given time. Many situations exist in which a task might require multiple instances of multiple types of resources. The formation of deadlocks depends on how a task requests resources (formally known as a *resource request model*). The deadlock detection algorithms are constructed according to the resource request models.

16.3.1 Resource Request Models

When tasks ask for resources, the way the task makes the requests can be classified into these request models:

- the Single resource request model,
- the `AND` resource request model,
- the `OR` resource request model, and
- the `AND-OR` resource request model.

In the Single resource request model, exemplified in both Figure 16.1 and Figure 16.2, a task can have at most one outstanding resource request at any given time. In the request model, a task asks for resources as in 'wants a printer.'

In the `AND` resource request model, a task can have multiple simultaneous requests outstanding at any given time. For example, a task can request resources as (R1 *and* R2) or (R1 *and* R2 *and* R3). A task is blocked until all of the requested resources are granted. In this request model, a task asks for resources as in "wants both a printer and a scanner." The task resumes execution only when it successfully acquires both the printer and the scanner.

In the `OR` resource request model, a task can request a set of resources, but the task can resume execution as soon as any one of the resources from the request set becomes available. For example, a task can request resources as (R1 *or* R2) or (R1 *or* R2 *or* R3). In this request model, a task asks for resources as in "wants either a printer or a scanner." The task resumes execution when it acquires either the printer or the scanner.

In the `AND-OR` resource request model, a task can make resource requests in any combination of the `AND` and `OR` models. For example, a task can request a set of resources as (R1 *or* R2 *and* (R3 *or* R4)). In this request model, the task asks for resources as in "wants either a printer or a scanner, and wants either a memory buffer or a message queue." The task can resume execution when it acquires both the printer and the memory buffer, when it acquires both the printer and the message queue, when it acquires the scanner and the memory buffer, or when it acquires the scanner and the message queue. A generalization of the `AND-OR` model is the *C(n,k)* model. In this model, a task can make *n* resource requests and can resume execution as soon as *k* resources are granted, where $k \leq n$.

16.3.2 Deadlock Detection

A deadlock condition is called a *stable deadlock* when no task in the deadlocked set expects a timeout or an abort that can eliminate the deadlock. A stable deadlock is permanent and requires external influence to eliminate. The external influence is the deadlock detection and recovery by the underlying RTOS.

Deadlock detection is the periodic deployment of an algorithm by the RTOS. The algorithm examines the current resource allocation state and pending resource requests to determine whether deadlock exists in the system, and if so, which tasks and resources are involved.

The deadlock detection algorithm that the RTOS deploys is a global algorithm because it is used to detect deadlocks in the entire system. In general, each task of the deadlocked set is not aware of the deadlock condition. As a result, the recovery algorithm is more intrusive on the normal execution of the tasks belonging to the deadlocked set. The recovery algorithms and reasons why these algorithms are intrusive on the execution of the tasks involved in the deadlock are discussed shortly.

A *temporal deadlock* is a temporary deadlock situation in which one or more tasks of the deadlocked set either times out or aborts abnormally due to timing constraints. When the task times out or aborts, it frees the resources that might have caused the deadlock in the first place, thus eliminating the deadlock. This form of detection and recovery is localized to an individual task, and the task has deadlock awareness.

A system that is capable of deadlock detection is more efficient in terms of resource utilization when compared to a system without deadlock detection. A system capable of deadlock detection is not conservative when granting resource allocation requests if deadlock is allowed to occur. Therefore, resources are highly utilized. A system without deadlock detection is conservative when granting resource allocation requests. A resource request is denied if the system believes there is a potential for deadlock, which may never occur. The conservatism of the system results in idle resources even when these resources could be used.

Deadlock detection does not solve the problem; instead, the detection algorithm informs the recovery algorithm when the existence of deadlock is discovered.

For deadlock in the Single resource request model, a cycle in the resource graph is a necessary and sufficient condition.

For deadlock in the `AND` resource request model, a cycle in the resource graph is a necessary and sufficient condition. It is possible for a task to be involved in multiple deadlocked sets.

For deadlock in the `OR` request model, a knot is a necessary and sufficient condition.

Therefore, deadlock detection involves finding the presence of a cycle in the resource graph for both the Single and the `AND` resource request models. Deadlock detection involves finding the presence of a knot in the resource graph for the `OR` resource request model.

For deadlock in the `AND-OR` model, no simple way exists of describing it. Generally, the presence of a knot after applying the algorithm to the `OR` model first and then subsequently applying the algorithm to the `AND` model and finding a cycle is an indication that deadlock is present.

The following sections present two deadlock detection algorithms-one for the single resource request model and one for the `AND` resource request model-to illustrate deadlock detection in practice.

For node *A* in the resource graph, the reachable set of *A* is the set of all nodes *B*, such that a directed path exists from *A* to *B*. A *knot* is the request set *K*, such that the reachable set of each node of *K* is exactly *K*.

Single-Instance Resource Deadlock Detection

The deadlock detection algorithm for systems with a single instance of each resource type, and tasks making resource requests following the single resource request model, is based on the graph theory. The idea is to find cycles in the resource allocation graph, which represents the circular-wait condition, indicating the existence of deadlocks.

Figure 16.3 shows the resource allocation graph. The graph represents the following:

- a circle represents a resource,
- a square represents a task or thread of execution,
- an arrow going from a task to a resource indicates that the task wants the resource, and
- an arrow going from a resource to a task indicates that the task currently holds the resource.

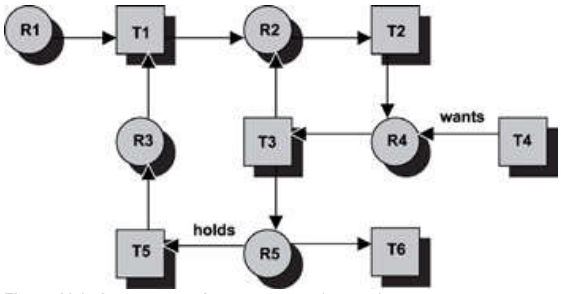

Figure 16.3: Current state of resource allocations and requests.

In the following discussions, *node* refers either to the circle (resource) or the square (task) in Figure 16.3. *Arc* refers to the arrow. The deadlock detection algorithm can be stated in these seven steps:

1. Make a list of all the nodes, *N*, from the graph.
2. Pick a node from *N*. Create another list, *L*, initially empty, which is used for the graph traversal.
3. Insert the node into *L* and check if this node already exists in *L*. If so, a cycle exists; therefore, a deadlock is detected, and the algorithm terminates. Otherwise, remove the node from *N*.
4. Check whether any un-traversed outgoing arcs from this node exist. If all of the arcs are traversed, go to step 6.
5. Choose an un-traversed outgoing arc originating from the node and mark the arc as traversed. Follow the chosen arc to the new node and return to step 3.
6. At this stage, a path in the graph terminates, and no deadlocks exist. If more than one entry is in *L*, remove the last entry from *L*. If more than one entry remains in *L*, make the last entry of *L* the current node and go to step 4.
7. If the list *N* is not empty, go to step 2. Otherwise, the algorithm terminates, and no deadlocks exist in the system.

The actual implementation from step 3 to step 6 translates into a depth first search of the directed graph.

Applying this algorithm to the system depicted in Figure 16.3 provides the following:

Step 1: *N* = { R1, T1, R2, T2, R3, T3, R4, T4, T5, R5, T6 }

Step 2: *L* = { <*empty*> }; pick node R1

Step 3: *L* = { R1 }; no cycles are in *L; N* = { T1, R2, T2, R3, T3, R4, T4, T5, R5, T6 }

Step 4: R1 has one outgoing arc

Step 5: Mark the arc; reaches node T1; go back to step 3

Step 3: *L* = { R1, T1 }; *N* = { R2, T2, R3, T3, R4, T4, T5, R5, T6 }; no cycles are in *L*

The algorithm continues from step 3 to step 5 and reiterates until it reaches node T3, in which the list *L* = { R1, T1, R2, T2, R4, T3 } and the list *N* = { R3, T4, T5, R5, T6 }. Two outgoing arcs are at node T3. When the downward arc is picked, *L* = { R1, T1, R2, T2, R4, T3, R5 }. Two outgoing arcs are at node R5. When the rightward arc is picked, *L* = { R1, T1, R2, T2, R4, T3, R5, T6 }.

Step 4: T6 does not have any outgoing arcs; continue to step 6

Step 6: Remove T6 from the list *L; L* = { R1, T1, R2, T2, R4, T3, R5 }; return to step 4

Step 4: Pick the unmarked leftward arc at R5

Step 5: Mark the arc; reaches node T5; return to step 3

Step 3: *L* = { R1, T1, R2, T2, R4, T3, R5, T5 }; *N* = { R3, T4 }; no cycles are in *L*

Step 4: Pick the only outgoing arc at T5

Step 5: Mark the arc; reaches node R3; go back to step 3

Step 3: L = { R1, T1, R2, T2, R4, T3, R5, T5, R3 }; N = { T4 }; still no cycles are in L

Step 4: Pick the only outgoing arc at R3

Step 5: Mark the arc; reaches node T1; go back to step 3

Step 3: L = { R1, T1, R2, T2, R4, T3, R5, T5, R3, T1 }; Node T1 already exists in L. A cycle is found in the graph, and a deadlock exists. The algorithm terminates.

The deadlock set is comprised of the entire nodes enclosed by the two occurrences of node T1 inclusively. Therefore, the discovered deadlock set is {T1, R2, T2, R4, T3, R5, T5, R3}. One thing worth noting is that the algorithm detects deadlocks if any exist. Which deadlock is detected first depends on the structure of the graph. Closer examination of the resource graph reveals that another deadlock exists. That deadlock set is {R2, T2, R4, T3}. At node T3 if the upward arc is chosen first instead of the downward arc, this later deadlock occurrence would be discovered, and the algorithm would terminate much sooner.

Multi-Instance Resource Deadlock Detection

The deadlock detection algorithm takes a different approach for systems with multiple instances of each resource type, and tasks make resource requests following the AND model. An underlying assumption is that a resource allocation system is present. The *resource allocation system* is comprised of a set of different types of resources, R1, R2, R3, …, R*n*. Each type of resource has a fixed number of units. The resource allocation system maintains a resource allocation table and a resource demand table.

Each row of tables C and D represents a task T. Each column of tables C and D is associated with a resource type. C is the resource allocation table representing resources already allocated. D is the resource demand table representing additional resources required by the tasks.

N = Total System Resources Table	N_1	N_2	N_3	…	N_k

where N_i is the number of units of resource type R_i for all i where { $1 \le i \le k$ }.

A = Available System Resources Table	A_1	A_2	A_3	…	A_k

where A_i the number of units remaining for resource type R_i available for allocation.

C = Tasks Resources Assigned Table	C_{11}	C_{12}	C_{13}	…	C_{1k}
	C_{21}	C_{22}		…	C_{2k}
	…			…	
	C_{m1}			…	C_{mk}
D = Tasks Resources Demand Table	D_{11}	D_{12}	D_{13}	…	D_{1k}
	D_{21}	D_{22}		…	D_{2k}
	…			…	
	D_{m1}			…	D_{mk}

For example in table C, there are C_{11} units of resource R1, C_{12} units of resource R2, and so on, which are allocated to task T1. Similarly, there are C_{21} units of resource R1, C_{22} units of resource R2, and so on, which are allocated to task T2. For example in table D, task T1 demands additional D_{11} units of resource R1, additional D_{12} units of resource R2, and so on, in order to complete execution.

The deadlock detection algorithm is as follows:

1. Find a row i in table D, where $D_{ij} < A_j$ for all $1 \le j \le k$. If no such row exists, the system is deadlocked, and the algorithm terminates.
2. Mark the row i as complete and assign $A_j = A_j + D_{ij}$ for all $1 \le j \le k$.
3. If an incomplete row is present, return to step 1. Otherwise, no deadlock is in the system, and the algorithm terminates.

Step 1 of the algorithm looks for a task whose resource requirements can be satisfied. If such a task exists, the task can run to completion. Resources from the completed task are freed back into the resource pool, which step 2 does. The newly available resources can be used to meet the requirements of other tasks, which allow them to resume execution and run to completion.

When the algorithm terminates, the system is deadlocked if table T has incomplete rows. The incomplete rows represent the tasks belonging to the deadlocked set. The algorithm is illustrated in the following example.

N =	4	6	2	
A =	1	2	0	
C =	0	2	0	Task 1
	1	1	0	Task 2
	1	1	1	Task 3
	1	0	1	Task 4
D =	2	2	2	Task 1
	1	1	0	Task 2
	0	1	0	Task 3
	1	1	1	Task 4

Step 1: Task 1 cannot continue because the available resources do not satisfy its requirements.

Task 2 can continue because what it needs can be met.

Step 2:	A = 2	3	0

Step 3: Task 1, task 3, and task 4 remain. Return to step 1.

Step 1: Task 1 still cannot continue. The requirement from task 3 can be met.

Step 2:	A = 3	4	1

Step 3: Task 1 and task 4 remain. Return to step 1.

Step 1: Task 1 still cannot continue, but task 4 can.

Step 2:	A = 4	4	2

Step 3: Task 1 remains. Return to step 1.

Step 1: Task 1 can continue.

Step 2:	A = 4	6	2

Step 3: No more tasks remain, and the algorithm terminates. No deadlock is in the system.

Now if the resource requirement for task 3 were `[ 0 1 1 ]` instead of `[ 0 1 0 ]`, task 1, task 3, and task 4 cannot resume execution due to lack of resources. In this case, these three tasks are deadlocked.

It is worth noting that executing a deadlock detection algorithm takes time and can be non-deterministic.

16.3.3 Deadlock Recovery

After deadlock is detected, the next step is to recover from it and find ways to break the deadlock. No one magic solution exists to recover from deadlocks. Sometimes it is necessary to execute multiple recovery methods before resolving a deadlock, as illustrated later.

For preemptible resources, resource preemption is one way to recover from a deadlock. The deadlocked set is transferred to the recovery algorithm after the detection algorithm has constructed the set. The recovery algorithm can then exercise preemption by taking resources away from a task and giving these resources to another task. This process temporarily breaks the deadlock. The latter task can complete execution and free its resources. These resources are used in turn to satisfy the first task for its completion. Resource preemption on preemptible resources does not directly affect the task's execution state or result, but resource preemption can affect a task's timing constraints. The duration of resource preemption can cause the preempted task to abort, which results in an incomplete execution and indirectly affects the result of a task.

For non-preemptible resources, resource preemption can be detrimental to the preempted task and can possibly affect the results of other tasks as well. For example, consider the situation in which one task is in the midst of writing data into a shared memory region, while at the same time a second task requests read access from the same memory region. The write operation is invalidated, when another task causes a deadlock, and the system recovers from the deadlock by preempting the resource from the writing task. When the second task gets the resource and begins accessing the shared memory, the data read is incoherent and inconsistent. For this reason, a shared memory region is classified as a non-preemptible resource. The preempted task writes the remaining data when the access to the shared memory is returned. The data is no longer useful, and the write operation is wasted effort. Sometimes this type of resource preemption is as good as eliminating the preempted task from the system altogether.

On the other hand, the effects of non-preemptible resource preemption can be minimized if a task has a built-in, self-recovery mechanism. A task can achieve self-recovery by defining checkpoints along its execution path. As soon as the task reaches a checkpoint, the task changes a global state to reflect this transition. In addition, the task must define a specific entry point to be invoked by the deadlock recovery algorithm after the task is allowed to resume execution. The entry point is nothing more than the beginning of the task's built-in, self-recovery routine. In general, the recovery involves rolling back and restarting execution from the beginning of the previous checkpoint. The concept is illustrated in Listing 16.1.

Listing 16.1: Checkpoints and recovery routine.

```
<code>                                  recovery_entry()
...                                     {
<code>                                       switch (state)
...                                          {
/* reached checkpoint #1 */                    case CHECKPOINT_1:
state = CHECKPOINT_1;                               recovery_method_1();
...                                                 break;
<code>                                       case CHECKPOINT_2:
...                                                 recovery_method_2();
/* reached checkpoint #2 */                         break;
state = CHECKPOINT_2;                        ...
                                             }
...                                     }
```

In Listing 16.1, a resource preemption is performed on a writer task and the preempted resource is given to the reader task. The writer task's self-recovery involves returning to the previous checkpoint and perhaps repeating the write operation, followed by a broadcast notification to all other tasks that the shared memory region has just been updated. This process can reduce the impact on other tasks.

The reassignment target of the preempted resource plays an important role in breaking the deadlock. For example, assume the deadlocked set {T1, R2, T2, R4, T3, R5, T5, R3} has been discovered, as shown in Figure 16.3. In addition, suppose resource R2 is preempted from T2 as the first recovery step. Figure 16.4 shows the resource graph if R2 were reassigned to T3.

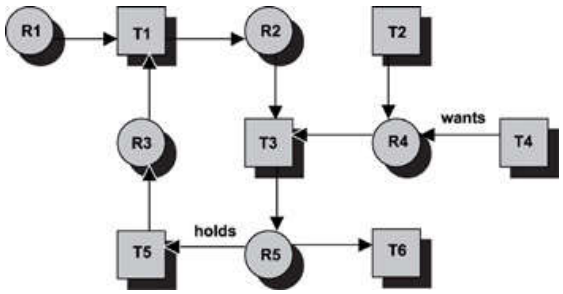

Figure 16.4: Resource preemption with a new deadlock.

The problem is not solved because a new deadlock is formed by this resource assignment. Instead, if R2 were given to T1 first, the deadlock is broken as shown in Figure 16.5.

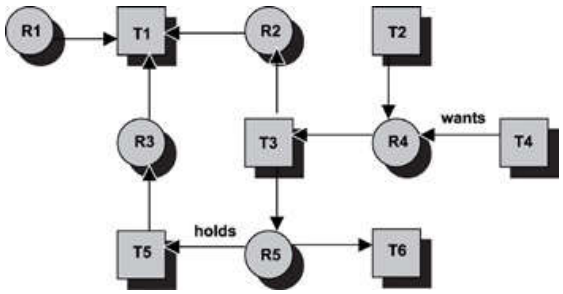

Figure 16.5: Deadlock eliminated by proper resource reassignment.

Consequently, T1 can complete and then frees R1, R2, and R3. This process in term enables T5 to complete and releases R5. Now, both R2 and R5 are available to T2, which allows it to run to completion. Finally, T2 is given a second chance to execute, and the deadlock is eliminated by proper resource reassignment.

16.3.4 Deadlock Avoidance

> *Deadlock avoidance* is an algorithm that the resource allocation system deploys. The algorithm predicts whether the current allocation request, if granted, can eventually lead to deadlock in the future.

Deadlock avoidance is similar to the deadlock detection algorithm outlined in the 'Multi-Instance Resource Deadlock Detection' on page 266. Each time a resource request is made, the system tests whether granting such a request might allow the remaining resources to be given to different tasks in subsequent allocations so that all tasks can run to completion. Revisiting the example given in 'Multi-Instance Resource Deadlock Detection' provides the following:

N =	4	6	2	

A =	1	2	0	
C =	0	2	0	Task 1
	1	1	0	Task 2
	1	1	1	Task 3
	1	0	1	Task 4
D =	2	2	2	Task 1
	1	1	0	Task 2
	0	1	0	Task 3
	1	1	1	Task 4

If task 2 requests one unit of resource R1, granting such a request does not lead to deadlock because a sequence of resource allocations exists, i.e., giving the remaining resources to task 2, to task 3, followed by allocation to task 4, and finally to task 1, which allows all tasks to complete. This request from task 2 is safe and is allowed. If task 4 were to make the same request for R1 and if such a request were granted, this process would prevent task 2 from completing, which would result in a deadlock such that no task could resume execution. The request from task 4 is an unsafe request, and the deadlock avoidance algorithm would reject the request and put task 4 on hold while allowing other tasks to continue.

In order for deadlock avoidance to work, each task must estimate in advance its maximum resource requirement per resource type. This estimation is often difficult to predict in a dynamic system. For more static embedded systems or for systems with predictable operating environments, however, deadlock avoidance can be achieved. The estimations from all tasks are used to construct the demand table, D. This resource estimation only identifies the potential maximum resource requirement through certain execution paths. In the majority of cases, there would be overestimation. Overestimation by each task can lead to inefficient resource utilization in a heavily loaded system. This problem is caused because the system might be running with most of the resources in use, and the algorithm might predict more requests as being unsafe. This issue could result in many tasks being blocked, while holding resources that were already allocated to them.

16.3.5 Deadlock Prevention

Deadlock prevention is a set of constraints and requirements constructed into a system so that resource requests that might lead to deadlocks are not made. Deadlock prevention differs from deadlock avoidance in that no run-time validation of resource allocation requests occurs. Deadlock prevention focuses on structuring a system to ensure that one or more of the four conditions for deadlock i.e., mutual exclusion, no preemption, hold-and-wait, and circular wait is not satisfied.

This set of constraints and requirements placed on resource allocation requests is as follows:

- **Eliminating the hold-and-wait deadlock condition.** A task requests at one time all resources that it will need. The task can begin execution only when every resource from the request set is granted.

  This requirement addresses the hold-and-wait condition for deadlock. A task that obtains all required resources before execution avoids the need to wait for anything during execution. This approach, however, has limited practicality and several drawbacks. In a dynamic system, tasks have difficulty predicting in advance what resources will be required. Even if all possible resource requirements could be accurately predicted, this prediction does not guarantee that every resource in this predicted set would be used. Execution paths, which external factors affect, determine which resources are used.

  One major drawbacks to this approach is the implicit requirement that all resources must be freed at the same time. This requirement is important because a resource can be needed in multiple code paths; it can be used and later be reused. So, the resource must be kept until the end of task execution. Some of the resources, however, might be used once or used only briefly. It is inefficient for these resources to be kept for a long time because they cannot be reassigned to other tasks.

- **Eliminating the no-preemption deadlock condition**. A task must release already acquired resources if a new request is denied. The task must then initiate a new request including both the new resource and all previously held resources.

  This requirement addresses the no-preemption condition for deadlock. This approach is slightly more dynamic than the previous method in that resources are acquired on an as-needed basis and only those resources needed for a particular execution path, instead of all possible resources, are acquired.

  This approach, however, is not much better than the previous one. For tasks holding non-preemptible resources, this requirement means that each task must restart execution either from the beginning or from well-defined checkpoints. This process nullifies partially complete work. Potentially, a task might never complete, depending on the average number of tasks existing in the system at a given time and depending on the overall system scheduling behavior.

- **Eliminating the circular-wait deadlock condition**. An ordering on the resources must be imposed so that if a task currently holds resource R_i, a subsequent request must be for resource R_j where $j > i$. The next request must be for resource R_k where $k > j$, and so on.

  This imposition addresses the circular-wait condition for deadlock. Resources are organized into a hierarchical structure. A task is allowed to acquire additional resources while holding other resources, but these new resources must be higher in the hierarchy than any currently held resources.