**EE 382V: Distributed Systems** **Fall 2017**

# Lecture 8: September 15

*Lecturer: Vijay Garg* *Scribe: Shiyang Cheng*

## 8.1 Dining Philosophers Problem

The problem is described by a cycle of philosophers sitting between each other. The adjacent philosophers share some resources, such as fork. After one philosopher get all shared resources around him, then he can eat. This problem is an abstraction of many resources allocation problems in a network. We should are required to devise a protocal to coordinate access to the shared resources.

We model the problem as an undirected graph called a *conflict graph*. The vertices is processes and edge represent the resources shared between its start point(start process) and end point (end process). And at a same time, only one process can hold the resource. If the process need all the resources shared in the graph to perform its operation at all time, the conflict graph for a simple mutual exclusive algorithm is a complete graph, for example 4 vertices in figure 8.1.

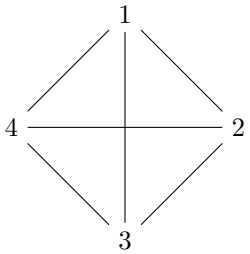


Figure 8.1: Conflict graph for mutex

### 8.1.1 Dining Philosophers Problem Requirement

- **Safety** two neighboring processes cannot be in critical section at same time.
- **Liveness** Any philosopher that in hungry eat in finite time
- **fairness**

### 8.1.2 Original Problem Statement

The original problem is defined the philosophers can think, be hungry and eat. Philosophers can decide to start thinking and eating. When the philosohpers are hungry, it will use the algorithm to decide how to get the resource needed to perform eating. This state transition is shown in Figure 8.2.

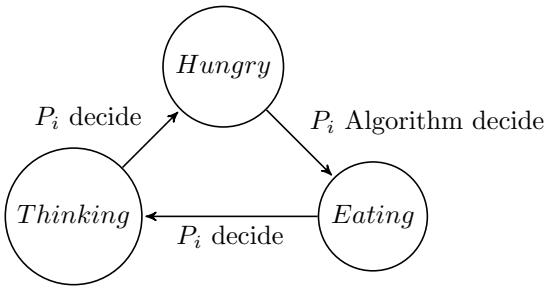


Figure 8.2: Original Problem State Transition

### 8.1.3 Rules

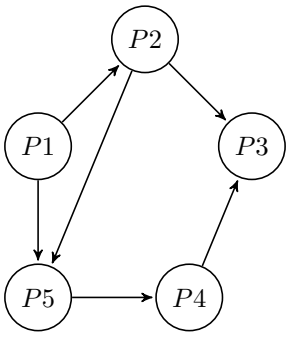


Figure 8.3: Conflict Resolution Graph

We use orientated acyclic graph to indicate a resolution of philosopher problem. We call the resolution graph is *conflict resolution graph*. For example Figure 8.3.

In the graph we call a vertex *source* if it does not have incoming edge. Any finite-directed acyclic graph must have at least one source.

**Invariant** Conflict resolution graph is acyclic at all time.

If one graph is conflict resolution graph, there will be at least one source node. We can proof this by contridiction. If there is no source in a graph, every vertex have a incoming edges in a graph. This graph must exists a cycle. This is conflict with the acyclic graph property.

**The algorithm rules**:

- **Eating rule**: A process can eat only if it has all the forks for the edges incident to it.
- **Edge reversal**: On finishing the eating session, a process reverses orientations of all the outgoing edges to incoming edges.

**Claim**: conflict resolution graph stays acyclic after applying eating rule.
Proof:

- *Mutual Exclusion*: If a resource is shared by processes by processes $P_i$ and $P_j$ then there is an edge between $P_i$ and $P_j$. Only oneof these processes can be a source in the graph. Hence, a resource is not used by more than one process at a time.

- *Starvation Freedom*:

  How many time everybody to become source.

  NumEats(u) := #times u become source

  **Claim**: let i and j be connected by a path of length d, then $|NumEat(i) - NumEats(j)| \leq k$.

  The proof is by induction on k, the length of the shortest path between $P_i$ and $P_j$. When k is 1, the assertion is clearly true, because once process $P_i$ has become a source, it can become a source only when the edge between $P_i$ to $P_j$ points to $P_j$ again. This can happen only after $P_j$ has become a source and reversed the edges. Assume that the claim is true for all nodes at distance less than k. Now assume that the distance between processes $P_i$ and $P_j$ is k. Consider any intermediate node $P_h$ on the shortest path between $P_i$ and $P_j$. From the induction hpythoesis[GARG02],

  $$|NumEat(i) - NumEats(h)| \leq dist(i, h)$$

  $$|NumEat(h) - NumEats(j)| \leq dist(h, j)$$

  Since $dist(i, j) = dist(i, h) + dist(h, j)$, we can get that

  $$|NumEat(i) - NumEats(j)| \leq k$$

### 8.1.4 Algorithm

**Invarant**:
Only hungry philosopher can hold clean forks.

**Variables**:

- *request token associated with a fork*: this indicate the process hold the token request the fork

- *dirty*: this means the fork has been used, before the process pass the fork, it will clean the fork

The conflict graph for the mutual exclusion on N processes is a complete graph on N nodes.For any philosopher to eat, she will need to request only those forks that she is missing. That can be at most $N - 1$. This results in $2(N - 1)$ message in the worst case.

$P_i$::
**var**
*hungry, eating, thinking*: boolean;
*fork(f)*: boolean // $P_i$ holds the fork *f*;
*request(f)*: boolean //$P_i$ holds the request token for the fork *f*;
*dirty(f)*:boolean //fork *f* is dirty;
intially
1. All forks are dirty.
2. Every fork and request token are held by different philosophers.
3. Conflict resolution graph is acyclic

To Request a fork:
if *hungry* and *request(f)* and $\neg fork(f)$ then
send request token for fork *f*;
*request(f) := false;*
Releasing a fork:
if *request(f)* and
*neg eating* and *dirty(f)* then
*dirty(f)* := *false*;
*fork(f)* := *false*;
send fork *f*;
Upon receiving a request token for fork *f*:
*request(f)*:= *true*;
Upon receiving a fork *f*:
*fork(f)*:= *true*;

## 8.2 Quorum-Based Algorithms

Quorum-based algorithms do not suffer from single point failure. Instead of ask the permission to access critical section, it will ask for subset of all nodes. If we assure that two request sets have nonempty intersection, we can assure their is no more than one process access the critical section at same time. A simple example is that the size of request set is $\lceil \frac{N+1}{2} \rceil$.

### 8.2.1 Maekawa's Algorithm

In Maekawa's algorithm, the nodes are distributed to square matrix. One node need to ask for permission from nodes in its current column and in its row. If all the node grant the access, it can enter the critical section. Let's check the below matrix. There are two nodes want to get the access. One node's quorum is indicated by red and the other indicated by green. The F node and K node can only grant access to one node at same time. So we can use this algorithm to achieve safety property.

$$\begin{pmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{pmatrix}$$

The quorums of *crumbing wall* is a little different from Maekawa's algorithm. The quorums is the union of full row and a reprentative form every row below that full row.

## References

[GARG02] Vijay K. GARG, Elements of Distributed Computing, pp. 95–97.

[GARG17] Vijay K. GARG, Distributed System Class Notes, pp. 105–105.