Distributed 2PC & Deadlock

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 36
Nov 21, 2014

Reading: OSC Ch 7 (deadlock)
Consistency Review

• Problem: shared state replicated across multiple clients, do they see a consistent view?
  – Propagation: Writes become visible to reads
  – Serializability: The order of writes seen by each client’s series of reads and writes is consistent with a total order
    • As if all writes and reads had been serviced at a single point
    • The total order is not actually generated, but it could be

• Many distributed systems provide weaker semantics
  – Eventual consistency
### In Everyday Life

<table>
<thead>
<tr>
<th>Where do we meet?</th>
<th>Where do we meet?</th>
<th>Where do we meet?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>At Nefeli’s</td>
<td>At Top Dog</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Where do we meet?</td>
<td>Where do we meet?</td>
<td>Where do we meet?</td>
</tr>
<tr>
<td>At Nefeli’s</td>
<td>At Nefeli’s</td>
<td>At Nefeli’s</td>
</tr>
<tr>
<td>At Top Dog</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Alternative: timestamp every write, present entire log in timestamp order, with tie breaker

11/21/14
Unfinished Business: Multiple Servers

- What happens if cannot update all the replicas?
- Availability => Inconsistency
Durability and Atomicity

• How do you make sure transaction results persist in the face of failures (e.g., server node failures)?

• Replicate store / database
  – Commit transaction to each replica

• What happens if you have failures during a transaction commit?
  – Need to ensure atomicity: either transaction is committed on all replicas or none at all
Two Phase (2PC) Commit

• 2PC is a distributed protocol

• High-level problem statement
  – If no node fails and all nodes are ready to commit, then all nodes **COMMIT**
  – Otherwise **ABORT** at all nodes

• Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)
2PC Algorithm

- One coordinator
- N workers (replicas)

High level algorithm description
  - Coordinator asks all workers if they can commit
  - If all workers reply “VOTE-COMMIT”, then coordinator broadcasts “GLOBAL-COMMIT”,
  - Otherwise coordinator broadcasts “GLOBAL-ABORT”
  - Workers obey the GLOBAL messages
Coordinator Algorithm

Coordinator sends VOTE-REQ to all workers

- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn’t receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

- Wait for VOTE-REQ from coordinator
- If ready, send VOTE-COMMIT to coordinator
  - And immediately abort
- If not ready, send VOTE-ABORT to coordinator
- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort
Failure Free Example Execution

coordinator

worker 1

worker 2

worker 3

time

VOTE-REQ

VOTE-COMMIT

GLOBAL-COMMIT
State Machine of Coordinator

- Coordinator implements simple state machine

```
<table>
<thead>
<tr>
<th>State</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td>Recv: START, Send: VOTE-REQ</td>
</tr>
<tr>
<td></td>
<td>Recv: VOTE-ABORT, Send: GLOBAL-ABORT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Recv: all VOTE-COMMIT, Send: GLOBAL-COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>
```
State Machine of Workers

INIT
- Recv: VOTE-REQ
- Send: VOTE-ABORT

READY
- Recv: VOTE-REQ
- Send: VOTE-COMMIT
- Recv: GLOBAL-ABORT
- Recv: GLOBAL-COMMIT

ABORT

COMMIT
Dealing with Worker Failures

• How to deal with worker failures?
  – Failure only affects states in which the node is waiting for messages
  – Coordinator only waits for votes in “WAIT” state
  – In WAIT, if doesn’t receive N votes, it times out and sends GLOBAL-ABORT
Example of Worker Failure

Coordinator

Worker 1

Worker 2

Worker 3

VOTE-REQ

VOTE-COMMIT

INIT

WAIT

ABORT

COMM

Global-Abort

Timeout
Dealing with Coordinator Failure

• How to deal with coordinator failures?
  – worker waits for VOTE-REQ in INIT
    • Worker can time out and abort (coordinator handles it)
  – worker waits for GLOBAL-* message in READY
    • If coordinator fails, workers must **BLOCK** waiting for coordinator
to recover and send GLOBAL_* message
Example of Coordinator Failure #1

Diagram:

- **Coordinator**
- **Worker 1**
- **Worker 2**
- **Worker 3**

- **INIT** → **READY** → **ABORT** → **COMM**
- **Abort Requests** from Workers:
  - **Worker 1**→ **timeout**
  - **Worker 2**→ **timeout**
  - **Worker 3**→ **timeout**

- **VOTE-REQ** to Coordinator
- **VOTE-ABORT** from Coordinator
Example of Coordinator Failure #2

INIT

READY

ABORT

COMM

VOTE-REQ

VOTE-COMMIT

GLOBAL-ABORT

worker 1

worker 2

worker 3

restart

cordinator

block waiting for coordinator

coordinator restarted
Durability

• All nodes use stable storage* to store which state they are in

• Upon recovery, it can restore state and resume:
  – Coordinator aborts in INIT, WAIT, or ABORT
  – Coordinator commits in COMMIT
  – Worker aborts in INIT, ABORT
  – Worker commits in COMMIT
  – Worker asks Coordinator in READY

* - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
Blocking for Coordinator to Recover

- A worker waiting for global decision can ask fellow workers about their state
  - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
  - Thus, worker can safely abort or commit, respectively
  - If another worker is still in INIT state, then both workers can decide to abort
  - If all workers are in ready, need to BLOCK (don’t know if coordinator wanted to abort or commit)
Admin Break

- MidTerm (mult by 4/3)
What’s a Deadlock?

- Situation where all entities (e.g., threads, clients, ...) have acquired certain resources and need to acquire additional resources,
  - but those additional resources are held by some other entity that won’t release them
Bridge Crossing Example

- Each segment of road can be viewed as a resource
  - Car “owns” the segment under them
  - Must acquire segment that they are moving into
- Must acquire both halves of bridge to cross
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast \(\Rightarrow\) no one goes west
OS analog of the bridge

• Exclusive Access to Multiple Resources:

\[ x=1, \quad y=1 \]

\[
\begin{align*}
\text{Thread A} & \\
& x.\text{Down}(); \\
& y.\text{Down}(); \\
& \ldots \\
& y.\text{Up}(); \\
& x.\text{Up}();
\end{align*}
\]

\[
\begin{align*}
\text{Thread B} & \\
& y.\text{Down}(); \\
& x.\text{Down}(); \\
& \ldots \\
& x.\text{Up}(); \\
& y.\text{Up}();
\end{align*}
\]

• Say, \( x \) is free-list and \( y \) is directory

Deadlock

A: \( x.\text{Down}(); \)
B: \( y.\text{Down}(); \)
A: \( y.\text{Down}(); \)
B: \( x.\text{Down}(); \)
\ldots
Deadlock vs. Starvation

- **Deadlock**: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
  - Thread B owns Res 2 and is waiting for Res 1

- **Starvation**: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads

- Deadlock $\Rightarrow$ Starvation, but not vice versa
  - Starvation can end (but doesn’t have to)
  - Deadlock can’t end without external intervention
OS analog of the bridge

• Exclusive Access to Multiple Resources:
  \(x=1, \ y=1\)

  Thread A                      Thread B
x.\text{Down}();             y.\text{Down}();
y.\text{Down}();             x.\text{Down}();
...                          ...
y.\text{Up}();              x.\text{Up}();
x.\text{Up}();              y.\text{Up}();

• Say, \(x\) is free-list and \(y\) is directory structure

• Deadlock is typically not deterministic
  – Timing in this example has to be “just so”

• Deadlocks occur with multiple resources
  – Can’t solve deadlock for each resource independently
Can this deadlock?

void transaction(account *from, account *to, double amount) {
    acquire(from->lock);
    acquire(to->lock);
    withdraw(from, amount);
    deposit(to, amount);
    release(from->lock);
    release(to->lock);
}

• Under what conditions?
Dining Philosophers Problem

- **N** chopsticks/ N philosophers
  - Need two chopsticks to eat
  - Free for all: Philosopher will grab any one they can
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
Four requirements for Deadlock

- **Mutual exclusion**
  - Only one thread at a time can use a resource

- **Hold and wait (incremental allocation)**
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads

- **No preemption**
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

- **Circular wait**
  - e.g., There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads,
    - \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    - \( T_2 \) is waiting for a resource that is held by \( T_3 \), \( \ldots \)
    - \( T_n \) is waiting for a resource that is held by \( T_1 \)
Methods for Handling Deadlocks

• Deadlock **prevention**: design system to ensure that it will *never* enter a deadlock
  – E.g., monitor all lock acquisitions
  – Selectively deny those that *might* lead to deadlock

• Allow system to enter deadlock and then recover
  – Requires deadlock **detection** algorithm
    • E.g., Java JMX `findDeadlockedThreads()`
  – Some technique for forcibly preempting resources and/or terminating tasks

• Ignore the problem and hope that deadlocks never occur in the system
  – Used by most operating systems, including UNIX
  – Resort to manual version of recovery
Techniques for Deadlock Prevention

• Eliminate the Shared Resources
  – E.g., give each Philosopher two chopsticks, open the other bridge lane, ...
  – Or at least two virtual chopsticks
  – OK, if sharing was do to resource limitations
  – Not if sharing is due to true interactions
    • Must modify Directory Structure AND File Index AND the Block Free list
    • Must enter the intersection to turn left
Techniques for Deadlock Prevention

• Eliminate the Shared Resources
• Eliminate the Mutual Exclusion
  – E.g., many processes can have read-only access to file
  – But still need mutual-exclusion for writing
Techniques for Deadlock Prevention

- Eliminate the Shared Resources
- Eliminate the Mutual Exclusion
- Eliminate Hold-and-Wait
Acquire all resources up front

• Philosopher grabs for both chopsticks at once
  – If not both available, don’t pickup either, try again later
• Phone call signaling attempts to acquire resources all along the path, “busy” if any point not available
• File Systems: lock {dir. Structure, file index, free list}
  – Or the piece of each in a common block group
• Databases: lock all tables touched by the query
• Hard in general, but often natural in specific cases
Techniques for Deadlock Prevention

• Eliminate the Shared Resources
• Eliminate the Mutual Exclusion
• Eliminate Hold-and-Wait
• Permit pre-emption
Incremental Acquisition with Pre-emption

- Philosopher grabs one, goes for other, if not available, releases the first
  - Analogous for sequence of system resources
- Danger of turning deadlock into livelock
  - Everyone is grabbing and releasing, no one every gets two
- Works great at low utilization
  - Potential for thrashing (or failure) as utilization increases
- Similar to CSMA (carrier sense multiple access) in networks
- Randomize the back-off
Techniques for Deadlock Prevention

- Eliminate the Shared Resources
- Eliminate the Mutual Exclusion
- Eliminate Hold-and-Wait
- Permit pre-emption
- Eliminate the creation of circular wait
  - Dedicated resources to break cycles
  - Ordering on the acquisition of resources
Cyclic Dependence of resources

- Suppose everyone grabs left first
- Acquisition of the right chopstick depends on the acquisition of the left one
- A cycle of dependences forms
Ordered Acquisition to prevent cycle from forming

- Suppose everyone grabs lowest first
- Dependence graph is acyclic
- Someone will fail to grab chopstick 0!
- How do you modify the rule to retain fairness?
- OS: define ordered set of resource classes
  - Acquire locks on resources in order
  - Page Table => Memory Blocks => ...

11/21/14 cs162 fa14 L35
Deadlock Detection

- There are threads that never become ready
- Are they deadlocked or just ... ?
A Simple Resource Graph

• System Model
  – A set of Threads $T_1, T_2, \ldots, T_n$
  – Resource types $R_1, R_2, \ldots, R_m$
    * locks in this case
  – Each thread utilizes a resource as follows:
    * Request() / Use() / Release()

• Resource-Allocation Graph:
  – V is partitioned into two types:
    * $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    * $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  – request edge – directed edge $T_i \rightarrow R_j$
    * Wait-List
  – assignment edge – directed edge $R_j \rightarrow T_i$
    * Owns
Resource Allocation Graph Examples

Simple Resource Allocation Graph

Deadlocked Resource Allocation Graph

11/21/14 cs162 fa14 L35 40
How would you look for cycles?
Resource Allocation Graph Examples

Simple Resource Allocation Graph

Deadlocked Resource Allocation Graph
How would avoid cycle creation?

• On attempt to acquire an owned lock
  – Check to see if adding the request edge would create a cycle
More General Case

• Each resources has a capacity (# instances)
• Each thread requests a portion of each resource
General Resource-Allocation Graph

• System Model
  – A set of Threads $T_1, T_2, \ldots, T_n$
  – Resource types $R_1, R_2, \ldots, R_m$
    \textit{CPU cycles, memory space, I/O devices}
  – Each resource type $R_i$ has $W_i$ instances.
  – Each thread utilizes a resource as follows:
    \begin{itemize}
      \item Request() / Use() / Release()
    \end{itemize}

• Resource-Allocation Graph:
  – $V$ is partitioned into two types:
    \begin{itemize}
      \item $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
      \item $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
    \end{itemize}
  – request edge – directed edge $T_i \rightarrow R_j$
  – assignment edge – directed edge $R_j \rightarrow T_i$
Resource Allocation Graph Examples

- Recall:
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

1. Simple Resource Allocation Graph
2. Allocation Graph With Deadlock
3. Allocation Graph With Cycle, but No Deadlock
Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let $[X]$ represent an $m$-ary vector of non-negative integers (quantities of resources of each type):
    
    - $[\text{FreeResources}]$: Current free resources each type
    - $[\text{Request}_x]$: Current requests from thread $X$
    - $[\text{Alloc}_x]$: Current resources held by thread $X$

  - See if tasks can eventually terminate on their own
    
    $[\text{Avail}] = [\text{FreeResources}]$
    
    Add all nodes to UNFINISHED
    
    do {
      done = true
      Foreach node in UNFINISHED {
        if ($[\text{Request}_{\text{node}}] \leq [\text{Avail}]$) {
          remove node from UNFINISHED
          $[\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{\text{node}}]$
          done = false
        }
      }
    } until(done)

  - Nodes left in UNFINISHED ⇒ deadlocked
Deadlock Detection Algorithm Example

[Request_{T1}] = [1,0]; Alloc_{T1} = [0,1]
[Request_{T2}] = [0,0]; Alloc_{T2} = [1,0]
[Request_{T3}] = [0,1]; Alloc_{T3} = [1,0]
[Request_{T4}] = [0,0]; Alloc_{T4} = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_{node}] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_{node}]
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

[Request\textsubscript{T1}] = [1,0]; Alloc\textsubscript{T1} = [0,1]
[Request\textsubscript{T2}] = [0,0]; Alloc\textsubscript{T2} = [1,0]
[Request\textsubscript{T3}] = [0,1]; Alloc\textsubscript{T3} = [1,0]
[Request\textsubscript{T4}] = [0,0]; Alloc\textsubscript{T4} = [0,1]
[Avail] = [0,0]
UNFINISHED = \{T1,T2,T3,T4\}

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request\textsubscript{T1}] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc\textsubscript{T1}]
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

\[
\begin{align*}
[\text{Request}_{T1}] &= [1,0]; \quad \text{Alloc}_{T1} = [0,1] \\
[\text{Request}_{T2}] &= [0,0]; \quad \text{Alloc}_{T2} = [1,0] \\
[\text{Request}_{T3}] &= [0,1]; \quad \text{Alloc}_{T3} = [1,0] \\
[\text{Request}_{T4}] &= [0,0]; \quad \text{Alloc}_{T4} = [0,1] \\
[\text{Avail}] &= [0,0] \\
\text{UNFINISHED} &= \{T1,T2,T3,T4\}
\end{align*}
\]

do {
  done = true
  Foreach node in UNFINISHED {
    if ([\text{Request}_{node}] \leq [\text{Avail}]) {
      remove node from UNFINISHED
      [\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{node}]
      done = false
    }
  }
} until(done)
Deadlock Detection Algorithm Example

\[\text{Request}_T = [1, 0]; \text{Alloc}_T = [0, 1]\]
\[\text{Request}_T = [0, 0]; \text{Alloc}_T = [1, 0]\]
\[\text{Request}_T = [0, 1]; \text{Alloc}_T = [1, 0]\]
\[\text{Request}_T = [0, 0]; \text{Alloc}_T = [0, 1]\]
\[\text{Avail} = [0, 0]\]
\[\text{UNFINISHED} = \{T_1, T_2, T_3, T_4\}\]

\[\text{do} \{\text{done} = \text{true} \]
\[\text{Foreach node in UNFINISHED} \{\text{if ([Request}_T <= [Avail])} \{\text{remove node from UNFINISHED}\]
\[\text{[Avail]} = \text{[Avail]} + \text{[Alloc}_T\}
\[\text{done} = \text{false}\}
\[\}\text{until(done)\}
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{[Request}_{T_1}] &= [1,0]; \ \text{Alloc}_{T_1} = [0,1] \\
\text{[Request}_{T_2}] &= [0,0]; \ \text{Alloc}_{T_2} = [1,0] \\
\text{[Request}_{T_3}] &= [0,1]; \ \text{Alloc}_{T_3} = [1,0] \\
\text{[Request}_{T_4}] &= [0,0]; \ \text{Alloc}_{T_4} = [0,1] \\
\text{[Avail]} &= [0,0] \\
\text{UNFINISHED} &= \{T_1,T_3,T_4\}
\end{align*}
\]

do {
    done = true
    Foreach node in UNFINISHED {
        if (\text{[Request}_{T_2}] \leq \text{[Avail]}) {
            \text{remove node from UNFINISHED} \\
            \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{T_2]} \\
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{[Request}_{T_1}] & = [1,0]; \quad \text{Alloc}_{T_1} = [0,1] \\
\text{[Request}_{T_2}] & = [0,0]; \quad \text{Alloc}_{T_2} = [1,0] \\
\text{[Request}_{T_3}] & = [0,1]; \quad \text{Alloc}_{T_3} = [1,0] \\
\text{[Request}_{T_4}] & = [0,0]; \quad \text{Alloc}_{T_4} = [0,1] \\
\text{[Avail]} & = [1,0] \\
\text{UNFINISHED} & = \{T_1,T_3,T_4\}
\end{align*}
\]

do {
\begin{align*}
\text{done} & = \text{true} \\
\text{Foreach node in UNFINISHED} & \{ \\
\text{if (}[\text{Request}_{T_2}] & \leq [\text{Avail}]\} \{ \\
\text{remove node from UNFINISHED} & \} \\
\text{[Avail]} & = [\text{Avail}] + [\text{Alloc}_{T_2}] \\
\text{done} & = \text{false} \\
\}
\} \text{ until (done) }
\]
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{[Request}_{T_1}\text{]} &= [1,0]; \quad \text{Alloc}_{T_1} = [0,1] \\
\text{[Request}_{T_2}\text{]} &= [0,0]; \quad \text{Alloc}_{T_2} = [1,0] \\
\text{[Request}_{T_3}\text{]} &= [0,1]; \quad \text{Alloc}_{T_3} = [1,0] \\
\text{[Request}_{T_4}\text{]} &= [0,0]; \quad \text{Alloc}_{T_4} = [0,1] \\
\text{[Avail]} &= [1,0] \\
\text{UNFINISHED} &= \{T_1, T_3, T_4\} \\
\end{align*}
\]

do {
  \text{done} = \text{true} \\
  \text{Foreach node in UNFINISHED} \\
  \quad \text{if} (\text{[Request}_{T_2}\text{]} \leq \text{[Avail]}) \\
  \quad \quad \text{remove node from UNFINISHED} \\
  \quad \quad \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{T_2}\text{]} \\
  \quad \text{done} = \text{false} \\
} until(\text{done})
Deadlock Detection Algorithm Example

\[ \text{[Request}_{T1}] = [1,0]; \text{Alloc}_{T1} = [0,1] \]
\[ \text{[Request}_{T2}] = [0,0]; \text{Alloc}_{T2} = [1,0] \]
\[ \text{[Request}_{T3}] = [0,1]; \text{Alloc}_{T3} = [1,0] \]
\[ \text{[Request}_{T4}] = [0,0]; \text{Alloc}_{T4} = [0,1] \]
\[ \text{[Avail]} = [1,0] \]
UNFINISHED = \{T1,T3,T4\}

\[ \text{do } \{ \]
\[ \quad \text{done} = \text{true} \]
\[ \quad \text{Foreach node in UNFINISHED } \{ \]
\[ \quad \quad \text{if } (\text{[Request}_{node}] \leq \text{[Avail]}) \{ \]
\[ \quad \quad \quad \text{remove node from UNFINISHED} \]
\[ \quad \quad \quad \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{node]} \]
\[ \quad \quad \quad \text{done} = \text{false} \]
\[ \quad \quad \} \]
\[ \} \quad \text{until}(\text{done}) \]
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{[Request}_{T_1}\text{]} &= [1,0]; \text{ Alloc}_{T_1} = [0,1] \\
\text{[Request}_{T_2}\text{]} &= [0,0]; \text{ Alloc}_{T_2} = [1,0] \\
\text{[Request}_{T_3}\text{]} &= [0,1]; \text{ Alloc}_{T_3} = [1,0] \\
\text{[Request}_{T_4}\text{]} &= [0,0]; \text{ Alloc}_{T_4} = [0,1] \\
\text{[Avail]} &= [1,0] \\
\text{UNFINISHED} &= \{T_1,T_3,T_4\}
\end{align*}
\]

do {
\text{done} = \text{true}
\text{Foreach node in UNFINISHED} { \\
\text{if (}} \text{[Request}_{T_3}\text{]} \leq \text{[Avail]} \text{)} { \\
\text{remove node from UNFINISHED} \\
\text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{T_3}\text{]} \\
\text{done} = \text{false} \\
} \\
} until(\text{done})
Deadlock Detection Algorithm Example

\[ \text{Request}_{T_1} = [1,0]; \text{Alloc}_{T_1} = [0,1] \]
\[ \text{Request}_{T_2} = [0,0]; \text{Alloc}_{T_2} = [1,0] \]
\[ \text{Request}_{T_3} = [0,1]; \text{Alloc}_{T_3} = [1,0] \]
\[ \text{Request}_{T_4} = [0,0]; \text{Alloc}_{T_4} = [0,1] \]
\[ \text{Avail} = [1,0] \]
UNFINISHED = \{T1,T3,T4\}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_{node}] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_{node}]
      done = false
    }
  }
} until(done)
Deadlock Detection Algorithm Example

[Request\textsubscript{T1}] = [1,0]; Alloc\textsubscript{T1} = [0,1]
[Request\textsubscript{T2}] = [0,0]; Alloc\textsubscript{T2} = [1,0]
[Request\textsubscript{T3}] = [0,1]; Alloc\textsubscript{T3} = [1,0]
[Request\textsubscript{T4}] = [0,0]; Alloc\textsubscript{T4} = [0,1]
[Avail] = [1,0]
UNFINISHED = \{T1,T3,T4\}

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request\textsubscript{T4}] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc\textsubscript{T4}]
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

\[
\begin{align*}
[\text{Request}_{T_1}] &= [1,0]; \quad \text{Alloc}_{T_1} = [0,1] \\
[\text{Request}_{T_2}] &= [0,0]; \quad \text{Alloc}_{T_2} = [1,0] \\
[\text{Request}_{T_3}] &= [0,1]; \quad \text{Alloc}_{T_3} = [1,0] \\
[\text{Request}_{T_4}] &= [0,0]; \quad \text{Alloc}_{T_4} = [0,1] \\
[\text{Avail}] &= [1,0] \\
\text{UNFINISHED} &= \{T_1,T_3\}
\end{align*}
\]

do {
    done = true
    Foreach node in UNFINISHED {
        if ([\text{Request}_{T_4}] \leq [\text{Avail}]) {
            \text{remove node from UNFINISHED} \\
            [\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{T_4}] \\
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

\[ \text{Request}_{T_1} = [1,0]; \text{Alloc}_{T_1} = [0,1] \]
\[ \text{Request}_{T_2} = [0,0]; \text{Alloc}_{T_2} = [1,0] \]
\[ \text{Request}_{T_3} = [0,1]; \text{Alloc}_{T_3} = [1,0] \]
\[ \text{Request}_{T_4} = [0,0]; \text{Alloc}_{T_4} = [0,1] \]
\[ \text{Avail} = [1,1] \]
UNFINISHED = \{T1,T3\}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_{T_4}] \leq \text{Avail}) {
      remove node from UNFINISHED
      \[ \text{Avail} = \text{Avail} + \text{Alloc}_{T_4} \]
      done = false
    }
  }
} until(done)
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{Request}_{T_1} &= [1,0]; \text{Alloc}_{T_1} = [0,1] \\
\text{Request}_{T_2} &= [0,0]; \text{Alloc}_{T_2} = [1,0] \\
\text{Request}_{T_3} &= [0,1]; \text{Alloc}_{T_3} = [1,0] \\
\text{Request}_{T_4} &= [0,0]; \text{Alloc}_{T_4} = [0,1] \\
\text{Avail} &= [1,1] \\
\text{UNFINISHED} &= \{T_1,T_3\}
\end{align*}
\]

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_{T_4}] \leq [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_{T_4}]
      done = false
    }
  }
} until(done)
Deadlock Detection Algorithm Example

\[
\begin{align*}
[\text{Request}_{T_1}] &= [1,0]; \text{Alloc}_{T_1} = [0,1] \\
[\text{Request}_{T_2}] &= [0,0]; \text{Alloc}_{T_2} = [1,0] \\
[\text{Request}_{T_3}] &= [0,1]; \text{Alloc}_{T_3} = [1,0] \\
[\text{Request}_{T_4}] &= [0,0]; \text{Alloc}_{T_4} = [0,1] \\
[\text{Avail}] &= [1,1] \\
\text{UNFINISHED} &= \{T_1, T_3\}
\end{align*}
\]

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_{T_4}] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [\text{Alloc}_{T_4}]
            done = false
        }
    }
} until (done)
Deadlock Detection Algorithm Example

[Request_{T_1}] = [1,0]; \text{Alloc}_{T_1} = [0,1]
[Request_{T_2}] = [0,0]; \text{Alloc}_{T_2} = [1,0]
[Request_{T_3}] = [0,1]; \text{Alloc}_{T_3} = [1,0]
[Request_{T_4}] = [0,0]; \text{Alloc}_{T_4} = [0,1]
[\text{Avail}] = [1,1]
\text{UNFINISHED} = \{T_1,T_3\}

do {
    \text{done} = \text{true}
    \text{Foreach node in UNFINISHED} {
        \text{if} ([\text{Request}_{\text{node}}] \leq [\text{Avail}]) {
            \text{remove node from UNFINISHED}
            \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{\text{node}]}
            \text{done} = \text{false}
        }
    }
} until(\text{done})
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{[Request}_{T_1}\text{]} &= [1,0]; \text{ Alloc}_{T_1} = [0,1] \\
\text{[Request}_{T_2}\text{]} &= [0,0]; \text{ Alloc}_{T_2} = [1,0] \\
\text{[Request}_{T_3}\text{]} &= [0,1]; \text{ Alloc}_{T_3} = [1,0] \\
\text{[Request}_{T_4}\text{]} &= [0,0]; \text{ Alloc}_{T_4} = [0,1] \\
\text{[Avail]} &= [1,1] \\
\text{UNFINISHED} &= \{T_1,T_3\}
\end{align*}
\]

\[
\text{do} \{ \\
\quad \text{done} = \text{true} \\
\quad \text{Foreach node in UNFINISHED} \{ \\
\quad\quad \textbf{if} ([\text{Request}_{T_1}] \leq [\text{Avail}]) \{ \\
\quad\quad\quad \text{remove node from UNFINISHED} \\
\quad\quad\quad [\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{T_1}] \\
\quad\quad\quad \text{done} = \text{false} \\
\quad\quad \} \\
\quad \} \text{ until(done)}
\]
Deadlock Detection Algorithm Example

[Request\textsubscript{T1}] = [1,0];  Alloc\textsubscript{T1} = [0,1]
[Request\textsubscript{T2}] = [0,0];  Alloc\textsubscript{T2} = [1,0]
[Request\textsubscript{T3}] = [0,1];  Alloc\textsubscript{T3} = [1,0]
[Request\textsubscript{T4}] = [0,0];  Alloc\textsubscript{T4} = [0,1]
[Avail] = [1,1]
UNFINISHED = \{T3\}

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request\textsubscript{T1}] \leq [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc\textsubscript{T1}]
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

\[ \text{[Request}_{T_1}\] = [1,0]; \text{Alloc}_{T_1} = [0,1] \]
\[ \text{[Request}_{T_2}\] = [0,0]; \text{Alloc}_{T_2} = [1,0] \]
\[ \text{[Request}_{T_3}\] = [0,1]; \text{Alloc}_{T_3} = [1,0] \]
\[ \text{[Request}_{T_4}\] = [0,0]; \text{Alloc}_{T_4} = [0,1] \]
\[ \text{[Avail]} = [1,2] \]
\[ \text{UNFINISHED} = \{T_3\} \]

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request}_{T_1}\] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc}_{T_1}\]
            done = false
        }
    }
} until (done)
Deadlock Detection Algorithm Example

[Request_{T1}] = [1,0]; Alloc_{T1} = [0,1]
[Request_{T2}] = [0,0]; Alloc_{T2} = [1,0]
[Request_{T3}] = [0,1]; Alloc_{T3} = [1,0]
[Request_{T4}] = [0,0]; Alloc_{T4} = [0,1]
[Avail] = [1,2]
UNFINISHED = \{T3\}

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_{T1}] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_{T1}]
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

[Request_{T_1}] = [1,0]; Alloc_{T_1} = [0,1]
[Request_{T_2}] = [0,0]; Alloc_{T_2} = [1,0]
[Request_{T_3}] = [0,1]; Alloc_{T_3} = [1,0]
[Request_{T_4}] = [0,0]; Alloc_{T_4} = [0,1]

[Avail] = [1,2]
UNFINISHED = {T_3}

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_{node}] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_{node}]
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

\[ \text{[Request}_{T_1}] = [1, 0]; \text{Alloc}_{T_1} = [0, 1] \]
\[ \text{[Request}_{T_2}] = [0, 0]; \text{Alloc}_{T_2} = [1, 0] \]
\[ \text{[Request}_{T_3}] = [0, 1]; \text{Alloc}_{T_3} = [1, 0] \]
\[ \text{[Request}_{T_4}] = [0, 0]; \text{Alloc}_{T_4} = [0, 1] \]
\[ \text{[Avail]} = [1, 2] \]
\[ \text{UNFINISHED} = \{T_3\} \]

\[
\text{do } \{
\text{done} = \text{true}
\text{Foreach node in \text{UNFINISHED} } \{
\text{if } ([\text{Request}_{T_3}] \leq \text{[Avail]}) \{
\text{remove node from \text{UNFINISHED}}
\text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{T_3]} \text{done} = \text{false}
\}
\}
\} \text{ until(done) }
\]
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{Request}_{T_1} &= [1,0]; \quad \text{Alloc}_{T_1} = [0,1] \\
\text{Request}_{T_2} &= [0,0]; \quad \text{Alloc}_{T_2} = [1,0] \\
\text{Request}_{T_3} &= [0,1]; \quad \text{Alloc}_{T_3} = [1,0] \\
\text{Request}_{T_4} &= [0,0]; \quad \text{Alloc}_{T_4} = [0,1] \\
\text{Avail} &= [1,2] \\
\text{UNFINISHED} &= \{\}
\end{align*}
\]

do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_{T_3}] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_{T_3}]
            done = false
        }
    }
} until(done)
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{[Request}_{T_1}] &= [1,0]; \quad \text{Alloc}_{T_1} = [0,1] \\
\text{[Request}_{T_2}] &= [0,0]; \quad \text{Alloc}_{T_2} = [1,0] \\
\text{[Request}_{T_3}] &= [0,1]; \quad \text{Alloc}_{T_3} = [1,0] \\
\text{[Request}_{T_4}] &= [0,0]; \quad \text{Alloc}_{T_4} = [0,1] \\
\text{[Avail]} &= [2, 2] \\
\text{UNFINISHED} &= \{ \}
\end{align*}
\]

\[\begin{array}{l}
do \\
\text{done} = \text{true} \\
\text{Foreach node in UNFINISHED} \\
\quad \text{if } ([\text{Request}_{T_3}] \leq [\text{Avail}]) \\
\quad \quad \text{remove node from UNFINISHED} \\
\quad \quad [\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{T_3}] \\
\quad \text{done} = \text{false} \\
\end{array}\]

} until(done)
Deadlock Detection Algorithm Example

\[ \text{[Request}_{T_1} \] = [1,0]; \text{Alloc}_{T_1} = [0,1] \]
\[ \text{[Request}_{T_2} \] = [0,0]; \text{Alloc}_{T_2} = [1,0] \]
\[ \text{[Request}_{T_3} \] = [0,1]; \text{Alloc}_{T_3} = [1,0] \]
\[ \text{[Request}_{T_4} \] = [0,0]; \text{Alloc}_{T_4} = [0,1] \]
\[ \text{[Avail]} = [2,2] \]
\[ \text{UNFINISHED} = {} \]

\[
\text{do} \{
\text{done} = \text{true}
\text{Foreach node in UNFINISHED} \{
\text{if ([Request}_{T_3} \] <= [Avail]) \{
\text{remove node from UNFINISHED}
[\text{Avail]} = [\text{Avail}] + [\text{Alloc}_{T_2}]
\text{done} = \text{false}
\}
\}
\} \text{until(done)}
\]
Deadlock Detection Algorithm Example

\[
\begin{align*}
\text{[Request}_{T_1}\text{]} & = [1,0]; \quad \text{Alloc}_{T_1} = [0,1] \\
\text{[Request}_{T_2}\text{]} & = [0,0]; \quad \text{Alloc}_{T_2} = [1,0] \\
\text{[Request}_{T_3}\text{]} & = [0,1]; \quad \text{Alloc}_{T_3} = [1,0] \\
\text{[Request}_{T_4}\text{]} & = [0,0]; \quad \text{Alloc}_{T_4} = [0,1] \\
\text{[Avail]} & = [2,2] \\
\text{UNFINISHED} & = {}
\end{align*}
\]

\[
\text{do \{} \\
\quad \text{done} = \text{true} \\
\quad \text{\underline{Foreach node in UNFINISHED \{} \} if ([Request}_{T_3}\text{] \leq [Avail])] \} \\
\quad \text{remove node from UNFINISHED} \\
\quad \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{T_3}\text{]} \\
\quad \text{done} = \text{false} \\
\quad \} \\
\} \text{ until(done)}
\]
Banker’s Algorithm for Preventing Deadlock

• Toward right idea:
  – State maximum resource needs in advance
  – Allow particular thread to proceed if:
    \[(\text{available resources} - \#\text{requested}) \geq \text{max remaining that might be needed by any thread}\]

• Banker’s algorithm (less conservative):
  – Allocate resources dynamically
    • Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    • Keeps system in a “SAFE” state, i.e. there exists a sequence \(\{T_1, T_2, \ldots, T_n\}\) with \(T_1\) requesting all remaining resources, finishing, then \(T_2\) requesting all remaining resources, etc..
  – Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources
Banker’s Algorithm

• Technique: pretend each request is granted, then run deadlock detection algorithm, substitute
  \([\text{Request}_{\text{node}}] \leq [\text{Avail}]\) \rightarrow ([\text{Max}_{\text{node}}]-[\text{Alloc}_{\text{node}}] \leq [\text{Avail}])

\[
\begin{align*}
[\text{FreeResources}]: & \quad \text{Current free resources each type} \\
[\text{Alloc}_X]: & \quad \text{Current resources held by thread } X \\
[\text{Max}_X]: & \quad \text{Max resources requested by thread } X
\end{align*}
\]

\begin{align*}
[\text{Avail}] &= [\text{FreeResources}] \\
\text{Add all nodes to UNFINISHED} \\
\text{do } & \{ \\
\quad \text{done} = \text{true} \\
\quad \text{Foreach node in UNFINISHED } & \{ \\
\quad\quad \text{if } ([\text{Max}_{\text{node}}] - [\text{Alloc}_{\text{node}}] \leq [\text{Avail}]) & \{ \\
\quad\quad\quad \text{remove node from UNFINISHED} \\
\quad\quad\quad [\text{Avail}] &= [\text{Avail}] + [\text{Alloc}_{\text{node}}] \\
\quad\quad\quad \text{done} = \text{false} \\
\quad\quad & \} \\
\quad & \} \\
\text{until}(\text{done})
\end{align*}
Banker’s Algorithm Example

- Banker’s algorithm with dining philosophers
  - “Safe” (won’t cause deadlock) if when try to grab chopstick either:
    - Not last chopstick
    - Is last chopstick but someone will have two afterwards
  - What if k-handed philosophers? Don’t allow if:
    - It’s the last one, no one would have k
    - It’s 2\textsuperscript{nd} to last, and no one would have k-1
    - It’s 3\textsuperscript{rd} to last, and no one would have k-2
    - …
Summary: Deadlock

• Four conditions for deadlocks
  – Mutual exclusion
    • Only one thread at a time can use a resource
  – Hold and wait
    • Thread holding at least one resource is waiting to acquire additional resources held by other threads
  – No preemption
    • Resources are released only voluntarily by the threads
  – Circular wait
    • ∃ set \{T_1, \ldots, T_n\} of threads with a cyclic waiting pattern

• Starvation vs. Deadlock
  – Starvation: thread waits indefinitely
  – Deadlock: circular waiting for resources

• Deadlock detection and preemption

• Deadlock prevention
  – Loop Detection, Banker’s algorithm