Deadlock

Chapter 32 in “Three Easy Steps”
Chapter 19 in Harmony Book

CS 4410
Operating Systems

The slides are the product of many rounds of teaching CS 4410 by Professors Agarwal, Alvisi, Bracy, George, Schneider, Sirer, Van Renesse.
Dining Philosophers [Dijkstra 68]

Pi: do forever
    acquire( left(i) );
    acquire( right(i) );
    eat
    release( left(i) );
    release( right(i) );
    think
end

left(i): i
right(i): i+1 mod 5
Dining Philosophers in Harmony

```python
from sync import Lock, acquire, release

const N = 5

forks = [Lock(),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            acquire(?forks[left])
            acquire(?forks[right])
            # dine
            release(?forks[left])
            release(?forks[right])
            # think

for i in {0..N-1}:
    spawn diner(i)
```
Dining Philosophers in Harmony

```
from synch import Lock, acquire, release

const N = 5
forks = [Lock(),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
    while choose({False, True}):
        acquire(?forks[left])
        acquire(?forks[right])
        # dine
        release(?forks[left])
        release(?forks[right])
        # think

for i in {0..N-1}:
    spawn diner(i)
```

Summary: some execution cannot terminate

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: init()
  - Line 5: Initialize forks to [ False, False, False ]
  - Thread terminated
- Schedule thread T2: diner(1)
  - Line 9: Choose True
  - Line synch/36: Set forks[1] to True (was False)
  - Preempted in diner(1) --> acquire(?forks[2]) about to execute atomic section in line synch/35
- Schedule thread T3: diner(2)
  - Line 9: Choose True
  - Line synch/36: Set forks[2] to True (was False)
  - Preempted in diner(2) --> acquire(?forks[0]) about to execute atomic section in line synch/35
- Schedule thread T1: diner(0)
  - Line 9: Choose True
  - Line synch/36: Set forks[0] to True (was False)
  - Preempted in diner(0) --> acquire(?forks[1]) about to execute atomic section in line synch/35

Final state (all threads have terminated or are blocked):

- Threads:
  - T1: (blocked) diner(0) --> acquire(?forks[1])
    - about to execute atomic section in line synch/35
  - T2: (blocked) diner(1) --> acquire(?forks[2])
    - about to execute atomic section in line synch/35
  - T3: (blocked) diner(2) --> acquire(?forks[0])
    - about to execute atomic section in line synch/35
- Variables:
  - forks: [ True, True, True ]

harmony –cN=3
Problematic Emergent Properties

**Starvation:** Process waits forever

**Deadlock:** A set of processes exists, where each is blocked and can become unblocked only by actions of another process in the set.

- Deadlock implies Starvation (but not *vice versa*

- Starvation often tied to **fairness:** A process is not forever blocked awaiting a condition that (i) becomes continuously true or (ii) infinitely-often becomes true.

*Testing for starvation or deadlock is difficult in practice*
More Examples of Deadlock

Example (initially \(\text{in1} = \text{in2} = \text{False}\)):

\[
\begin{align*}
\text{in1} &= \text{True}; \quad \textbf{await not} \; \text{in2}; \quad \text{in1} = \text{False} \\
// \\
\text{in2} &:= \text{True}; \quad \textbf{await not} \; \text{in1}; \quad \text{in2} = \text{False}
\end{align*}
\]

Example (initially \(\text{lk1} = \text{lk2} = \text{released}\)):

\[
\begin{align*}
\text{acquire(lk1)}; \quad \text{acquire(lk2)}; \quad \text{release(lk2)}; \quad \text{release(lk1)}; \\
// \\
\text{acquire(lk2)}; \quad \text{acquire(lk1)}; \quad \text{release(lk1)}; \quad \text{release(lk2)};
\end{align*}
\]
System Model

• Set of resources requiring “exclusive” access
  • Might be “k-exclusive access” if resource has capacity for k
  • Examples: buffers, packets, I/O devices, processors, …

• Protocol to access a resource causes blocking:
  • If resource is free, then access is granted; process proceeds
  • If resource is in use, then process blocks
    – Use resource
    – Release resource

When is deadlock possible?
1. **Mutual Exclusion.** Acquire can block invoker

2. **Hold & wait.** A process can be blocked while holding resources

3. **No preemption.** Allocated resources cannot be reclaimed. Explicit release operation needed

4. **Circular waits** are possible
   
   *Let* $p \rightarrow q$ *denote “p waits for q to release a resource”. Then* $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \rightarrow P_1$
Deadlock is Undesirable

- **Deadlock prevention**: Ensure that a necessary condition cannot hold
- **Deadlock avoidance**: System does not allocate resources that will lead to a deadlock
- **Deadlock detection**: Allow system to deadlock; detect it; recover
Deadlock Prevention: Negate 1

#1: Eliminate mutual exclusion / bounded resources:

- Make resources sharable without locks
  - E.g., time-shared CPU
  - Harmony book has examples of non-blocking concurrent data structures
- Have sufficient resources available, so acquire never delays
  - E.g., make sure bounded queue is “large enough”
#2: Eliminate hold and wait

Don’t hold some resources when requesting others

- Re-write code:

```plaintext
acquire(?foo_lock);
foo1();
acquire(?bar_lock);
bar();
release(?bar_lock);
foo2();
release(?foo_lock);
```

```plaintext
acquire(?foo_lock);
foo1();
release(?foo_lock);
acquire(?bar_lock);
bar();
release(?bar_lock);
acquire(?foo_lock);
foo2();
release(?foo_lock);
```

- **Assuming bar() does not access shared variables protected by foo_lock, are these the same?**
#2: Eliminate hold and wait
Don’t hold some resources when requesting others

• Re-write code:

```
acquire(?foo_lock);
foo1();
acquire(?bar_lock);
bar();
release(?bar_lock);
foo2();
release(?foo_lock);
```

```
acquire(?foo_lock);
foo1();
release(?foo_lock);
acquire(?bar_lock);
bar();
release(?bar_lock);
acquire(?foo_lock);
foo2();
release(?foo_lock);
```

• Answer: no. The state that foo_lock protects may change between foo1() and foo2() in code on the right
Deadlock Prevention: Negate 2

#2: Eliminate hold and wait

Don’t hold some resources when requesting others
  • Re-write code

  • Another approach: request all resources at once
    – Problems:
      – Processes don’t know what they need ahead of time
      – No mechanism to request all resources at the same time
      – Starvation (if waiting on many popular resources)
      – Low utilization (need resource only for a bit)
Simultaneous Acquisition in Harmony

```python
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(),] * N

def diner(which):
    left, right = (which, (which + 1) % N):
    while choose({False, True}):
        synch.acquire(mutex)
        while forks[left] or forks[right]:
            if forks[left]:
                synch.wait(conds[left], mutex)
            if forks[right]:
                synch.wait(conds[right], mutex)
        assert not (forks[left] or forks[right])
        forks[left] = forks[right] = True
        synch.release(mutex)
    # dine
    synch.acquire(mutex)
    forks[left] = forks[right] = False
    synch.notify(conds[left])
    synch.notify(conds[right])
    synch.release(mutex)
    # think
```

wait for both forks and then grab them both

release both forks
Simultaneous Acquisition in Harmony

```python
def diner(which):
    left, right = (which, (which + 1) % N):
    while choose({False, True}):
        with synch.acquire(mutex):
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(conds[left], mutex)
                if forks[right]:
                    synch.wait(conds[right], mutex)
    assert not (forks[left] or forks[right])
    forks[left] = forks[right] = True
    synch.release(mutex)
    # dine
    synch.acquire(mutex)
    forks[left] = forks[right] = False
    synch.notify(conds[left])
    synch.notify(conds[right])
    synch.release(mutex)
    # think
```
Simultaneous Acquisition in Harmony

```python
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(),] * N

def diner(which):
    left, right = (which, (which + 1) % N):
    while choose({False, True}):
        synch.acquire(mutex)
        while forks[left]:
            synch.wait(conds[left], mutex)
        while forks[right]:
            synch.wait(conds[right], mutex)
        assert not (forks[left] or forks[right])
        forks[left] = forks[right] = True
        synch.release(mutex)
# dine
synch.acquire(mutex)
forks[left] = forks[right] = False
synch.notify(conds[left])
synch.notify(conds[right])
synch.release(mutex)
# think
```
Simultaneous Acquisition in Harmony

```python
def diner(which):
    left, right = (which, (which + 1) % N):
    while choose({False, True}):
        synch.acquire(mutex)
        while forks[left]:
            synch.wait(conds[left], mutex)
        while forks[right]:
            synch.wait(conds[right], mutex)
        assert not (forks[left] or forks[right])
        forks[left] = forks[right] = True
        synch.release(mutex)
    # think
    synch.acquire(mutex)
    forks[left] = forks[right] = False
    synch.notify(conds[left])
    synch.notify(conds[right])
    synch.release(mutex)
```

Wait for left fork, then wait for right fork. Wouldn’t this be just as good?

NO!

(run through harmony if you don’t believe me)
#3: Allow preemption

Requires mechanism to save / restore resource state:

- multiplexing vs undo/redo

- Examples of multiplexing:
  - processor registers (contexts)
  - regions of memory (pages)

- Examples of undo/redo
  - database transaction processing
#4: Eliminate circular waits.

Let $R = \{R_1, R_2, \ldots, R_n\}$ be the set of resource types.
Let $(R, <)$ be a non-symmetric relation:

- for every $r, s, t$:
  - $\neg (r < r)$ [irreflexive]
  - $(r < s \land s < t) \Rightarrow r < t$ [transitive]
  - $\neg (r < s \land s < r)$ [non-symmetric]
  - $r \neq s \Rightarrow (r < s \lor s < r)$ [total order]

**Rule:** Request resources in increasing order by $<$
(All resources from type $R_i$ must be requested together)

**Rule:** To request resources of type $R_i$, first release all resources from type $R_j$ where $R_i < R_j$. 
Why < Rule Works

**Thm:** Total order resource allocation avoids circular waits

Proof: By contradiction. Assume a circular wait exists

\[ P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow \ldots \rightarrow P_n \rightarrow P_1. \]

P1 requesting R1 held by P2.
P2 requesting R2 held by P3. (So $R_1 < R_2$ holds)

\[ \ldots \]

Conclude: $R_1 < R_2, \ R_2 < R_3, \ldots, \ R_n < R_1$

By transitivity: $R_1 < R_1$. Violates irreflexivity.

A contradiction!
Dining Philosophers (Again)

Pi: do forever
   acquire( F(i) );
   acquire( G(i) );
   eat
   release( F(i) );
   release( G(i) );
end

F(i): \text{min}(i, i+1 \mod 5)
G(i): \text{max}(i, i+1 \mod 5)
Ordering Resources in Harmony

```python
if left < right:
    sync.acquire(?forks[left])
    sync.acquire(?forks[right])
else:
    sync.acquire(?forks[right])
    sync.acquire(?forks[left])
```

or

```python
sync.acquire(?forks[min(left, right)])
sync.acquire(?forks[max(left, right)])
```
Havender’s Scheme (OS/360)
Hierarchical Resource Allocation
Every resource is associated with a level.

- **Rule H1**: All resources from a given level must be acquired using a single request.
- **Rule H2**: After acquiring from level \( L_j \) must not acquire from \( L_i \) where \( i < j \)
- **Rule H3**: May not acquire from \( L_i \) unless already released from \( L_j \) where \( j > i \).

Example of allowed sequence:
1. acquire(\( W@L_1, X@L_1 \))
2. acquire(\( Y@L_3 \))
3. release(\( Y@L_3 \))
4. acquire(\( Z@L_2 \))
Deadlock Detection

Create a **Wait-For Graph**

- 1 Node per Process
- 1 Outgoing Edge per Waiting Process, P (from P to the process it’s waiting for)

Note: graph holds for a single instant in time

**Cycle** in graph indicates deadlock
Testing for cycles (= deadlock)

Reduction Algorithm:
Find a node with no outgoing edges
• Erase node
• Erase any edges coming into it
• Repeat until no such node

Intuition: Deleted node is for process that is not waiting. It will eventually finish and release its resources, so any process waiting for those resources will longer be waiting.

Erase whole graph ⇔ graph has no cycles
Graph remains ⇔ deadlock
Graph can be fully reduced, hence there was no deadlock at the time the graph was drawn.
(Obviously, things could change later!)
No node with no outgoing edges…
Irreducible graph, contains a cycle (only some processes are in the cycle)

→ deadlock
Question: Does choice of node for reduction matter?

Answer: No.

Explanation: an unchosen candidate at one step remains a candidate for later steps. Eventually—regardless of order—every node will be reduced (if there’s no deadlock).
Question:
Suppose no deadlock detected at time T. Can we infer about a later time T+x?

Answer: Nothing.
Explanation: The very next step could be to run some process that will request a resource...
... establishing a cyclic wait
... and causing deadlock
Implementing Deadlock Detection

• Track resource allocation (who has what)
• Track pending requests (who’s waiting for what)

Maintain a wait-for graph.

When to run graph reduction?
• Whenever a request is blocked?
• Periodically?
• Once CPU utilization drops below a threshold?
Deadlock Recovery

Blue screen & reboot?

Kill one/all deadlocked processes
• Pick a victim
• Terminate
• Repeat if needed

Preempt resource/processes till deadlock broken
• Pick a victim (# resources held, execution time)
• Rollback (partial or total, not always possible)
Deadlock in traffic
Deadlock Avoidance

How do cars do it?
- Try not to block an intersection
- Don’t drive into the intersection if you see that you might get stuck there

Why does this work?
- Prevents a wait-for relationship
- Cars won’t take up a resource if they see they won’t be able to acquire the next one…
Deadlock Avoidance

*state*: allocation to each process

*safestate*: a state from which some execution is possible that does not cause deadlock

- Requires knowing max allocation for each process and who holds what resources
- Check that
  - Exists sequence P1 P2 … Pn of processes where:
    - Forall i where 1 ≤ i ≤ n:
      - Pi can be satisfied by Avail + resources held by P1 … Pi-1.

Assumes no synchronization between processes, except for resource requests
Safe State Example

Suppose: 12 tape drives and 3 processes: \( p_0, p_1, \) and \( p_2 \)

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could still ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_0 )</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>( p_1 )</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

3 drives remain

Is this a safe state (i.e., is there a sequence of granting requests that will work without deadlock)?
Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

3 drives remain

Current state is safe because a safe sequence exists: [p1, p0, p2]
- p1 can complete with remaining resources
- p0 can complete with remaining+p1
- p2 can complete with remaining+p1+p0

What if p2 requests 1 drive? Grant or not?
Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could still ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

2 drives remain

Is this state safe? (Is there a sequence of requests that works?)
Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could still ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

4 drives remain

Is this state safe? (Is there a sequence of requests that works?)
Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

<table>
<thead>
<tr>
<th></th>
<th>max need</th>
<th>current usage</th>
<th>could still ask for</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

4 drives remain

Is this state safe? (Is there a sequence of requests that works?)

(potentially) STUCK...

(non-terminating state)
Suppose: 12 tape drives and 3 processes: p0, p1, and p2

Current state is safe because a safe sequence exists: [p1, p0, p2]
- p1 can complete with remaining resources
- p0 can complete with remaining+p1
- p2 can complete with remaining+p1+p0

What if p2 requests 1 drive? Grant or not?
Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

Current state is safe because a safe sequence exists: [p1, p0, p2]
- p1 can complete with remaining resources
- p0 can complete with remaining+p1
- p2 can complete with remaining+p1+p0

What if p2 requests 1 drive? Grant or not? NO (block or deny)
Banker’s Algorithm

- from 10,000 feet:
  - Process declares its worst-case needs, asks for what it “really” needs, a little at a time
  - Algorithm decides when to grant requests
    - Build a graph assuming request granted
    - Reducible? yes: grant request, no: wait

Problems:
- Fixed number of processes
- Need worst-case needs ahead of time
- Expensive
  - not used much in practice