

Deadlocks and their avoidance

Serially reusable resources:

Resources that can be safely used by *only one process at a time*. After one process finishes using the resource, it may be safely assigned to another process.

Examples include:

- tape drive
- printer
- files
- file directories
- regions of a database
- internal O/S data structures (e.g. a semaphore's wait queue)

Some operating systems export an alloc / free mechanism analogous to the enqueue/dequeue mechanism of MVS / Z-OS to the application layer. Others employ *ad hoc* enqueue / dequeue mechanisms internally.

Deadlock:

A condition in which two or more processes become permanently blocked due to resource contention.

Example of a deadlock:

Assume that *alloc* requests block the requestor if the requested resource is not available.

P1	P2
alloc(R1)	alloc(R2)
alloc(R2) /* blocked */	alloc(R1) /* blocked */

Processes P1 and P2 are now permanently deadlocked.

Four conditions necessary for deadlock to occur.

1. Serially-reusable resources (only 1 process at a time can use them)
2. No preemption of resources is allowed.
3. Process(es) allowed to hold onto allocated resources while waiting on an allocate request for another resource.
4. A circular chain of blocked processes and resources exists.

Deadlock Prevention

Deadlock cannot occur if we can guarantee that *at least one* of these conditions can't hold.

Condition 1: Can be negated by declaring all resources sharable! While this would prevent deadlock, the results would be destructive: interleaved lines on printer, data blocks on tapes, corrupted file directories.

Condition 2: Can be negated by requiring that a process acquire all of its resources at one time. Either the process gets all of its resources and proceeds, OR it gets none of them and it waits. The MVS batch job facility works in this way. A batch job can be indefinitely delayed under this approach but no deadlock can occur.

Condition 3: Can be negated by allowing preemption in which the OS unilaterally revokes a resource that was previously allocated. The process from which the allocation is revoked *must be blocked* until the allocation can be restored. Preemption must be transparent. For example when a memory allocation is preempted the contents may be written out to disk and read back in when the memory allocation is restored.

Condition 4: Can be negated by hierarchical allocation. All resources are ordered into some arbitrary hierarchy. For example, files could be organized alphabetically. In hierarchical allocation a process can never request a resource lower in the hierarchy than any resource it currently holds.

Example:

$R1 < R2 < R3 < R4 < R5 \dots < Rn$

P1	P2
alloc(R1)	alloc(R2)
alloc(R2)	alloc(R1) not allowed

Disadvantages:

It may be difficult to come up with a reasonable hierarchy.

A very important use is serializing access to internal OS data structures.

The OS designer must develop a natural hierarchy of locks.

Example locks: (borrowed from MVS...apply to MPU environments)

1. Dispatcher "Ready List" (HIGHEST PRIORITY LOCK)
2. Memory management control blocks
3. I/O Device queues of waiting requests

For example, while holding an I/O device lock it is legal to try to obtain a memory management lock *but not vice versa*.

It is incumbent upon the kernel programmer *to be aware of and to honor* the lock hierarchy!!

Consumable and Serially Reusable Resources

Resources can be viewed as 1 of 2 classes.

Class 1: Serially reusable resources:. All instances of the resource are created when the system is started. SR resources are never created nor destroyed during normal system operation.

Class 2: Consumable resources: e.g. messages being passed in a pipe system. Messages may be created and destroyed during normal operation. A process that attempts to *read()* a pipe that is never written is blocked forever, but its not possible to know for sure that the pipe will never be written.

Deadlock management algorithms

Apply only to serially reusable resources.

Deadlock Detection

Graphical deadlock detection

A square represents a process

A circle a resource class

A small circle represents a resource instance

An arc from resource instance to process \rightarrow process *holds* resource

arc from process to resource class \rightarrow process is waiting on resource

The resulting graph is known as a *directed bi-partite* graph.

directed means its possible to traverse arcs only in the direction of the arrow.

bi-partite means that arcs connect processes to resources and vice versa.. never processes to processes or resources to resources.

For single unit resources deadlock exists \Leftrightarrow graph has a *cycle*.

For mult-unit resources deadlock exists \Leftrightarrow graph has a *knot*.

The *reachable set* of a node N , $R(N)$, is the set of all nodes that may be reached by starting at N and proceeding though the graph following the arcs in the proper direction.

A graph has a *cycle* if there exists a node N for which N is an element of $R(N)$.

A graph has a *knot* if there exists a subset of nodes S , such that for every node N in S , $R(N) = S$.

It is possible for a graph to have a *cycle* but not have a *knot*.

A graph that has a *knot* always has a *cycle*.

If *only* single unit resources are in use any *cycle* will always be a *knot*.

Table driven deadlock detection

For computer-based deadlock detection, a table-driven approach is far more reasonable.

Several tables are employed:

The AVAILABLE table

R0	R1	R2	R3	R4	
1	0	3	1	0	<--# of each resource type

At boot time the table is initialized to hold the number of each class that exists.

At allocation the number of units of units allocated is subtracted from the available table

At free the number of units freed is added back into the available table.

The REQUESTED table

	R0	R1	R2	R3	R4
P0					
P1					
P2					
P3					

A process will have a non-zero entry in this table only if it is blocked.

The entry will reflect how many additional units are are required.

If all rows in this table have a non-zero entry a deadlock exists.

If one or zero rows have a non-zero entry a deadlock does not exist.

Otherwise ???

ALLOCATED

	R0	R1	R2	R3	R4
P0					
P1					
P2					
P3					

Entries in the allocated table identify the number of instances of resources of each class that are currently held by a process.

Deadlock Detection Algorithm

1. Make a "working copy" of all three tables and copy current contents to working copy. All adjustments below are made to the working copies of the tables... Never the real thing.
2. Unmark all processes.
3. Clear the *found_flag*

For all *unmarked* processes j

```
{  
    if (requested[j][i] <= available[i] for all  $i$ )  
    {  
        mark process  $j$   
        set the found_flag  
        add Allocated[j][i] to Available[i] for all  $I$   
    }  
}
```

4. if (*found_flag* is set)
 Repeat step 3.
5. If (all processes are marked)
 system does not contain a deadlock
else
 unmarked processes are involved in a deadlock.

When a deadlock occurs, the deadlocked processes must be sequentially cancelled until it is resolved.

Deadlock Avoidance

Objective is to constrain allocation in such a way that deadlocks can't occur.

Deadlock Avoidance is somewhat less constraining than Deadlock Prevention.

The Banker's Algorithm

Processes must declare *in advance* the maximum number of resources in each class that it will hold at any time.

Resource manager retains enough resources sufficient to ensure that deadlock does not occur even if every unblocked process asserts its maximum claim.

Basic algorithm extends the detection algorithm.

A new data structure, called the Claims Matrix is maintained by the resource manager.

CLAIMS (each process has left)

	R0	R1	R2	R3	R4
P0					
P1					
P2					
P3					

The Available, Requested, and Allocated tables are maintained just as in the Deadlock Detection algorithm. Each time an allocation is made, the amount allocated is subtracted from both Claims and Available.

If allocation can't be granted, the number of units is entered in the Requested matrix and the process is blocked.

When a resource is freed, the number of units freed is subtracted from the Allocated matrix and is added to both the Claims and the Available matrices.

Details of the algorithm

1. Make a "working copy" of all three tables and copy current contents to working copy. All adjustments below are made to the working copies of the tables... Never the real thing.
2. "Pretend" to grant the request. That is, update the available, allocated and claims tables as if the request had been granted.
3. Unmark all processes.
4. Clear the *found_flag*

For all *unmarked* processes *j*

```
{  
    if (claims[j][i] <= available[i] for all i)  
    {  
        mark process j  
        set the found_flag  
        add Allocated[j][i] to Available[i] for all i  
    }  
}
```

5. if (*found_flag* is set)
 Repeat step 3.
6. if (all processes are marked)
 request is safe and may be granted.
else
 block requesting process

In summary

The Banker's Algorithm is:

Safe and effective but has disadvantages

Each process must know ahead of time the number of resources needed

Considerable overhead in running algorithm each time before resource is allocated.

Essential differences between detection and avoidance algorithms:

Run deadlock detection *when we a process is blocked*

Run deadlock avoidance *before granting a request for a resource.*

Deadlock management algorithms are not normally include in modern operating systems, but may be included in multithreaded applications and or Database management systems.