The Dining-Philosophers problem

- Multiple resources (Dijkstra 1968)
- “Five philosophers sit around a table, which is set with 5 plates (one for each philosopher), 5 chopsticks, and a bowl of rice. Each philosopher alternately thinks and eats. To eat, she needs the two chopsticks next to her plate. When finished eating, she puts the chopsticks back on the table, and continues thinking.”
- Philosophers are *processes*, and chopsticks are *resources.*
Specification😊

Rice

[Diagram of people sitting around a table with rice]
Problems with dining philosophers

• The system may *deadlock*: if all 5 philosophers take up their left chopstick simultaneously, the system will halt (unless one of them puts one back)

• A philosopher may starve if her neighbors have alternating eating patterns
Reader-writers with a monitor

• chopsticks[i]: how many chopsticks are available for philosopher i

• Methods:
  – start_eating(i): if chopsticks[i] < 2, then philosopher i must wait; otherwise decrement the chopstick counts, and eat
  – stop_eating(i): increments the neighboring chopsticks counts
Dining philosophers with a monitor

monitor DiningPhilosophers
int chopsticks[5] = { 2, 2, 2, 2, 2 };
Condition c_available[5];

void start_eating(int i) {
    if(chopsticks[i] != 2)
        c_available[i].wait;
    chopsticks[(i - 1) mod 5]--;
    chopsticks[(i + 1) mod 5]--;
}

void stop_eating(int i) {
    chopsticks[(i - 1) mod 5]++;
    chopsticks[(i + 1) mod 5]++;
    if(chopsticks[(i - 1) mod 5] == 2)
        c_available[(i - 1) mod 5].signal;
    if(chopsticks[(i + 1) mod 5] == 2)
        c_available[(i + 1) mod 5].signal;
}
Dining philosophers with a monitor

- Deadlock is not possible because both chopsticks are taken up at the same time (in the same critical section)
- However, philosophers may still starve (why?)
Remember these problems

- Producer/consumer
- Bounded-buffer
- Readers/writers
- Dining philosophers
The Deadlock Problem
What is deadlock?

- When some system processes are blocked on resource requests that can *never* be satisfied unless *drastic* actions are taken, the processes are *deadlocked*.

- Three approaches to deadlock
  - *Prevent* deadlock by careful system analysis
  - *Detect* deadlock when it happens, and take corrective action
  - Ignore the problem and hope for the best (this is the Unix and Windows model)
Why worry?

- Not all systems need deadlock analysis
  - Some are simple
  - It may not matter (reboot may be an option)
- Some systems are critical
  - The control system in your car; on a plane; high-energy physics
  - Life-support systems (during surgery, say)
  - Online services, where deadlock is expensive
- Each of these systems is managed by an OS
Causes

- Strict deadlock is caused by cyclic resource requests
- *Effective* deadlock is caused by resource depletion (for example, not enough memory to run a large process)
A model of deadlock

- For now, the *state* of an operating system is the allocation status of its resources.
- The system state is changed by processes that issue *request*, *acquire*, and *release* resources (these are the only operations we’ll deal with).
- If a process is not blocked in a system state, it may change the state to a new one.
Definitions I

1. A system is a pair \((\sigma, \pi)\), where \(\sigma\) is a set of system states \(\{S_1, S_2, S_3, \ldots\}\), and \(\pi\) is a set of processes \(\{p_1, p_2, p_3, \ldots\}\).

2. A process \(p_i\) is a function from system states to sets of system states (the process can change the state into several possible states).
   
   If \(p_i\) can change the state from \(S_i\) to \(S_j\), we say \(S_i \xrightarrow{p_i} S_j\).
Definitions II

3. A process is blocked in state $S_i$ if it cannot affect the state in any way.

4. A process is deadlocked if it is blocked, and no matter how the state changes, the process remains blocked.

5. A system state $S_i$ is deadlocked if there is a process $p_j$ deadlocked in that state.

6. A system state $S_i$ is safe if there is no path to a deadlocked state.
Two-resource example

- Processes \( p_1 \) and \( p_2 \) write to a common file \( D \) and require a scratch tape \( T \). Process \( p_1 \) requests \( D \) before \( T \); process \( p_2 \) requests \( T \) before \( D \).

<table>
<thead>
<tr>
<th>States for ( p_1 )</th>
<th>States for ( p_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. holds no resources</td>
<td>0. holds no resources</td>
</tr>
<tr>
<td>1. requested ( D )</td>
<td>1. requested ( T )</td>
</tr>
<tr>
<td>2. holds ( D )</td>
<td>2. holds ( T )</td>
</tr>
<tr>
<td>3. requested ( T ), holds ( D )</td>
<td>3. requested ( D ), holds ( T )</td>
</tr>
<tr>
<td>4. holds ( T ), holds ( D )</td>
<td>4. holds ( D ), holds ( T )</td>
</tr>
<tr>
<td>5. released ( T ), holds ( D )</td>
<td>5 released ( D ), holds ( T )</td>
</tr>
</tbody>
</table>
Deadlock graph
Reusable resource graphs

- A directed graph is a pair \((N, E)\) where \(N\) is a set of nodes, and \(E\) is a set of edges (pairs of nodes).

- A reusable resource graph has the following interpretation and restrictions:

  1. \(N\) is divided into two sets: a set of process nodes \(\pi = \{p_1, p_2, \ldots, p_n\}\), and a set of resource nodes \(\rho = \{R_1, R_2, \ldots, R_m\}\). Process nodes are drawn with a circle, and resource nodes are drawn with a square.

  2. Each resource may have multiple units, drawn as small circles within the resource square (if there is only one unit, it may be omitted).
Reprocessable resource graphs II

3. The graph is *bipartite* with respect to \( \pi \) and \( \rho \). That is, there are no edges within \( \pi \) or within \( \rho \). An edge from \( \pi \) to \( \rho \) is a request; an edge from \( \rho \) to \( \pi \) is an assignment.

4. If \( t_j \) is the number of units in resource \( R_j \), there are no more than \( t_j \) assignment edges from \( R_j \).

5. The sum of requests and assignments from any process for resource \( R_j \) must not exceed the number of units in \( R_j \).
Example

\[ \text{S1} \xrightarrow{1} \text{S2} \xrightarrow{1} \text{S3} \xrightarrow{1} \text{S4} \]

request \quad assignment \quad release
How do we detect deadlock?

- The question: is it possible that every process can eventually make progress?

- Simulate the *most favorable* behavior for each unblocked process:
  - Each unblocked process acquires any resources it needs,
  - releases them *all*,
  - then becomes dormant
Resource graph reduction

• A reusable resource graph is *reduced* by a process $p_i$, which is neither blocked nor an isolated node, by removing all edges to and from $p_i$. This corresponds to that process acquiring all its resources, releasing them, and becoming dormant.

• A graph is *irreducible* if there are no nodes that can be reduced.

• A graph is *completely reducible* if there is a sequence of deletions that removes all the nodes in the graph.
Reduction example

Reduce p1
(p2 cannot be reduced)

Reduce p2
Reduction ordering independence

- *All* reduction sequences lead to the *same* irreducible graph
- Proof by contradiction
- Intuition: as a graph is reduced, it only becomes “easier” to reduce a process because edges to that process are deleted. If a process is unblocked to begin with, it will always be reduced. If there is a sequence of steps that make a process unblocked, that process will be part of every reduction sequence.
Reduction proof

Suppose \( S \) reduces to \( T_1 \) with sequence \( seq_1 \) and to \( T_2 \) with sequence \( seq_2 \) and \( T_1 \neq T_2 \). Then there must be a process \( q \) in \( seq_1 \) that is not included in \( seq_2 \) (or the reductions would be the same). Let \( seq_1 = (q_1, q_2, \ldots, q_n) \). By induction, show that \( q \neq q_i \):

**Base case** \( q \neq q_1 \) since \( q_1 \) is unblocked, and remains unblocked no matter how many reductions are performed (so \( q_1 \) is in \( seq_2 \)).

**Induction step** Assume \( q \neq q_i \) for \( i = 1, \ldots, j \). Since reduction by \( q_{j+1} \) is now possible in \( seq_1 \), process \( q_{j+1} \) must also be in \( seq_2 \) because there is a sequence of reductions that unblock \( q_{j+1} \).

Therefore, \( q \) is not part of \( seq_1 \) and we have a contradiction.
The deadlock theorem

- A state $S$ is a deadlock state iff the reusable resource graph for $S$ is not reducible.

**Necessary:** assume $S$ is a deadlock state, and process $p_i$ is deadlocked in $S$. That means $p_i$ is blocked in any state that follows from $S$. That means any sequence of reductions (process operations) leaves $p_i$ blocked, so the graph is not completely reducible.

**Sufficient:** assume $S$ is not completely reducible. Then there is a process that is blocked in every reduction sequence. Since a reduction sequence release all possible resources, $p_i$ will remain blocked forever.