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

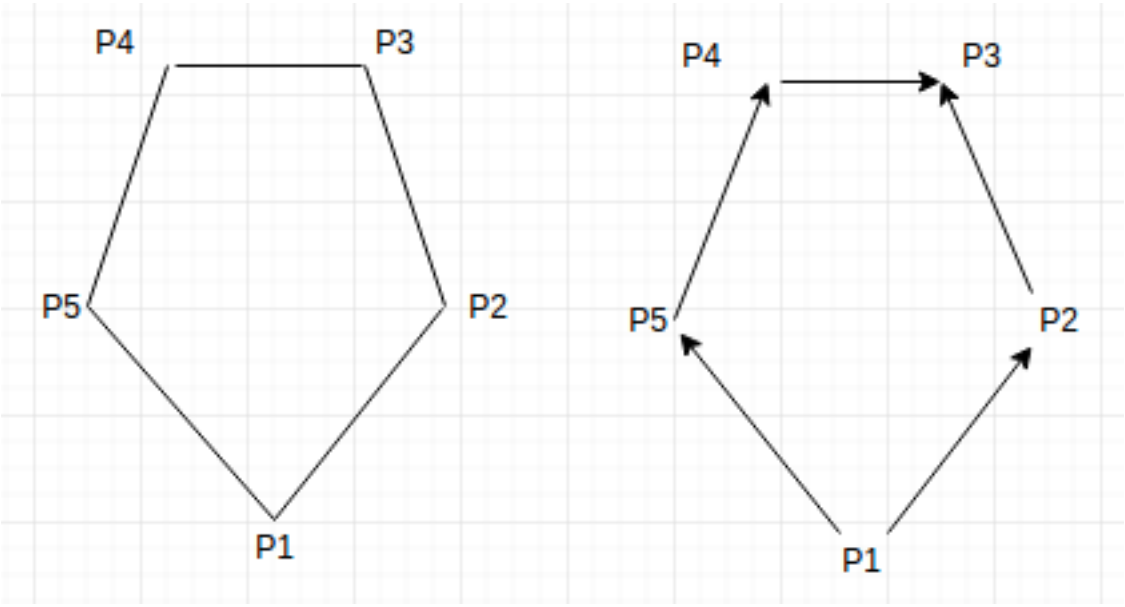
Lecture 5: September 14

Lecturer: Vijay Garg *Scribe: Saharsh Oza and Ryan Meek*

5.1 Dining Philosophers

The dining philosopher formulation solves the resource allocation problem in distributed systems. This is a different problem than mutual exclusion which is solved by the Lamport's / Ricart & Agarwala's algorithms.

The problem can be visualized as below. The figure on the left is a conflict graph. The edges between the processes represent resources shared between them. These are referred to as forks in the dining philosophers problem. An edge between 2 processes implies that there is a conflict between them. The figure on the left shows a conflict resolution graph. The process that has the tail end of the arrow has priority over the node that the arrow is pointing to and holds the fork. In the figure, P1 is a source node as it holds all the forks it needs.



The analogy between the dining philosopher's problem and resource allocation is the following. Every philosopher is a process and every edge represents a fork. 'Eating' is the act of entering a critical section.

It is important to note that there cannot be a cycle in the conflict resolution graph. If a node u has priority all nodes it is connected to, then it is a source node. We claim that there is at least one source node in a finite directed acyclic graph.
Proof:

- If a process with node u_0 has priority over all its connected nodes, it is a source node.

- If u_0 is not a source node, then there must be a process that has priority over it. We can follow the parent nodes u_1, u_2, ... u_k in this manner until we reach a node that has no parent node. If not, then we must encounter a node that we already traversed. This would create a cycle. But that would contradict our assumption that the graph is acyclic.

Requirements:

- Safety: Philosophers using the same fork cannot eat at the same time. ie Processes using a shared resource can't enter critical section at the same time.
- Liveness: Every philosopher must eventually get to eat. ie Every process must be able to get critical section eventually.

5.2 Heavy Load Algorithm

The heavy load algorithm of dining philosophers is defined by 2 rules.

- Eating Rule: A philosopher can eat only if he holds all the forks for the edges incident to it. ie a process can use a shared resource only if it is the source
- Edge Reversal: On finishing eating (if desired), the philosopher gives up all the forks it holds. ie a process must change the orientation of all the edges incident to it to point to itself. ie the process must become the sink.

Lemma: Edge reversal of a source maintains the acyclic property of the graph
Proof
Assume that this was not true and edge reversal of the source node u resulted in a cycle. Then there are 2 cases:

- u is part of the cycle. This is not possible as there are no outgoing edges from u after the edge reversal. As a result, there cannot be a cycle.
- u is not part of the cycle. This is not possible as it would imply that there was already a cycle, contradicting our assumption that the graph was initially acyclic.

The heavy load algorithm maintains both mutual exclusion (by ensuring no 2 processes with an edge between them can use a resource concurrently) and fairness (prevents starvation).

Starvation Freedom Let *numsource(i)* denote the number of times that Process i was the source. Let the path between processes i and j be d. We claim that $|numsource(i) - numsource(j)| <= k$

Proof
We prove this claim via induction. The base base of k=1, the assertion is clear. Now let us assume that this is true for all nodes at a distance less than k. Let the distance between processes P_i and P_j is k. Consider any intermediate node P_h on the path between P_i and P_j. From the induction hypothesis,
$|numsource(i) - numsource(h)| <= dist(i, h)$
$|numsource(h) - numsource(j)| <= dist(h, j)$
Since $dist(i, j) = dist(i, h) + dist(h, j)$, we get that $|numsource(i) - numsource(j)| <= k$

5.3 Light Load Algorithm

The key difference between the heavy and light load algorithm is that the philosopher gets a fork only when hungry. This reduces the to and fro communication of forks between processes.

The light load algorithm of dining philosophers is defined by a few rules.

- Forks: A fork can be either clean or dirty.
- A node u has priority of v if
  - u has the fork and it is clean
  - v has the fork and it is dirty
  - Fork is in transit from v to u
- All processes have 3 booleans. request, fork, and dirty.
- Initially: all forks are dirty and placed so that CRG is acyclic
- To request a fork f:
  - if(hungry(f) and request(f) and !fork(f)) then
    - send request(f) to other side
    - request(f) := false
- To release a fork(f)
  - if request(f) and !eating and dirty(f) then
    - dirty(f) := false
    - send fork f to the other side
    - fork(f) := false
- On eating: all forks get dirty
- Be able to eat: need all the forks
- Upon receiving request(f): request(f) := true
- Upon receiving fork(f) := true

Invariant A non hungry philosopher can only hold dirty forks.
Note: This is a hygienic solution as the philosophers always send clean forks. The light load algorithm (message complexity of $2 * m$) is better than Ricart & Agarwala (message complexity of $2(n-1)$) in terms of message complexity. Here m is the number of missing forks that have to be requested by a philosopher. This is useful when there are few processes in the system exchanging forks.

Theorem: Every hungry philosopher eventually gets to eat.
Proof
Define depth(v) as the length of the longest path from any source node to v. We prove this by induction. The base case is $depth(v) = 0$ is trivial. Let us assume that this is true to some depth(u). If we add a new node v. Now, $depth(v) > depth(u)$.

...