Deadlock
(Chapter 32)

CS 4410
Operating Systems

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

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

left(i): i+1 mod 5
right(i): i
import synch;

const N = 5;

def diner(which):
    let left, right = (which, (which % N) + 1):
        while choose({ False, True }):
            lock(?forks[left]);
            lock(?forks[right]);
            # dine
            unlock(?forks[left]);
            unlock(?forks[right]);
            # think
            ;
    ;
    forks = dict{ Lock() for i in {1..N} };
    for i in {1..N}:
        spawn diner(i);
## Issue: Non-terminating State

<table>
<thead>
<tr>
<th>Process</th>
</tr>
</thead>
</table>
| init     | /
| diner/1  | 822-834 835 (choose True) 836-841 587
| diner/2  | 822-834 835 (choose True) 836-841 587
| diner/3  | 822-834 835 (choose True) 836-841 587
| diner/4  | 822-834 835 (choose True) 836-841 587
| diner/5  | 822-834 835 (choose True) 836-841 587

<table>
<thead>
<tr>
<th>Shared Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>forks</td>
</tr>
<tr>
<td>dict{ 1:False, 2:False, 3:False, 4:False, 5:False }</td>
</tr>
<tr>
<td>dict{ 1:True, 2:False, 3:False, 4:False, 5:False }</td>
</tr>
<tr>
<td>dict{ 1:True, 2:True, 3:False, 4:False, 5:False }</td>
</tr>
<tr>
<td>dict{ 1:True, 2:True, 3:True, 4:False, 5:False }</td>
</tr>
<tr>
<td>dict{ 1:True, 2:True, 3:True, 4:True, 5:False }</td>
</tr>
<tr>
<td>dict{ 1:True, 2:True, 3:True, 4:True, 5:True }</td>
</tr>
</tbody>
</table>
# Dining Philosophers in Harmony

<table>
<thead>
<tr>
<th>Process</th>
<th>Status</th>
<th>Stack Trace</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>diner/1</td>
<td>blocked</td>
<td>848 diner(1): lock(?forks[right]); 593 lock(?forks[2]): await not tas(lk); 574 tas(?forks[2]): atomic:</td>
<td>result = () which = 1 left = 1 right = 2</td>
</tr>
<tr>
<td>diner/2</td>
<td>blocked</td>
<td>848 diner(2): lock(?forks[right]); 593 lock(?forks[3]): await not tas(lk); 574 tas(?forks[3]): atomic:</td>
<td>result = () which = 2 left = 2 right = 3</td>
</tr>
<tr>
<td>diner/3</td>
<td>blocked</td>
<td>848 diner(3): lock(?forks[right]); 593 lock(?forks[4]): await not tas(lk); 574 tas(?forks[4]): atomic:</td>
<td>result = () which = 3 left = 3 right = 4</td>
</tr>
<tr>
<td>diner/4</td>
<td>blocked</td>
<td>848 diner(4): lock(?forks[right]); 593 lock(?forks[5]): await not tas(lk); 574 tas(?forks[5]): atomic:</td>
<td>result = () which = 4 left = 4 right = 5</td>
</tr>
<tr>
<td>diner/5</td>
<td>blocked</td>
<td>848 diner(5): lock(?forks[right]); 593 lock(?forks[1]): await not tas(lk); 574 tas(?forks[1]): atomic:</td>
<td>result = () which = 5 left = 5 right = 1</td>
</tr>
</tbody>
</table>
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 \( in1 = in2 = False \)):

\[
in1 = True; \ \textbf{await not} \ \text{in2}; \ \text{in1} = False; \\
// \\
in2 := True; \ \textbf{await not} \ \text{in1}; \ \text{in2} = False;
\]

Example (initially semaphores \( m1 = m2 = 1 \)):

\[
\text{P}(m1); \ \text{P}(m2); \ \text{V}(m1); \ \text{V}(m2); \\
// \\
\text{P}(m2); \ \text{P}(m1); \ \text{V}(m1); \ \text{V}(m2);
\]
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?
Necessary Conditions for Deadlock
Edward Coffman 1971

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
  – Harmony book Chapter 22 has examples of non-blocking data structures
• Have sufficient resources available, so acquire never delays
  – E.g., unbounded queue, or simply make sure bounded queue is “large enough”
Deadlock Prevention: Negate 2

#2: Eliminate hold and wait
Don’t hold some resources when waiting for others.

• Re-write code:

```python
def foo():
    lock(?mutex);
    doSomeStuff();
    bar();
    doOtherStuff();
    unlock(?mutex);
```

• Or request all resources before execution begins

```python
def foo():
    lock(?mutex);
    doSomeStuff();
    unlock(?mutex);
    bar();
    lock(?mutex);
    doOtherStuff();
    unlock(?mutex);
```

• Problems:
  – Processes don’t know what they need ahead of time.
  – Starvation (if waiting on many popular resources).
  – Low utilization (need resource only for a bit).
#3: Allow preemption

Requires mechanism to save / restore resource state:

- multiplexing vs undo/redo

- Examples of multiplexing:
  - processor registers
  - Regions of memory (pages)

- Examples of undo/redo
  - Database transaction processing
Deadlock Prevention: Negate 1 of 4

#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:
- not $r < r$ [irreflexive]
- if $r < s$ and $s < t$ then $r < t$ [transitive]
- not $r < s$ and $s < r$ [non-symmetric]
- for every $r$ and $s$ ($r \neq s$): $r < s$ or $s < r$ [total order]

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

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

**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 R1 < R2 holds)

\[ \ldots \]

Conclude: R1 < R2, R2 < R3, ..., Rn < R1

By transitivity: R1 < R1. A contradiction!
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($W@L1, X@L1$)
2. acquire($Y@L3$)
3. release($Y@L3$)
4. acquire($Z@L2$)
Dining Philosophers (Again)

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

F(i):  min(i, i+1 mod 5)
G(i):  max(i, i+1 mod 5)
def diner(which):
    let left, right = which, (which % N) + 1:
    if left < right:
        lock(?forks[left]);
        lock(?forks[right]);
    else:
        lock(?forks[right]);
        lock(?forks[left]);

# dine
unlock(?forks[left]);
unlock(?forks[right]);
# think

;
def diner(which):
    left, right = (which, (which % N) + 1):
    while choose({False, True}):
        lock(mutex);
        while forks[left] or forks[right]:
            if forks[left]:
                wait(conds[left]);
            if forks[right]:
                wait(conds[right]);
        assert not (forks[left] or forks[right]);
        forks[left] = True;
        forks[right] = True;
        unlock(mutex);

        # dine
        lock(mutex);
        forks[left] = False;
        forks[right] = False;
        notify(conds[left]);
        notify(conds[right]);
        unlock(mutex);

        # think
        ;
        ;
        ;
def diner(which):
    left, right = (which, (which % N) + 1):
    while choose({False, True}):
        lock(mutex):
            while forks[left] or forks[right]:
                if forks[left]:
                    wait(conds[left]):
                if forks[right]:
                    wait(conds[right]):
            assert not (forks[left] or forks[right]):
            forks[left] = True:
            forks[right] = True:
            unlock(mutex):

            # dine
            lock(mutex):
            forks[left] = False:
            forks[right] = False:
            notify(conds[left]):
            notify(conds[right]):
            unlock(mutex):

            # think
            ;
            ;
            ;
```python
def diner(which):
    let left, right = (which, (which % N) + 1):
    while choose({False, True}):
        lock(?mutex);
        while forks[left] or forks[right]:
            if forks[left]:
                wait(?conds[left]);
            if forks[right]:
                wait(?conds[right]);
            assert not (forks[left] or forks[right]);
        forks[left] = True;
        forks[right] = True;
        unlock(?mutex);

        # dine
        lock(?mutex);
        forks[left] = False;
        forks[right] = False;
        notify(?conds[left]);
        notify(?conds[right]);
        unlock(?mutex);

        # think
        ;
        ;
        ;
    ```
def diner(which):
    let left, right = (which, (which % N) + 1):
    while choose({False, True}):
        lock(?mutex);
        while forks[left] or forks[right]:
            if forks[left]:
                wait(?conds[left]);
            if forks[right]:
                wait(?conds[right]);
        assert not (forks[left] or forks[right]);
        forks[left] = True;
        forks[right] = True;
        unlock(?mutex);

        # dine
        lock(?mutex);
        forks[left] = False;
        forks[right] = False;
        notify(?conds[left]);
        notify(?conds[right]);
        unlock(?mutex);

        # think
        ;
        ;
        ;
Deadlock Detection

Create a **Wait-For Graph**
- 1 Node per Process
- 1 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

**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 $\leftrightarrow$ graph has no cycles
Graph remains $\leftrightarrow$ 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)
$\Rightarrow$ 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.
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 Avoidance

How do cars do it?
- Try not to block an intersection
- Don’t drive into the intersection if you can see that you’ll be 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

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

- Requires knowing max allocation for each process.
- Check that
  - Exists sequence P1 P2 … Pn of processes where:
    For all 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: 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>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?)

STUCK...
(non-terminating state)
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>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

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

<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>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?  NO
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
Avoiding Deadlock in Harmony

```python
def diner(which):
    left, right = (which, (which % N) + 1):
    while choose({False, True}):
        P(semaphore);
        lock(forks[left]);
        lock(forks[right]);
        # dine
        unlock(forks[left]);
        unlock(forks[right]);
        V(semaphore);
        # think
    forks = dict{Lock() for i in {1..N} };
    semaphore = Semaphore(N - 1);
    fork[i := left, N]
```