Resource Allocation and Deadlock Handling
Conditions for Deadlock

[Coffman-etal 1971] 4 conditions must hold simultaneously for a deadlock to occur:

- **Mutual exclusion**: only one process at a time can use a resource.

- **Hold and wait**: a process holding some resource can request additional resources and wait for them if they are held by other processes.

- **No preemption**: a resource can only be released voluntarily by the process holding it, after that process has completed its task.
  - Q: examples preemptable/non-preemptable resources?

- **Circular wait**: there exists a circular chain of 2 or more blocked processes, each waiting for a resource held by the next process in the chain.
Resource Allocation & Handling of Deadlocks

- **Structurally restrict** the way in which processes request resources
  - **deadlock prevention**: deadlock *is not* possible

- Require processes to give **advance info** about the (max) resources they will require; then **schedule processes** in a way that avoids deadlock.
  - **deadlock avoidance**: deadlock *is* possible, but OS uses advance info to avoid it

- Allow a deadlock state and then **recover**

- **Ignore** the problem and pretend that deadlocks never occur in the system (can be a “solution” sometimes?!...)


Resource Allocation with Deadlock Prevention

Restrain the ways requests can be made; attack at least one of the 4 conditions, so that deadlocks are impossible to happen:

- **Mutual Exclusion** - (cannot do much here ...)

- **Hold and Wait** - must guarantee that when a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources at once or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

- **No Preemption** - If a process holding some resources requests another resource that cannot be immediately allocated, it releases the held resources and has to request them again (risk for starvation).

- **Circular Wait** - impose total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration (e.g. first the tape, then the disk).

- Examples?
Fight the circular wait:
Dining philosophers example

request forks in increasing fork-id
var f[0..n]: bin-semaphore /init all 1 /

P_i: (i!=n)
Repeat
  Wait(f[i])
  Wait(f[(i+1)modn])
Eat
Signal(f[(i+1)modn])
Signal(f[i])
Think
forever

Pn
Repeat
  Wait(f[(i+1)modn])
  Wait(f[i])
Eat
Signal(f[i])
Signal(f[(i+1)modn])
Think
forever

Idea:
• Hierarchical ordering of resources
• Proc’s request their needed resources in increasing order
Fight the hold and wait: Dining philosophers example

semaphore S[N]
int state[N]

\[Pi\]: do
  <think>
  take_forks(i)
  <eat>
  leave_forks(i)
forever

test(i)
  if state(i) == HUNGRY && state(left(i)) != EATING && state(left(i)) != EATING then
    state(i) := EATING
    signal(S[i])

\[\text{take\_forks}(i)\]
  wait(mutex)
  state(i) := HUNGRY
  test(i)
  signal(mutex)
  wait(S[i])

\[\text{leave\_forks}(i)\]
  wait(mutex)
  state(i) := THINKING
  test(left(i))
  test(right(i))
  signal(mutex)

Idea: apply mutex algorithm for each neighbourhood, instead of for each fork.
Fight the no-preemption: Dining philosophers example

```plaintext
var f[0..n]: record
    s: bin-semaphore
    available: boolean /init all 1 /

P_i:
Repeat
    While <not holding both forks> do
        Lock(f[i])
        If !trylock(f[(i+1) mod n]) then release f[i];
    od
    Eat
    Release(f[i])
Release(f[(i+1) mod n])
Think
forever

trylock(fork):
    wait(fork.s)
    If fork.available then
        fork.available := false
        ret := true
    else ret := false
    Signal(fork.s)
    Return(ret)

Lock(fork):
Repeat
    Until (trylock(fork))
Release(fork):
    wait(fork.s)
    fork.available := true
    Signal(fork.s)
```

Idea: release held resources and retry when the next one is not available
Deadlock avoidance: System Model

- Resource types $R_1, R_2, \ldots, R_m$
  - e.g. CPU, memory space, I/O devices, files
  - each resource type $R_i$ has $W_i$ instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Resource-Allocation Graph

A set of vertices $V$ and a set of edges $E$.
- $V$ is partitioned into two sets:
  - $P = \{P_1, P_2, \ldots, P_n\}$ the set of processes
  - $R = \{R_1, R_2, \ldots, R_m\}$ the set of resource types
- request edge: $P_i \rightarrow R_j$
- assignment edge: $R_j \rightarrow P_i$
Example of a Resource Allocation Graph

\[ R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_4 \]
Resource Allocation Graph With A cycle but no Deadlock
Basic Facts

• graph contains no cycles $\Rightarrow$ no deadlock.
  (i.e. cycle is always a necessary condition for deadlock)

• If graph contains a cycle $\Rightarrow$
  
  - if one instance per resource type, then deadlock.
  
  - if several instances per resource type, then possibility of deadlock

  • Thm: if immediate-allocation-method, then knot $\Rightarrow$ deadlock.
    
    - Knot = knot - strongly connected subgraph (no sinks) with no outgoing edges
Resource Allocation with Deadlock Avoidance

Requires *a priori information* available.

- e.g.: each *process* declares *maximum number of resources* of each type that it *may need* (e.g. memory/disk pages).

**Deadlock-avoidance algo:**

- *examines the resource-allocation state...*
  - *available and allocated resources*
  - *maximum possible demands* of the processes.

- ...to ensure there is *no potential for a circular-wait*:
  - *safe state* ⇒ no deadlocks in the horizon.
  - *unsafe state* ⇒ deadlock might occur (later...)
  - Q: how to do the safety check?

- *Avoidance* = ensure that *system will not enter an unsafe state.*

  **Idea:** If *satisfying a request* will result in an *unsafe state*, the requesting process is *suspended* until *enough resources are free-ed* by processes that will terminate in the meanwhile.
Claim edge $P_i \rightarrow R_j$: $P_j$ may request resource $R_j$
  - represented by a dashed line.

Claim edge converts to request edge when a process requests a resource.

When a resource is released by a process, assignment edge reconverts to a claim edge.

Resources must be claimed *a priori* in the system.
Example Resource-Allocation Graph For Deadlock Avoidance: Safe State
Example Resource-Allocation Graph For Deadlock Avoidance: Unsafe State
Safety checking: More on Safe State

**safe state** = there exists a *safe sequence* \(<P_1, P_2, ..., P_n>\) of terminating all processes:

- for each \(P_i\), the requests that it can still make can be granted by currently available resources + those held by \(P_1, P_2, ..., P_{i-1}\)

- The system can schedule the processes as follows:
  - if \(P_i\)'s resource needs are not immediately available, then it can
    - wait until all \(P_1, P_2, ..., P_{i-1}\) have finished
    - obtain needed resources, execute, release resources, terminate.
  - then the next process can obtain its needed resources, and so on.
Banker's Algorithm for Resource Allocation with Deadlock Avoidance

Data Structures:
- **Max**: $n \times m$ matrix.
  - $Max[i,j] = k$: $P_i$ may request max $k$ instances of resource type $R_j$.
- **Allocation**: $n \times m$ matrix.
  - $Allocation[i,j] = k$: $P_i$ is currently allocated $k$ instances of $R_j$.
- **Available**: length $m$ vector
  - $available[j] = k$: $k$ instances of resource type $R_j$ available.
- **Need**: $n \times m$ matrix:

RECALL: **Avoidance** = ensure that system will not enter an unsafe state.

**Idea**: If satisfying a request will result in an unsafe state, then requesting process is suspended until enough resources are free-ed by processes that will terminate in the meanwhile.
For each new Request, do /*Request, [j] = k: P, wants k instances of R, */
/* Check consequence if request is granted */

remember the current resource-allocation state;
Available := Available - Request;
Allocation := Allocation + Request;
Need := Need - Request;
If safety-check OK ⇒ the resources are allocated to P;
Else ( unsafe ) ⇒
    P, must wait and
    the old resource-allocation state is restored;
Banker’s Algorithm: safety check

- **Work** and **Finish**: auxiliary vectors of length \( m \) and \( n \), respectively.
- **Initialize:**
  
  \[
  \text{Work} := \text{Available} \\
  \text{Finish}[i] = \text{false} \text{ for } i = 1, 2, \ldots, n.
  \]

- **While** there exists \( i \) such that both
  
  \[
  (a) \text{Finish}[i] = \text{false} \\
  (b) \text{Need}_i \leq \text{Work}.
  \]

  do

  \[
  \text{Work} := \text{Work} + \text{Allocation}_i \\
  \text{Finish}[i] := \text{true}
  \]

- If **Finish** \([i] = \text{true} \) for all \( i \), then the system is in a safe state
  
  else state is unsafe
Very simple example execution of Bankers Algo (snapshot 1)

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>A B</td>
<td>A B</td>
<td>A B</td>
</tr>
<tr>
<td>$P_1$</td>
<td>1 0</td>
<td>11</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>$P_2$</td>
<td>0 0</td>
<td>11</td>
<td>1 1</td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence $< P_1, P_2 >$ satisfies safety criteria.
**Very simple example execution of Bankers Algo (snapshot 2)**

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>A B</td>
<td>A B</td>
<td>A B</td>
</tr>
<tr>
<td>P₁</td>
<td>1 0</td>
<td>11</td>
<td>0 1</td>
</tr>
<tr>
<td>P₂</td>
<td>0 1</td>
<td>11</td>
<td>1 0</td>
</tr>
</tbody>
</table>

- Allocating B to P₂ leaves the system in an **unsafe state** since there is no sequence that satisfies safety criteria (**Available** vector is 0!).
Deadlock Detection & Recovery

- Allow system to enter deadlock state
- Detection algorithm: what we did for checking safety in enhanced graph, now we do it in the resource allocation graph
  - Using resource-allocation graphs
  - Using Banker’s algo idea
- Recovery scheme
Deadlock Detection

**Note:**
- similar as detecting unsafe states using Banker’s algo
- Q: how is similarity explained?
- Q: if they cost the same why not use avoidance instead of detection&recovery?

Data structures:
- **Available:** vector of length \( m \): number of available resources of each type.
- **Allocation:** \( n \times m \) matrix: number of resources of each type currently allocated to each process.
- **Request:** \( n \times m \) matrix: current request of each process. Request \([ij] = k\): \( P_i \) is requesting \( k \) more instances of resource type \( R_j \).
Detection-Algorithm Usage

• When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?

• If algorithm is invoked arbitrarily, there may be many cycles in the resource graph ⇒ we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
Recovery from Deadlock: (1) Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until deadlock is eliminated.
- In which order should we choose to abort? Criteria?
  - effect of the process' computation (breakpoints & rollback)
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used/needs to complete.
  - How many processes will need to be terminated.
Recovery from Deadlock: (2) Resource Preemption

- Select a **victim**
  - minimize cost.
- **Rollback** - return to some safe state, restart process from that state
  - Must do checkpointing for this to be possible.
- **Watch for starvation** - same process may always be picked as victim, include number of rollbacks in cost factor.
Combined Approach to Deadlock Handling

- **Combine** the three basic approaches (prevention, avoidance, detection), allowing the use of the optimal approach for each type of resources in the system:
  - **Partition resources** into hierarchically ordered classes (deadlocks may arise only within each class, then)
  - use most appropriate technique for handling deadlocks within each class, e.g:
    - internal (e.g. interactive I/O channels): *prevention by ordering*
    - process resources (e.g. files): *avoidance by knowing max needs*
    - main memory: *prevention by preemption*
    - swap space (blocks in disk, drum, ...): *prevention by preallocation*
RA & Deadlock Handling in Distributed Systems

• **Note:** *no centralized control here!*
  - Each site only knows about its own resources
  - Deadlock may involve distributed resources
Resource Allocation in Message-Passing Systems

**Prevention** (recall strategies: no cycles; request all resources at once; apply preemptive strategies) (apply in gen. din.phil)

- using *priorities/hierarchical ordering* of resources
  - Use resource-managers (1 proc/resource) and request resource from each manager (in the hierarchy order)
  - Use mutex (each fork is a mutex, execute Rikart&Agrawala for each)

- **No hold&wait:**
  - Each process is mutually exclusive with both its neighbours => each group of 3 neighbours is 1 Rikart&Agrawala "instance"

- **No Preemption** - If a process holding some resources requests another resource that cannot be immediately allocated, it releases the held resources and has to request them again (risk for starvation: cf extra, optional reference, PetersonStyer (not included in study material) algo for avoiding starvation).
Distributed R.A. with Deadlock Avoidance or Deadlock Detection & Recovery

- **Centralized control** - one site is responsible for safety check or deadlock detection
  - Can be a bottleneck (in performance and fault-tolerance)
- **Distributed control** - all processes cooperate in the safety check or deadlock detection function
  - need of *consistent global state*
  - straightforward (expensive) approach: all processes try to learn global state
  - less expensive solutions tend to be complicated and/or unrealistic

- **Distributed deadlock avoidance or detection & recovery is not very practical**
  - Checking global states involves considerable processing overhead for a distributed system with a large number of processes and resources
  - Also: who will check if procs are all blocked?!