Resource Allocation - Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

Dining Philosophers - Properties

Safety:
Freedom from deadlock
Mutual exclusion
A philosopher may not eat until he has exclusive use of the two forks adjacent to him.

assert EXCLUSION = forall [i:1..N] []!(EATING[i] && EATING[(i%N)+1])

Liveness:
Freedom from starvation - for individual and all
assert SOMEEAT = exists [i:1..N] []<> EATING[i]
assert NoSTARVATION = forall [i:1..N] [] <> EATING[i]

Naïve algorithm

Philosopher (i): Loop
think; sitdown;
snd get to right fork;
rcv ok;
snd get to left fork;
rcv ok;
eat;
snd put to right fork;
snd put to left fork;
arise; ...

Fork: Loop
(rcv get from right phil; snd ok to right phil)
or
(rcv get from left phil; snd ok to left phil)

Impossibility Result for Symmetric Algorithm

Theorem: There is no deterministic, distributed and symmetric solution to the Dining Philosophers Problem.

Informal Proof:
Assume there is a system A which solves the problem for n processes. Consider an execution of A that begins with all processes in the same initial state. Each process proceeds "round-robin" by executing a step at a time.

By induction on the number r of round-robin rounds, all processes are in identical states after r rounds. Therefore if any process is able to eat (liveness property), then all process will be able to eat. This violates the exclusion property.
Impossibility Result for Symmetric Algorithm

How do we overcome this?

Algorithms must have the following basic properties:

1. **Distinguishability**
   In every state of the system, at least one process in every set of conflicting (competing) processes must be distinguishable from the others in the set (asymmetry).

2. **Fairness**
   Conflicts should be resolved without detriment to a particular process.

Asymmetric algorithm - using IDs

Properties?

**Safety:**
Freedom from deadlock

**Liveness:**
STARVATION-FREEDOM

Probabilistic algorithm -

Identical Philosophers, but randomly choose which fork to take first, and replace it if unable to also take the second fork.
Probabilistic algorithm

**Fork:**
- Loop{
  - rcv get or wait from first phil;
  - snd ok;
  - loop{ rcv put; break
  - or rcv get from second phil;
  - snd lok
  }
}

Forks refuse requests if the fork is already taken.

**Distinguishability?**
- identical yet probabilistic to break the symmetry.

**Fairness?**
- different conditions

---

Probabilistic algorithm

**Properties?**

**Safety:**
- Freedom from deadlock

**Liveness:**
- **STARVATION-FREEDOM**
  - Strong fairness?
  - Weak fairness?
  - No fairness?

- What if philosophers don't replace forks, but retain them, as before?
- Can we improve fairness of allocation? (eg. cf. Peterson)

---

Probabilistic (courteous) algorithm

**Philosopher (i):**
- Loop{
  - Set flags in left and right forks;
  - getforks::False;
  - While !getforks
  - {Random choice:
    - getforks(left, right)
    - or getforks(right, left)}
  - eat;
  - ...}
  - getforks(first, second):
    - (snd wait to first fork; rcv ok;
    - snd get to second fork; rcv m;
    - if m=ok snd replace to first fork
    - else getforks::True
  }

Philosophers set flags to indicate hunger, and behave as probabilistic philosophers.
Probabilistic (courteous) algorithm

Fork: (initially flags unset and turn=neutral)

Loop
- set (left/right) flag whenever rcv setflag from phil;
- Snd ok to wait req iff available and (only one flag set or turn=neutral or turn=philosopher side).
- Snd ok to get req iff available else snd !ok.
- reset (left/right) flag and turn to other side when rcv put.
- (null) when rcv replace;

Properties?

Safety:
- Freedom from deadlock
  EXCLUSION

Liveness:
- STARVATION-FREEDOM
  Strong fairness?
  Weak fairness?
  No fairness?

Probability Vs Absolute certainty?
(practice Vs theory?)

demo

Hygienic Philosophers algorithm

Philosophers communicate directly with one another, passing forks and request tokens between them.

- the algorithm maintains an acyclic precedence graph which ensures freedom from deadlock, exclusion and starvation.

Clean forks are passed between philosophers

- A fork is either clean or dirty. A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher cleans a fork when passing it (he is hygienic).

- An eating philosopher does not satisfy requests for forks until he has finished eating.

- When not eating, philosophers defer requests for forks that are clean and satisfy requests for forks that are dirty.
Hygienic Philosophers algorithm

Preserve a precedence graph, where an edge from P1 to P2 indicates that P1 has precedence over P2.

- **Pi has precedence** over Pj iff
  1. Pi has the fork and it is clean
  2. Pj has the fork and it is dirty
  3. The fork is in transit from Pi to Pj

- **Depth**
  - Maximum number of edges from a process with no predecessors, which has depth 0.

Depth for each Pi?

**Distinguishability is provided by acyclicity.** It has been proven that...
- An acyclic graph ensures no starvation or deadlock.
- At least one philosopher has precedence over both his neighbours. He eventually receives each (clean) fork and retains it until he eats, since (by precedence) his requests are eventually satisfied by a finishing or thinking philosopher yielding to his request.
- If initially all forks are dirty and the graph is acyclic, then it remains acyclic.

The direction of an arc only changes when a philosopher starts eating, which results in both edges being simultaneously directed towards him.

**Fairness:**
- A process in conflict will rise to the top (to zero depth).
  - Each philosopher with precedence - at zero depth - redirects both arcs so as to yield precedence to its neighbours.

**Hygienic Philosophers algorithm**

**messages:**
- **forktoken** \( f \): passes fork \( f \) to neighbour which shares \( f \) (\( f \) can take the value left or right)
- **reqtoken** \( f \): passes request token for fork \( f \) to neighbour

**boolean variables:**
- **fork** \( f \): philosopher holds fork \( f \)
- **reqf** \( f \): philosopher holds request token for fork \( f \)
- **dirty** \( f \): fork \( f \) is at philosopher and is dirty
- **hungry/eating/thinking**: state of philosopher

**Initialisation:**
1. All forks are dirty
2. Forks distributed among philosophers such that the precedence graph is acyclic.
3. If \( u \) and \( v \) are neighbours then either \( u \) holds the fork and \( v \) holds the request token or vice versa.

**Hygienic Philosophers algorithm**

The algorithm for each philosopher is described as a set of rules guard=>action which form a single guarded command.

1. **Requesting a fork** \( f \):
   - hungry, reqf \( f \), ~fork \( f \) \( \Rightarrow \) SEND(reqtoken \( f \)); reqf \( f \) := false

2. **Releasing a fork** \( f \):
   - ~eating, reqf \( f \), dirty \( f \) \( \Rightarrow \) SEND(forktoken \( f \))
   - dirty \( f \) := false; fork \( f \) := false

3. **Receiving a request token for** \( f \):
   - receive(reqtoken \( f \)) \( \Rightarrow \) reqf \( f \) := true

4. **Receiving a fork token for** \( f \):
   - receive(forktoken \( f \)) \( \Rightarrow \) fork \( f \) := true (~dirty \( f \))

5. **Philosopher hungry to eating transition**:
   - hungry, fork(left), fork(right), (~reqf \( f \) or ~dirty \( f \) ) \( \Rightarrow \)
   - eating := true; hungry := false; dirty(left) := true; dirty(right) := true;

6. **Philosopher eating to thinking transition**:
   - eating, eating time expired \( \Rightarrow \) thinking := true; eating := false

7. **Philosopher thinking to hungry transition**:
   - thinking, thinking time expired \( \Rightarrow \) hungry := true; thinking := false
**Hygienic Philosophers**

**Properties?**

**Safety:** Freedom from deadlock

**EXCLUSION**

**Liveness:**

**STARVATION-FREEDOM**

**Strong fairness?**

**Weak fairness?**

**No fairness?**

---

**Notes**

This section has introduced asynchronous resource algorithms which must avoid deadlock, provide exclusion and prevent starvation.

Symmetry, Distinguishability and Fairness are important properties.

Probabilistic algorithms can provide a sound practical means for the avoiding deadlock and starvation, with probability 1.

Distributed precedence provides an asymmetric state with symmetric code, distinguishability and fairness.