<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DEADLOCK CAUSES</td>
</tr>
<tr>
<td>3</td>
<td>DEADLOCK DETECTION, AVOIDANCE and PREVENTION</td>
</tr>
<tr>
<td>4</td>
<td>DETECTION-RECOVERY NO EQUIVALENT RESOURCES</td>
</tr>
<tr>
<td>5</td>
<td>DETECTION-RECOVERY EQUIVALENT RESOURCES</td>
</tr>
<tr>
<td>6</td>
<td>EXAMPLE OF CYCLE BUT NO DEADLOCK</td>
</tr>
<tr>
<td>7</td>
<td>DEADLOCK AVOIDANCE</td>
</tr>
<tr>
<td>8</td>
<td>DETECTION-AVOIDANCE EXAMPLE</td>
</tr>
<tr>
<td>9</td>
<td>DEADLOCK PREVENTION</td>
</tr>
<tr>
<td>10</td>
<td>CONSEQUENCES OF PREVENTION BY NUMBERING</td>
</tr>
<tr>
<td>11</td>
<td>MISCELLANEOUS EXAMPLES</td>
</tr>
<tr>
<td>12</td>
<td>PICTURING DEADLOCK-OCCURRENCE &amp; PREVENTION</td>
</tr>
</tbody>
</table>
Table Size determined with the expectation that every entry will leave eventually - assume also that once a Process holds a place in Table it will not give it up till it has opened all files it needs to continue.

MEMORY FULL DEADLOCK

down(x); down(y);<shared1> <shared1>
down(y);<shared2> up(x); up(y);
It can happen legitimately

Needs-Holds Graph Unique Resources

SEMAPHOR REQUEST ORDER DEADLOCK

get(tape_x); get(tape_y);
<PROG1>

semaphor x=1,y=1

P1 down(x); down(y);
down(y);<shared1> down(x);
<shared2> up(x); up(y);

<PROG1>

P2 down(x);
down(y);
<shared2>
up(x);
up(y);

<PROG2>

TAPE ORDER DEADLOCK (If only 1 Process can hold a Tape)

Deadlock can only happen if a Philosopher can hold 1 chopstick, needing another. It can’t happen if, as stated earlier a Philosopher holds both chopsticks or neither. This suggests a way to prevent deadlock?

MODIFIED DINING PHILosophers-- DEADLOCK

© M C Paull 2005
In a single Process there may be an endless loop. This could only be detected if one was aware that the Process never terminated. A range of other comparable difficulties arise when there are a number of Processes running in (pseudo) parallel and those Processes share Resources. This includes Deadlock. Processes in Deadlock cannot continue beyond the point at which they have entered that condition. They do not progress. As a practical matter such a situation is difficult to detect or avoid. Even though the theory of Deadlock is well understood, its insights, have not yet led to a widely accepted practical approach to its control. This fact is detectable, but there is not much done currently to go beyond this detection.

Here we review the theory of deadlock, its Detection, Recovery, Avoidance, and Prevention.

The basic approach to Detecting and Avoiding (Dodging) Deadlock in a principled manner consists of:

1. Constructing the Needs-Holds Graph - With One vertex for each Process and one for each Type of Resource - with a number assigned to each Resource Type giving the number of that type.
2. Looking for a Cycle in that Graph.

**Alg 1**
Simply Detects the existence of a Cycle:
- a. Start at any vertex find all its immediate neighbors.
- b. From each of these find all immediate neighbors, etc.
- c. Until a vertex repeats (there is a cycle) or one cannot continue (there is no cycle).

**Alg 1** is only conclusive if there is only one Resource of each type, [Unique Resources]. If there is more than one Resource of a given type [Equivalent Resources], all of which will satisfy a Need equally well, then finding a cycle is not sufficient to show Deadlock. (See example-page 4). In this case an approach in which Processes which are not involved in cycles and their Holds and Needs are systematically removed does work.

**Alg 2**
On a copy of the graph:
- a. See if any Processes NEEDs can all be satisfied.
- b. If so satisfy the needs with holds and remove that Process and all the Resources it holds from the graph.
- c. If any Process are left Repeat step a
  - If all Processes are finally removed by this procedure there is no Deadlock in the original graph, if not there is.

If one has identified the cause of the deadlock one can Recover by removing HOLDs from a Process which cannot, in any case, continue and give it to another Process [Preemption] which may thereby complete and release its held resources.

These algorithms can be extended from simply testing for deadlock to Avoidance (Dodging) of Deadlock. Before satisfying any NEED or HOLD one could use either algorithm above to determine whether that addition will cause Deadlock. If so the request is refused.

For Prevention there are a number of Approaches. One called Two-Phase Locking when applied in Data Bases is to request and acquire all Resources necessary to continue before continuing. This applies to the Hungry Philosophers problem-where Philosophers are able to Eat by requesting both chopsticks and either acquiring both or none (wait). It becomes less feasible if activity is required between acquisitions.

Another, more general, approach involves the sequential numbering of Resources with the proviso that Resources always be requested in ascending numerical order.

Deadlock, unlike an endless loop, is not necessarily the result of what would normally be called a coding error. For the most part a single Process or a number of Processes designed to run together can be carefully designed to avoid Deadlock. Deadlock most often results from a series of events which involve a number of Processes with minimal relation to one another. If the same set of Processes are run again that Deadlock may not recur.
DEADLOCK=1
Detection In Graph Representation
A Cycle In the Graph
Deadlock

Every Resource, even equivalent ones, must have unique identification. Cannot Need One of a Number of Equivalent Resources. In this case looking for a cycle is simple.
Assume each request is for a unique resource- If a Process is limited to Needing one outstanding (blocked) resource. From P0 there is a unique edge (representing a Need) that can leave P0 to R0. Now R0 can be Held by only 1 Process, P1, and again if there is deadlock. P1 must also have a unique Need-R2 etc.

Graph Basics
A vertex per Resource can be Held by at most one Process in this Model.
NEEDS: An edge from Process to Resource. (There can be many to the same Resource)
HOLDS: An edge from a Resource to a Process. (At most one Process can hold a resource)

The graphs represent the current state of NEEDS and HOLDS. There may be more Needs than those known currently (each requires an executed open) before a Process can proceed.

To Detect Deadlock requires an analysis of the state of Processes. But keeping track of all the resources HELD or NEEDED by all Processes may be unwieldy. In practice. Before analysis is done something must occur to initiate it ex. A Processes in the wait or blocked queue

* If process P_a tries to get a HOLD on resource, R, but R is already HELD process, P_b then P_a NEEDs R.
There is another definition of Need called U-NEEDs. For a set of active processes holding at least 1 resource, all additional resources required for a process to continue running, even if all have not yet been requested, constitute its U(limate)-NEEDs. U-NEEDS become important when one tries to AVOID deadlock.
DEADLOCK DETECTION:
By Array Analysis: If, after "removing" all Processes which can get the Resources they are known to NEED, any Processes remains. (If a Process's row NEEDS are all AVAILABLE it can be "removed").

DEADLOCK RECOVERY
By Preemption remaining Processes can be removed.

ARRAY REPRESENTATION
Two 2-D Arrays: Process's: NEEDS, and HOLDS. Two 1-D Arrays, TOTALs, and AVAILABLE, were: NEEDS All resources a Process has requested but has not been granted still needs before it can continue. HOLDS: All Resources actually given a Process to holds till it runs; TOTAL: Total number of each resource in the entire system, including all NEEDs and HOLDS. AVAILABLE: System Resources not currently held or recognized as needed by any Process

DETECTION-RECOVERY EQUIVALENT RESOURCES
* The ORDER in which removable Processes Needs are satisfied, and removed DOES NOT MATTER because: Assume the graph is in state S and a number of Processes are removable. If any one is removed -and all its Needs and Holds returned then certainly all those that were previously removable are again removable (plus perhaps some others).

ARRAY DEADLOCK IMPLIES CYCLE(S). After all Runnable Processes are "removed", start at any Process P1. all instances of at least one resource it Needs are unavailable because all its instances are Held. Let one of those be Held by P2. Then all instances of at least one resource that P2 Needs must also be Held. This must continue until a a resource Held by a Pj already in the path, thus forming a cycle.

© M C Paull 2005
This is an example of the existence of a cycle in the Needs-Holds graph in which there is more than one Resource [Equivalent Resources] of a given type. The number of resources of each type available at each at any time are indicated by an integer in the Needs vertex. This example shows that a cycle, even involving Resources all of whose equivalents are either Needed or Held does not in itself imply the existence of Deadlock. (However if there is Deadlock then there must be a cycle-and the Hold edge of every $n>1$ copy resources on that cycle for which there are $n$ copies must be on a cycle.)

**EXAMPLE OF CYCLE BUT NO DEADLOCK**

© M C Paull 2005
PROBLEM:
Given a set of HOLDS and U-NEEDS, called state S, and a request for one of the U_NEEDs, to become a current HOLD, and go into state S’. Will satisfying this request inevitably lead to Deadlock? If it will then don’t allow it. If not, that is, if all U_NEEDS can be satisfied, S’ is called a SAFE state.

ALGORITHM:
a) Make a copy of the original ARRAYS + the requested HOLDS - called COPY.
b) Now perform the following operations on COPY (Same As Deadlock-2):
   1. If AVAILABLE, Resources are sufficient to satisfy the ADDITIONAL U-NEEDs of any Process, P, then these are satisfied and P is removed. This leads to the release of Resources and increases those AVAILABLE. Now repeat b) until all Processes are removed (S’ is SAFE), or at any point there are remaining Processes which cannot be done (UNSAFE).
   2. If this cannot be done the state, S’, is UNSAFE.

BASIC PROBLEM: HOW CAN ALL ADDITIONAL U_NEEDS BE KNOWN?

DEADLOCK AVOIDANCE (Banker’s Algorithm)
Before P_b is given R_2, there is no deadlock, I is a SAFE state. Now we will test whether, if P_b is given R_2, (reducing P_b's availability to 1), going to state I', there is still no deadlock i.e. is I' SAFE? This is verified by doing Detection on I'. P_c can be done, making R_5 available and leaving R_3 available. Then P_a can be done, leaving R_3 available and making R_1 available. Then P_d can be done. That gives only P_b remaining-clearly it can complete its I/O and finally in state II all are completed.

In the case below we have the same initial condition except that now R_2 has a total of 1 resource. Give R_2 to P_b and do Detection to test whether the resultant state is SAFE

Now P_c and then P_a can be complete its I/O, but P_d cannot be done and then neither can P_b be done.
As in Dining Philosophers: A Process \((\phi_1)\) requests **all** resources needed to progress and eventually release them. This is an atomic request to the OS it is only \((\phi_2)\) granted if **all** are available.

By GROUPING (TWO-PHASE \((\phi_1,\phi_2)\) LOCKING):

**Prevention Procedure**
(1) Assign a distinct number to each resource (Copies of the same resource should each be given different numbers). \(<1, 2,...j, j+1,..., j+k,... n>\)

(2) Processes request HOLDs on resources by number. **The lowest numbered resource needed is requested first, the next highest numbered one next, etc.**

**Consequences:**
If there is a cycle then we can start at any Resource vertex, say R on a simple cycle. Follow the directed edges and the trip must return to R. So start on Resource vertex, R, numbered \(j\). The path, P, starting on S will, after going through one Process vertex, go to a Resource vertex numbered \(j + k_1, k > 0\). Then this Resource is held by another process Q, and Q also Needs another Resource which must be numbered \(j+k_1+k_2\). **Alternate vertices on path P will have Resource vertices with a higher number than the previous one.** So if one, numbered \(x\) is Held by a process, Z, and the next, if there is one, must be Needed by \(Z\) and so be numbered\(>x\). So P can never return to the initial Resource vertex S.

Note: Any fixed ordering of requests for Resources amongst those requesting the same Resources would do. This might be at least partially possible in compilation.

**EXAMPLE**

<table>
<thead>
<tr>
<th>Cannot Be Deadlocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>open(R1)</td>
</tr>
<tr>
<td>open(R2)</td>
</tr>
<tr>
<td>go</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>open(R2)</td>
</tr>
<tr>
<td>open(R3)</td>
</tr>
<tr>
<td>go</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>open(R3)</td>
</tr>
<tr>
<td>open(R4)</td>
</tr>
<tr>
<td>go</td>
</tr>
<tr>
<td>P4</td>
</tr>
<tr>
<td>open(R4)</td>
</tr>
<tr>
<td>open(R5)</td>
</tr>
<tr>
<td>go</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>open(R1)</td>
</tr>
<tr>
<td>open(R2)</td>
</tr>
<tr>
<td>open(R3)</td>
</tr>
<tr>
<td>go</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>open(R1)</td>
</tr>
<tr>
<td>open(R2)</td>
</tr>
<tr>
<td>open(R4)</td>
</tr>
<tr>
<td>go</td>
</tr>
</tbody>
</table>

DEADLOCK PREVENTION

© M C Paull 2005

9
If there are multiple copies of the same resource each with the same number, R, deadlock can occur. However, if two different processes wanted this resource and both asked for R_1, one would have to wait for the other to finish, even though R_2 is available.

Some consequences of prevention by numbering:
Monitor Trouble
procedure1(...)  
{ <PROG!1>
  wait(x)
  <PROG!2>
  signal(y)
  <PROG!3>
}

procedure2(...)  
{ <PROG!21>
  wait(y)
  <PROG!22>
  signal(z)
}

procedure3(...)  
{ <PROG!31>
  wait(z)
  <PROG!32>
  signal(x)
}

End Monitor

Monitor Deadlock-Can Resources be Numbered and accessed in order?

Philosopher pick up 1 chopstick at a time-so a Philosopher can hold 0, 1 or 2 chopsticks. This can lead to Deadlock (as shown earlier) unless they choose the chopsticks to pick up in numerical order.

A single printer can be given to more than 1 Process--actually they are each given buffering space which will be emptied into printer when filled. The resource is the buffers. Since these are not called by name Deadlock is possible-When all buffers are filled and each Process needs an additional buffer which is already Held by another process-See the deadlock example for C in figure 1.
Deadlock is illustrated graphically.

Two process segments are shown on the x and y axes. They are in separate processes Q and P. The red area represents regions impossible to enter. If Q and P proceed or are interleaved in an order which brings them into the grey area in (a), neither can proceed; they are deadlocked.

This can be prevented in one of two ways, as in b) by all processes always requesting shared resources in the same order, or as in c by making all requests necessary for progressing be requested in one operation.

This could only have been avoided if we knew about all 4 Needs before determining the order in which to make them holds.