

attempt to keep at least `desfree` pages free. If it is unable to do this for a 30-second average, then Solaris begins swapping processes, starting preferably with processes that have been idle for a long time.

- If free memory falls below `minfree`, then page out runs with every page fault.
- Recent releases of Solaris have enhanced the virtual memory management system, including recognizing pages from shared libraries, and protecting them from being paged out.

3.3 DEAD LOCKS

3.3.1. System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:

1. **Request** - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. **Example:** system calls `open()`, `malloc()`, `new()`, and `request()`.
2. **Use** - The process uses the resource.

Example: prints to the printer or reads from the file.

3. **Release** - The process relinquishes the resource. so that it becomes available for other processes.

Example: `close()`, `free()`, `delete()`, and `release()`.

- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or `wait()` and `signal()` calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

3.3.2. Deadlock Characterization

Necessary Conditions:

There are four conditions that are necessary to achieve deadlock:

Mutual Exclusion - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.

Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

No preemption - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

Circular Wait - A set of processes $\{ P_0, P_1, P_2, \dots, P_N \}$ must exist such that every $P[i]$ is waiting for $P[(i + 1) \% (N + 1)]$. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

Resource-Allocation Graph

In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:

*A set of resource categories, $\{ R_1, R_2, R_3, \dots, R_N \}$, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)

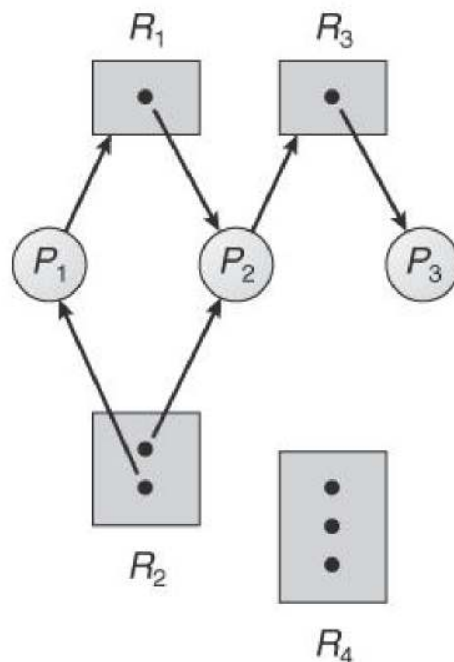
*A set of processes, $\{ P_1, P_2, P_3, \dots, P_N \}$

***Request Edges** - A set of directed arcs from P_i to R_j , indicating that process P_i has requested R_j , and is currently waiting for that resource to become available.

***Assignment Edges** - A set of directed arcs from R_j to P_i indicating that resource R_j has been allocated to process P_i , and that P_i is currently holding resource R_j .

Note that a request edge can be converted into an assignment edge by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)

For example:



*If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are directed graphs.) See the example in Figure 7.2 above.

*If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.

*If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:

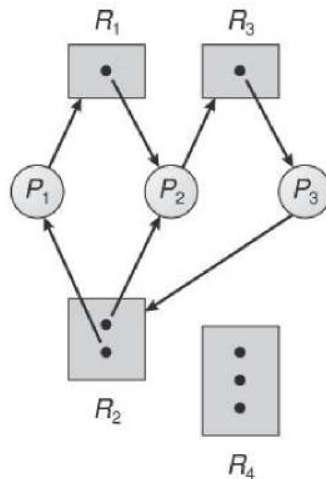


Fig: Resource Allocation Graph with cycle

3.3.3. Methods for Handling Deadlocks

Generally speaking there are three ways of handling deadlocks:

Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.

Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.

Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm.)

Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.

If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by

the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

3.3.4. Deadlock Prevention

Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

Shared resources such as read-only files do not lead to deadlocks.

Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:

- i. Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
- ii. Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
- iii. Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.

- i. One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
- ii. Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
- iii. Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- i. One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- ii. In other words, in order to request resource R_j , a process must first release all R_i such that $i \geq j$.
- iii. One big challenge in this scheme is determining the relative ordering of the different resources

Deadlock Avoidance

The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.

This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is

a conservative approach.)

In some algorithms the scheduler only needs to know the maximum number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the schedule of exactly what resources may be needed in what order.

When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.

A resource allocation state is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

- i. A state is safe if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- ii. More formally, a state is safe if there exists a safe sequence of processes $\{ P_0, P_1, P_2, \dots, P_N \}$ such that all of the resource requests for P_i can be granted using the resources currently allocated to P_i and all processes P_j where $j < i$. (I.e. if all the processes prior to P_i finish and free up their resources, then P_i will be able to finish also, using the resources that they have freed up.)
- iii. If a safe sequence does not exist, then the system is in an unsafe state, which MAY lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

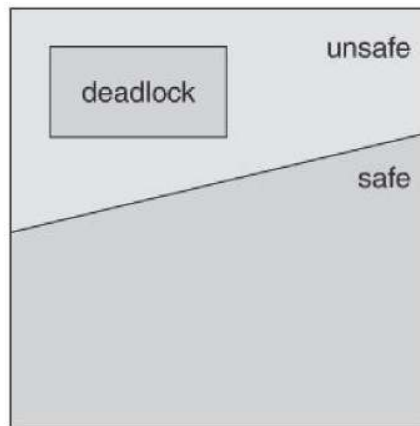


Fig: **Safe, unsafe, and deadlocked state spaces.**

For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

- i. What happens to the above table if process P2 requests and is granted one more tape drive?
- ii. Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

- i. If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- ii. In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges, noted by dashed lines, which point from a process to a resource that it may request in the future.
- iii. In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- iv. When a process makes a request, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- v. This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.

Consider for example what happens when process P2 requests resource R2:

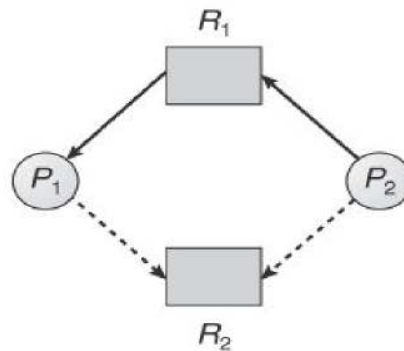


Fig: Resource allocation graph for dead lock avoidance

The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

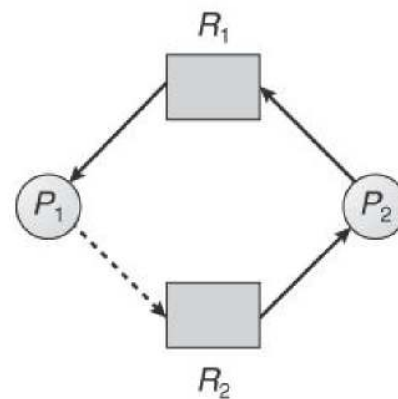


Fig: An Unsafe State in a Resource Allocation Graph

Banker's Algorithm

- i. For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.
- ii. The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- iii. When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- iv. When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types. N

Available: Vector of length m . If $available[j] = k$, there are k instances of resource type R_j
available n

Max: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j

Allocation: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j

Need: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$Need[i,j] = Max[i,j] - Allocation[i,j]$

Safety Algorithm

1. Let $Work$ and $Finish$ be vectors of length m and n , respectively.
Initialize: $Work = Available$ $Finish[i] = false$ for $i = 0, 1, \dots, n-1$
2. Find and i such that both:
 - (a) $Finish[i] = false$
 - (b) $Need_i \leq Work$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

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i . If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$

go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request;$

$Allocation_i = Allocation_i + Request;$

$Need_i = Need_i - Request;$

If safe \triangleright the resources are allocated to P_i

If unsafe $\triangleright P_i$ must wait, and the old resource-allocation state is restored

An Illustrative Example

Consider the following situation:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Example: P_1 Request (1,0,2) Check that $Request \leq Available$ (that is, $(1,0,2) \leq (3,3,2) \triangleright true$

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	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			

Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.

3.3.6. Deadlock Detection

- i. If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- ii. In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

6.1 Single Instance of Each Resource Type

- i. If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a wait-for graph.
- ii. A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- iii. An arc from P_i to P_j in a wait-for graph indicates that process P_i is waiting for a resource that process P_j is currently holding.

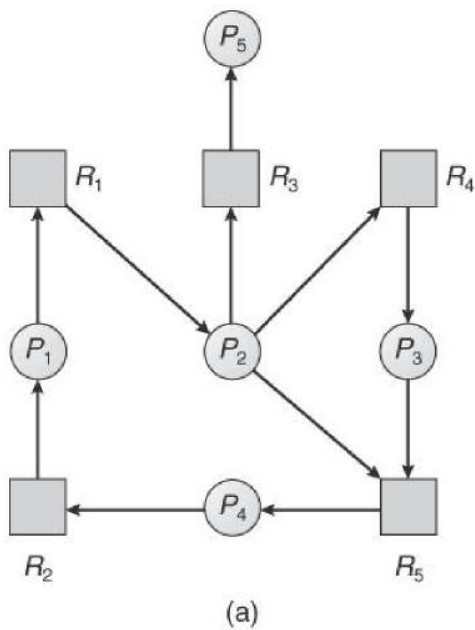


Fig: Resource Allocation Graph

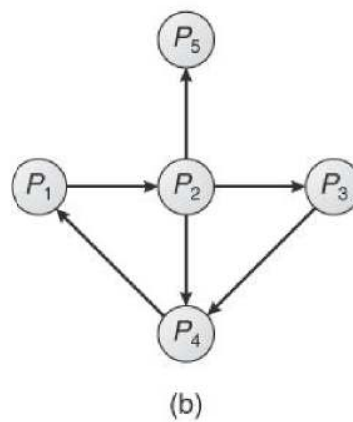


Fig: Corresponding Wait for Graph

As before, cycles in the wait-for graph indicate deadlocks.

This algorithm must maintain the wait-for graph, and periodically search it for cycles.

Several Instances of a Resource Type Available:

A vector of length m indicates the number of available resources of each type.

Allocation: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

Request: An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[ij] = k$, then process P_i is requesting k more instances of resource type R_j .

Detection Algorithm

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

(a) $Work = Available$ (b) For $i = 1, 2, \dots, n$,

if $Allocation_i \neq 0$, then

$Finish[i] = false$;

otherwise,

$Finish[i] = true$

2. Find an index i such that both:

(a) $Finish[i] == false$

(b) $Request_i \leq Work$

If no such i exists, go to step 4

3. $Work = Work + Allocation_i$

$Finish[i] = true$ go to step 2

4. If $Finish[i] == false$, for some i , $1 \leq i \leq n$, then the system is in deadlock state.

Moreover, if $Finish[i] == false$, then P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T0:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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	

Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Detection-Algorithm Usage

i. When should the deadlock detection be done? Frequently, or infrequently?

The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)

ii. There are two obvious approaches, each with trade-offs:

1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.

2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes

involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.

3.(As I write this, a third alternative comes to mind: Keep a historical log of resource allocations, since that last known time of no deadlocks. Do deadlock checks periodically (once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock. Unfortunately I'm not certain that breaking the original deadlock would then free up the resulting log jam.)

7. Recovery From Deadlock

There are three basic approaches to recovery from deadlock:

- i. Inform the system operator, and allow him/her to take manual intervention.
- ii. Terminate one or more processes involved in the deadlock
- iii. Preempt resources.

Process Termination

1. Two basic approaches, both of which recover resources allocated to terminated processes:

- i. Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
 - ii. Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

2. In the latter case there are many factors that can go into deciding which processes to terminate next:

- i. Process priorities.
- ii. How long the process has been running, and how close it is to finishing.
- iii. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
- iv. How many more resources does the process need to complete.
- v. How many processes will need to be terminated
- vi. Whether the process is interactive or batch.

Resource Preemption

When preempting resources to relieve deadlock, there are three important issues to be addressed:

Selecting a victim - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.

Rollback - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)

Starvation - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.