Chapter 7. Deadlocks

A process requests resources, when they are not available the process enters a waiting state. Sometimes it is unable to change state, because the resources it has requested are held by another waiting processes. This situation is a **deadlock**.

7.1. System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types, each consisting of some number of instances. Mutex locks and semaphores can be a source of deadlocks.

A process may utilize a resource in only the following sequence:

1. **Request:** The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

The request and release of resources may be system calls. For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. They could be either physical or logical resources.

7.2. Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Features that characterize deadlocks:

7.2.1. Necessary Conditions

All four conditions must hold for a deadlock to occur:

1. **Mutual Exclusion:** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another

process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No Preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed the task.
4. **Circular Wait:** A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

7.2.2. Resource-Allocation Graph

Deadlocks can be described as directed a **system resource-allocation** graph. This graph consists of a set of vertices (two types of nodes: P [set of active processes] and R [set of all resource types]) and a set of edges:

Request Edge: $P_i \rightarrow R_j$. Process P_i has requested an instance of resource type R_j and is currently waiting for that resource.

Assignment Edge: $R_j \rightarrow P_i$. An instance of resource type R_j has been allocated to process P_i.

Each resource instance is a dot in the graph. Request edge points to resource and assignment edge from an instance to a process. When process P_i requests an instance of resource R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is **instantaneously** transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource and the assignment edge is deleted.

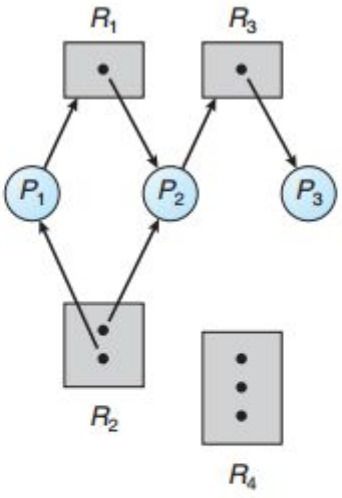

Figure 7.1 Resource-allocation graph.

$$P = \{P_1, P_2, P_3\}, R = \{R_1, R_2, R_3, R_4\}, E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$$

If the graph contains no cycles, then no process is deadlocked. If the graph contains a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. (In this case, a cycle is a necessary and sufficient condition for the existence of deadlock).

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. (In this case, a cycle is a necessary but not a sufficient condition for the existence of deadlock).

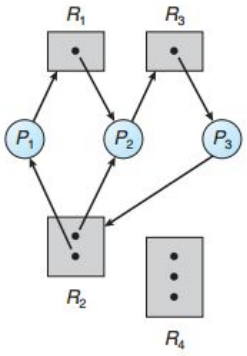

Figure 7.2 Resource-allocation graph with a deadlock.

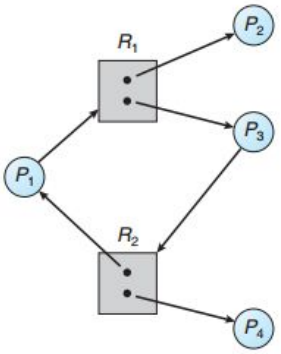

Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

7.3. Methods for Handling Deadlocks

- We can use a protocol to prevent or avoid deadlocks. The system will **never** enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem and pretend that deadlocks never occur. (Used by Linux and Windows. Avoiding deadlocks is responsibility of developers.)

To ensure that deadlocks never occur, the system can use: Deadlock Prevention and Deadlock Avoidance.

If none of the above is implemented, then the OS should use an algorithm to detect deadlocks and to recover from them. In the absence of algorithms, we may arrive at a situation in which the system is in a deadlocked state. Deadlocks occur infrequently (once per year).

7.4. Deadlock Prevention

By ensuring that at least one of the conditions cannot hold, we can **prevent** the occurrence of a deadlock. This method constrains how requests for resources can be made.

7.4.1. Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable.

7.4.2. Hold and Wait

To ensure that never occurs, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request

any additional resources, it must release all the resources that it is currently allocated.

Both have two main disadvantages: First, Resource utilization may be low, since resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

7.4.3. No Preemption

To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (must wait), then all resources the process is currently holding are preempted (implicitly released). The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it requests.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.This protocol is often applied to resources whose state can be easily saved and restored later (CPU, Memory).

7.4.4. Circular Wait

One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration. We assign to each resource a unique integer number. Suppose that function F returns the ordering number of a resource.

Protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$. Note also that if several instances of the same resource type are needed, a **single** request for all of them must be issued. If these two protocols are used, then the circular-wait condition cannot hold.

It is important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically.

7.5. Deadlock Avoidance

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation **state** is defined by the number of available and allocated resources and the maximum demands of the processes.

7.5.1. Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of processes $<P_1, P_2,$

…, P_n> is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with j < i. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. A deadlocked state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

A system can go from a safe state to an unsafe state. The idea is to ensure that the system will always remain in a safe state. Initially, the system is in a safe state.

7.5.2. Resource-Allocation-Graph Algorithm

This can be used only when each resource has only one instance. It is a variant of the previous Resource-Allocation Graph. A **claim edge** is added. It indicates ($P_i \rightarrow R_j$) that process P_i may request resource R_j at some time in the future. Same direction as a request edge, but it is a dashed line. When the process request the resource, the claim edge is converted to a request edge and when the resource is released by the process it is reconverted to a claim edge.

Before the process starts, all the claim edges must already appear in the graph. When the process request the resource it can only be granted if converting the request edge to an assignment edge does not result in the formation of a cycle. For this, a cycle-detection algorithm can be used. If no cycle, then safe state. If cycle, then unsafe state and the process have to wait for its requests to be satisfied.

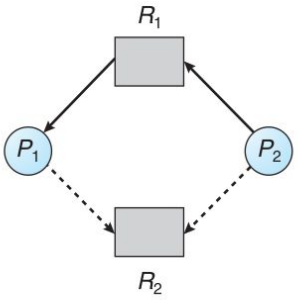

Figure 7.7 Resource-allocation graph for deadlock avoidance.

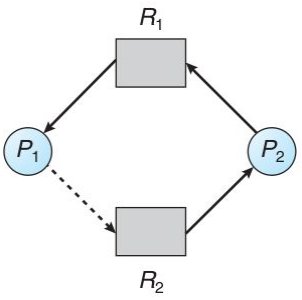

Figure 7.8 An unsafe state in a resource-allocation graph.

7.5.3. Banker's Algorithm

Used when resources can have multiple instances. Less efficient than the graph algorithm. Could be used in a banking system to ensure that the bank never allocate its available cash in a way that it could not satisfy the needs of its customers.

When a process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resource in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the request is granted, if not, the process must wait.

This algorithm requires the following data structures (n is the number of processes and m is the number of types of resources):

- **Available:** Vector of length m. If Available[j] == k, then k instances of resource type R_j are available.
- **Max:** n x m Matrix. If Max[i][j] == k, then process P_i may request at most k instances of R_j.
- **Allocation:** n x m Matrix. If Allocation[i][j] == k, then P_i is currently allocated k instances of R_j.
- **Need:** n x m Matrix. If Need[i][j] == k then P_i may need k more instances of resource type R_j to complete its task. Need[i][j] = Max[i][j] - Allocation[i][j]

Note: Let X and Y be vectors of length n. X <= Y if and only if X[i] <= Y[i] for all i = 1, 2, …, n. Y < X if Y <= X and Y != X.

$Available_i$ and $Need_i$ are rows and specifies the resources currently allocated for process P_i and the additional resources P_i may still need, respectively.

7.5.3.1. Safety Algorithm

It finds out if the system is in safe state or not. It may require m x n^2 operations.

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 Finish[i] == false && $Need_i$ <= Work. If no such i exists, go to 4.
3. Work = Work + $Allocation_i$, then Finish[i] = true, then Go to 2.
4. If Finish[i] == true for all i, then the system is in a safe state.

7.5.3.2. Resource-Request Algorithm

Determines whether requests can be safely granted.

Let **Request**$_i$ be the request vector for process P_i. If $Request_i$[j] == k, then P_i wants k instances of R_j. When a request is made the following actions are taken:

1. If $Request_i$ <= $Need_i$ go to 2. Otherwise, error, process has exceeded its maximum claim.
2. If Request <= Available, go to 3. Otherwise, wait, resources not available.
3. Have the system pretend to have allocated the requested resources by modifying the state as follows:

$$Available = Available - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need - Request_i$$

Apply the safety algorithm. If the result is a safe state, the request is granted. If it is unsafe, the request is denied, the process must wait, and the old state is restored (undo the previous operations).

7.5.3.3. An Illustrative Example

Resource A has 10 instances, B has 5 and C has 7. 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*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

1. Calculate the Need matrix (Max - Allocation):

	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

2. Verify the resource-allocation state and find a safe sequence:

(This is done by me to make it more clear. Need refers to $Need_{index}$)

Work = 3 3 2 Finish = 0 0 0 0 0 Index = 0 Safe Sequence = <>

Finish[0] is false but $Need_0$ is not less than Work. Index++.

Finish[1] is false and $Need_1$ is less than Work. So we perform Work + $Allocation_1$ and Finish [1] = true.

Work = 5 3 2 Finish = 0 1 0 0 0 Index++ SafeSequence = <P_1>

Finish[2] is false but $Need_2$ is not less than Work. Index++.

Finish[3] is false and $Need_3$ is less than Work. So we perform Work + $Allocation_3$ and Finish[3] = true.

Work = 7 4 3 Finish = 0 1 0 1 0 Index++ SafeSequence = <P_1, P_3>

Finish[4] == false and Need is less than Work. So we perform Work + $Allocation_4$ and Finish[4] = true.

Work = 7 4 5 Finish = 0 1 0 1 1 Index = 0 SafeSequence = <P_1, P_3, P_4>

Finish[0] == false but Need is not less than Work. Index++.

Finish[1] == true. Index++.

Finish[2] == false and Need is less than Work. So we perform Work + $Allocation_2$ and Finish[2] = true.

Work = 10 4 7 Finish = 0 1 1 1 1 Index++ SafeSequence = <P_1, P_3, P_4, P_2>

Finish[3] == true. Index++.

Finish[4] == true. Index = 0.

Finish[0] == false and Need is less than Work. So we perform Work + $Allocation_0$ and Finish[0] = true.

Work = 10 5 7 Finish = 1 1 1 1 1 SafeSequence = <P_1, P_3, P_4, P_2, P_0>

Finish[i] == true for all i. System is in safe state. (Note that Work equals the instances of each resource.)

3. Now P_1 requests (1, 0, 2). You have to create a new state assuming that the request is granted (Step 3 from the Resource-Request Algorithm):

	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	

4. After this you execute the safety algorithm. It should give you the safe sequence <P_1, P_3, P_4, P_0, P_2>

5. Note that if P_4 requests (3, 3, 0) it cannot be granted because the resources are not available (Step 3 from the Resource-Requ

est Algorithm.)

6. Now, P_0 requests (0, 2, 0). Then we get the following state:

	Allocation	Need	Available
P_0	0 3 0	7 2 3	
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	2 1 0
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must verify the safe state:

Work = 2 1 0 Finish = 0 0 0 0 0 Index = 0 SS = <>

Finish[0] == false but Need is not less than Work. Index++.

Finish[1] == false but Need is not less than Work. Index++.

Finish[2] == false but Need is not less than Work. Index++.

Finish[3] == false but Need is not less than Work. Index++.

Finish[4] == false but Need is not less than Work. Index = 0.

We notice that there is no i that satisfies step 2 of the Safety Algorithm, hence the system is not in safe state and the request of P_0 must be denied. http://www.geeksforgeeks.org/program-bankers-algorithm-set-1-safety-algorithm/

7.6. Deadlock Detection

If there is no prevention or avoidance algorithm the system may provide:

- An algorithm to detect a deadlock.
- An algorithm to recover from a deadlock.

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.

7.6.1. Single Instance of Each Resource Type

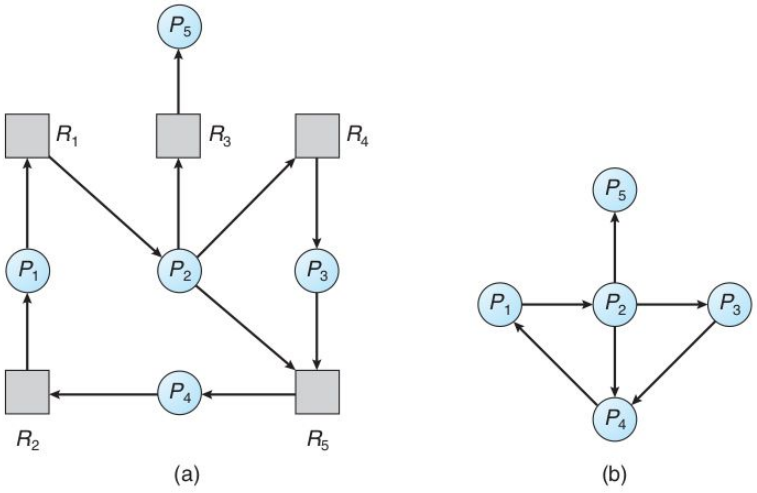

Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Deadlock-detection algorithm that uses a **wait-for** graph. It is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

An edge from P_i to P_j means that P_i is waiting for P_j to release a resource that it needs. This edge exists if and only if the corresponding resource-allocation graph contains $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$.

A deadlock is detected if a cycle is detected in the wait-for graph. The system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.

7.6.2. Several Instances of a Resource Type

Employs several time-varying data structures similar to those used in the banker's algorithm.

- Uses Available and Allocation as in Banker's.
- **Request:** n x m matrix. If Request[i][j] == k, then P_i is requesting k more instances of R_j.

The detection algorithm investigates every possible allocation sequence for the processes that remain to be completed. It requires m x n^2 operations to detect the deadlocked state.

1. Let **Work** and **Finish** be vectors of m and n, respectively. Initialize Work Available. For i = 0, 1, …, n, if $Allocation_i$!= 0, then Finish[i] = false. Otherwise Finish[i] = true.
2. Find an index i such Finish[i] == false && $Request_i$ <= Work.
   If no such i exists, go to 4.
3. Work = Work + $Allocation_i$
   Finish[i] = true
   Go to 2
4. If Finish[i] == false for some i, 0 <= i <= n, then the system is in a deadlocked state. Moreover, if Finish[i] == false, then process P_i is deadlocked.

Consider three resource types, A with 7 instances, B with 2 and C with 6.

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

The system is not in a deadlocked state. The sequence <P_0, P_2, P_3, P_1, P_4> results in Finish[i] == true for all i.

Suppose that P_2 requests (0, 0, 1). The request matrix is modified as follows:

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

The system is now deadlocked. The number of available resources is not sufficient. The deadlock consists of P_1, P_2, P_3 and P_4.

7.6.3. Detection-Algorithm Usage

When should we invoke the detection algorithm? It depends on two factors:

1. How **often** is a deadlock likely to occur?
2. How **many** processes will be affected by deadlock when it happens?

If deadlocks occur frequently, the algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of

waiting processes. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately.

Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked processes “caused” the deadlock.

7.7. Recovery from Deadlock

When a deadlock is detected the system can **recover** from it automatically. There are two options: Abort one or more to break the circular wait, or preempt some resources from one or more of the deadlocked processes.

7.7.1. Process Termination

Two methods to abort deadlocked processes. The system reclaims all resources allocated to the terminated processes:

- **Abort all deadlocked processes.** Expensive. The processes may have computed for a long time. These partial computations are discarded and probably will be recomputed.
- **Abort one process at a time until the deadlock cycle is eliminated.** Considerable overhead. After each process abortion, the deadlock-detection algorithm must be executed.

The abortion of processes could require the reset of some resources. If the partial termination method is used we must determine which process to abort first. Many factors may affect which process is chosen:

1. Priority of the process.
2. How long the process has computed and how much longer to completion.
3. Resources the process has used.
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

7.7.2. Resource Preemption

We successively preempt some resource from processes and give these resource to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? We must determine the order of preemption to minimize cost. Cost factors may include the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? It is missing some needed resource. We must roll back the process to some safe state and restart it from that state. It is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. It is more effective to roll back the process as far as necessary to break the deadlock, but requires to keep more information.
3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? Where victim selection is based primarily on cost factors, it may happen that the same process is always picked. This process never completes its designated task and results in starvation. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.