Resource Allocation

Pradipta De
pradipta.de@sunykorea.ac.kr
Outline

• Dining Philosophers Problem

• Drinking Philosophers Problem
Dining Philosophers Problem

Each philosopher goes through,
Think (R) ➔ hungry (T) ➔ eat (C) ➔
think (E)

Needs two forks to eat ➔ mutual exclusion problem on two resources

Fork to right = f(i)
Fork to left = f(i+1)
Simple (Wrong) Solution

• Each process on entering T region,
  – Wait for its right fork, then for its left fork
  – After getting both forks, execute C
  – Release both forks after exiting C

• If all the processes are identical, and the shared variables (forks) have same initial value, then there is no solution to DP problem
  – No symmetric solution to DP problem ... Why ?
Simple (Wrong) DP Solution

• Does it satisfy Mutual Exclusion?
  – A process reaching C does have two forks

• Does it satisfy progress property?
  – All processes enter try region one after another
  – Each gets the right fork, but now there is deadlock over left fork

• Must break symmetry of the ring
  – Unique identifiers to create ordering
  – Different programs for odd/even id processes
  – Initialize the shared variables differently
Right-Left DP Algorithm

- Main idea
  - Careful design of the order in which forks are sought

Assume an execution as follows:
P5 has both forks
P4 gets right fork, waits for left fork
P3 gets right fork, waits for left fork
P2 gets right fork, waits for left fork

Length of waiting chain = 3 i.e. n-2

Processes enter CS sequentially → time to enter CS is proportional to waiting chain length
Right-Left DP Algorithm

• Assume \( n \) processes (\( n \) is even)
• Two different programs for odd and even indexed processes

• Odd numbered processes seek
  – Right fork first, then left fork
• Even numbered processes seek
  – Left fork first, then right fork
Right-Left DP Algorithm

still need to break the tie between two processes requesting the same fork -- use the process index

has bounded time to enter CS → guarantees progress
**Conflict Graph**: An undirected graph in which each node represents a process and an edge between process $P_i$ and $P_j$ denotes one or more resources are shared between $P_i$ and $P_j$.

Precedence Graph adds orientation to each edge generating an acyclic graph, where edge $P_i$ to $P_j \Rightarrow P_i$ has precedence over $P_j$.

Conflict graph for single resource shared among all processes ME problem?
Edge Reversal Algorithm

Acyclic orientation with p2 and p4 as sources, or processes with precedence over its neighbors

Reverse orientation of edges after use

Orientation after p2 and p4 leave CS
Edge Reversal Algorithm

• Assume that a philosopher is “not hungry”, but the edge still reverses (light load condition)

• Do not reverse edge when CS completed
  – Process requests fork when hungry

• Distinguish two cases
  – Process has fork, but not used it (clean fork)
  – Process has fork, and have used it (dirty fork)

• Edge is from u->v (u has precedence over v) if
  – u holds fork, and it is clean
  – v holds fork, and it is dirty
  – Fork is in transit from v to u

• Start from acyclic precedence graph, and at each step algorithm maintains the graph to remain acyclic
Drinking Philosophers Problem

- A philosopher *does not always require* all the resources it shares with its neighbors.
  - May need only a subset of the resources, and the required resources may change across sessions

- Example: two processes Pi and Pj can access different files in one session (⇒ allow concurrent access), but same files in another session (⇒ requires conflict resolution)

- Philosopher states = {tranquil, thirsty, drinking}
- Requires a set of resources (modelled as bottles) when thirsty to enter drinking phase
Drinking Philosopher Solution

• Builds on Dining Philosophers Solution
• Use forks as auxiliary resources to resolve conflict
  – Bottles are real resource for sharing
  – Forks and bottles are shared between two processes

• Rules:
  – Thinking, thirsty P becomes hungry in finite time
  – Eating, nonthirsty P starts thinking
  – U sends the bottle to v, on receiving req, iff
    • U does not need the bottle
    • U is not drinking and does not hold the fork for edge u,v
Drinking Philosophers Algorithm

```javascript
var thirsty, drinking, tranquil: boolean;
bot(b), request(b), need(b): boolean;

initially
bottle and its request token are held by different philosophers.

To request a bottle b:
if thirsty, request(b), need(b), ¬bot(b) then
send request token for bottle b;
request(b) := false;

To release a bottle b:
if request(b), bot(b) then
if ¬ [ need(b) and ((state = drinking) or fork(f)) ] then
send bottle b;
bot(b) := false;

Upon receiving a request token for bottle b:
request(b) := true;

Upon receiving a bottle b:
bot(b) := true;
```

- Fork is with process u, Thirsty ➔ need bottle
- Bottle not with u
- Receives req, and has bottle, Satisfy condition to release bottle, then send bottle