Deadlock

Chapter 32 in “Three Easy Steps”
Chapter 19 in the Harmony Book
Dining Philosophers

\[ P_i: \text{ do forever} \]
\[ \quad \text{acquire}(\text{left}(i)) ; \]
\[ \quad \text{acquire}(\text{right}(i)) ; \]
\[ \quad \text{eat} ; \]
\[ \quad \text{release}(\text{left}(i)) ; \]
\[ \quad \text{release}(\text{right}(i)) ; \]
\[ \text{end} \]

left(i): i

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

```python
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)
```
Dining Philosophers in Harmony

<table>
<thead>
<tr>
<th>Turn</th>
<th>Thread</th>
<th>Instructions Executed</th>
<th>PC</th>
<th>Shared Variables</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T0: <strong>init</strong>()</td>
<td></td>
<td>1122</td>
<td>False False False False False</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T4: diner(3)</td>
<td></td>
<td>797</td>
<td>False False False True False</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T1: diner(0)</td>
<td></td>
<td>797</td>
<td>True False False True False</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T2: diner(1)</td>
<td></td>
<td>797</td>
<td>True True False True False</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T3: diner(2)</td>
<td></td>
<td>797</td>
<td>True True True True True</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>T5: diner(4)</td>
<td></td>
<td>797</td>
<td>True True True True True</td>
<td></td>
</tr>
</tbody>
</table>

/Users/rvr/github/harmony/harmony/harmony/harmony_model_checker/modules/synch.hny:31 atomically when not !binsem:

<table>
<thead>
<tr>
<th>ID</th>
<th>Status</th>
<th>Stack Trace</th>
<th>Stack Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>terminated</td>
<td><strong>init</strong>()</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>blocked</td>
<td>diner(0)</td>
<td>left: 0, result: None, right: 1</td>
</tr>
<tr>
<td>T2</td>
<td>blocked</td>
<td>diner(1)</td>
<td>left: 1, result: None, right: 2</td>
</tr>
<tr>
<td>T3</td>
<td>blocked</td>
<td>diner(2)</td>
<td>left: 2, result: None, right: 3</td>
</tr>
<tr>
<td>T4</td>
<td>blocked</td>
<td>diner(3)</td>
<td>left: 3, result: None, right: 4</td>
</tr>
<tr>
<td>T5</td>
<td>blocked</td>
<td>diner(4)</td>
<td>left: 4, result: None, right: 0</td>
</tr>
</tbody>
</table>

756 Load
757 LoadVar old
758 DelVar old
759 2-ary ==
760 StoreVar result
761 LoadVar result
762 JumpCond False 768
763 LoadVar p
764 DelVar p
Problematic Emergent Properties

- **Starvation**: Process waits forever

- **Deadlock**: a set of processes exist, where each is blocked and can become unblocked only by the action of another process in the same set
  - Deadlock implies Starvation (but not vice versa)
  - Starvation often tied to fairness — which requires that a process be not forever blocked on a condition that becomes (i) continuously true or (ii) infinitely-often true

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

Example 1 (initially in1 = in2 = False):

```plaintext
in1 = True;  await not in2;  in1 = False

//
in2 = True;  await not in1;  in2 = False
```

Example 2 (initially lk1 = lk2 = released):

```plaintext
acquire(lk1); acquire(lk2);  release(lk2); release(lk1)

//
acquire(lk2); acquire(lk1);  release(lk1); release(lk2)
```
System Model

- Set of resources requiring “exclusive” access
  - Might be “k exclusive access” if k instances of resource are available
  - Examples: buffers, packets, I/O devices, processors

- Protocol to access a resource causes blocking
  - If resource is free, access is granted and process proceeds
    - Uses resource
    - Releases resource
  - If resource is in use, process blocks
A Graph Theoretic Model of Deadlock

Computer system modeled as a RAG, a directed graph $G(V, E)$

- $V = \{P_1, \ldots, P_n\} \cup \{R_1, \ldots, R_n\}$

- $E = \{\text{edges from a resource to a process}\} \cup \{\text{edges from a process to a resource}\}$
Necessary conditions for deadlock

Deadlock only if they all hold

1. **Bounded resources**
   - Acquire can block invoker

2. **No preemption**
   - the resource is mine, MINE! (until I release it)

3. **Wait while holding**
   - holds one resource while waiting for another

4. **Circular waiting**
   - \( P_i \) waits for \( P_{i+1} \) and holds a resource requested by \( P_{i-1} \)

Not sufficient in general

---

---

---
Deadlock is Undesirable!

- Deadlock prevention: Ensure that a necessary condition cannot hold
- Deadlock avoidance: System does not allocate resources that may lead to a deadlock
- Deadlock detection: Allow system to deadlock; detect it; recover
Testing for cycles

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

Intuition: Node with no outgoing edges is not waiting on any resource
- It will eventually finish and release its resources
- Processes waiting for those resources will be able to acquire them and will no longer be waiting!

Erase all edges ⇐⇒ Graph has no cycles
Edges remain ⇐⇒ Deadlock
RAG Reduction

Deadlock?
NO! (no cycles)

Step 1: Satisfy P₃’s requests
Step 2: Satisfy P₂’s requests
Step 3: Satisfy P₁’s requests

Schedule [P₃ P₂ P₁] completely eliminates edges!
RAG Reduction

Deadlock?

NO! (no cycles)

Step 1: Satisfy P₃'s requests
Step 2: Satisfy P₂'s requests
Step 3: Satisfy P₁'s requests
Schedule [P₃ P₂ P₁] completely eliminates edges!

Deadlock?

Yes!

RAG has a cycle
Every node has some outgoing edge
Cannot satisfy any of P₁, P₂, P₃ requests!
Step 1: Satisfy P3's requests
Step 2: Satisfy P2's requests
Step 3: Satisfy P1's requests
Schedule [P3 P2 P1] completely eliminates edges!

Deadlock?
NO! (no cycles)

Deadlock?
Yes!
RAG has a cycle
Every node has some outgoing edge
Cannot satisfy any of P1, P2, P3 requests!

Deadlock?
NO!
RAG has a cycle
Schedule [P2 P1 P3 P4] completely eliminates edges!
More Musings on Deadlock

Does the order of RAG reduction matter?

- No. If $P_i$ and $P_j$ can both be reduced, reducing $P_i$ does not affect the reducibility of $P_j$

Does a deadlock disappear on its own?

- No. Unless a process is killed or forced to release a resource, we are stuck!

If a system is not deadlocked at time $T$, is it guaranteed to be deadlock-free at $T+1$?

- No. Just by requesting a resource (never mind being granted one) a process can create a circular wait!
Deadlock Prevention: Negate

Eliminate “Acquire can block invoker/bounded resources”

- Make resources sharable without locks
  - Wait-free synchronization
  - The Harmony book (Chapter 23) has examples of non-blocking data structures
- Have sufficient resources available, so acquire never delays (duh!)
  - E.g., use an unbounded queue, or make sure that queue is “large enough”
Deadlock Prevention: Negate

Allow preemption

- Requires mechanisms to save/restore resource state
  - multiplexing (registers, memory, etc).
  - undo/redo (database transaction processing)
- Allow OS to preempt resources of waiting processes
- Allow OS to preempt resources of requesting processes