Deadlocks can occur when several processes compete for a finite number of shared resources and a waiting process is unable to change state because the needed resources are held by other process(es).

System model resources that may be used by more than one process are the whole problem.

The resources will be utilized in the following sequence

request must wait if not available
use obvious
release obvious that it must be done promptly
Characterization These are necessary conditions and we will later learn they are not sufficient for deadlock to occur. If a deadlock is to happen, these must be simultaneous:

Mutual exclusion
Hold and wait
No preemption
Circular wait

Resource-Allocation Graph has a set of vertices $V$ and edges $E$.

Vertices will be either $P$ for one of $n$ processes or $R$ for one of $m$ resources.

Edges will be requests $P_i \rightarrow R_j$.

Assignment edges will be $R_i \rightarrow P_j$. 

Slides 9 through 12 of text slides
Observations or obvious?

No cycles, no deadlocks.

At least one cycle, one instance per resource type, and it is requested by more than one process, then deadlock.

If more than one instance ... then deadlock is possible.

How to handle deadlocks The three obvious choices are

• ensure that the system never enters a state where deadlock is possible

• (don’t do previous) and watch for deadlock and if it happens provide means for escape, backtrack, ...

• Be happy, don’t worry, ...
Prevention of deadlocks Based on the four necessary conditions

Mutual exclusion must be applied to sharable resources

Hold and wait To avoid this one could require a process to get all its necessary resources to continue. If it can’t, then it must return all.
Can lead to starvation or at least slowness.

No preemption watch this and look at previous.
  • If needs cannot be met, return all resources not in use.
  • Returned resources are added to the list of needed for this process.
  • Process can be restarted only if it can reclaim and then get the needed resources.

Circular wait Wow! Require that each process requests resources in an increasing order of enumeration.
**Avoidance** Require additional information.

- Simplest is to require maxima for each resource.
- Whenever there is a request then evaluate . . .
- . . . (The Bankers Algorithm next time.)

**Safe State** A system is in a **safe state** if from the current state there is a sequence of execution where one process can finish, with its resources added another can finish, . . .

- If a system is in a safe state, then no deadlocks
- If a system is in an unsafe state, deadlock is possible
- Avoidance is insuring that a system will never enter an unsafe state
Banker’s Algorithm

Similar to how a S&L may loan money to homebuilders.

Require each customer to declare maximums and make sure that enough cash is always kept on hand to be able to finish some projects as they progress.

Needed data structures

*available* array of length $m$, the number of resources. Its initial values are the numbers of each of the resources and it will be reduced as resources are allocated to each process.

*max* an $n \times m$ matrix of the maximum the $i^{th}$ process may request of the $j^{th}$ resource

*allocation* an $n \times m$ matrix of the currently allocated resources to processes

*need* an $n \times m$ matrix of the number of each resource each process (row) needs of the given resources.

Then later we need

*request* an $n \times m$ matrix of the number of the $j^{th}$ resource the $i^{th}$ process requests.

*work* a vector of length $m$, like *available*
finish a vector of length $n$ used to keep up with processes that have finished.
Safety Algorithm

1. Initialize work to available and all elements of finish to false. available has been established based on the initial allocations and is updated as we progress.

2. Find an $i$ such that $\text{finish}[i]$ is false and the $i^{th}$ row of need is term by term less than work.
   If none exists then go to step 4.

3. Since we found one, update work and finish
   \[
   \text{work} \leftarrow \text{work} + \text{allocation}
   \]
   \[
   \text{finish}[i] \leftarrow \text{true}
   \]
   go to step 2 and repeat it

4. If all elements of finish are true then the state is safe, otherwise it is not!
Resource-Request Algorithm

The $n \times m$ matrix need is a convinence and is really $max$ less allocation. The request is similar but is a part of the need being requested at this time.

1. if $request_i \leq need_i$, go to step 2. If not, something is wrong because too many have been requested compared to the original declaration.

2. if $request_i \leq available$ go to step 3. Otherwise, $P_i$ must wait!

3. Pretend to allocate and update available, allocation, and need by appropriate addition and subtraction of request

   $available \leftarrow available - request_i$
   $allocation_i \leftarrow allocation_i + request_i$
   $need_i \leftarrow need_i - request_i$

   Remember need is $max$ less allocation

   If the new state is safe then . . .
   if it is not then the previous state must be restored and $P_i$ must wait