Operating Systems

Karim Lounis

Queen’s University, Canada

Winter’19, Jan-Apr 2019
Dinning Philosophers problem
Dinning Philosophers problem

This problem consists of: A set of five philosophers sitting around a table $P = \{p_0, p_1, p_2, p_3, p_4\}$ & five forks $F = \{f_0, f_1, f_2, f_3, f_4\}$. Each philosopher $P_i$ has one fork in his left side and one fork in his right side s.t:

- Each philosopher has a plate of spaghetti or rice in front of him.
- A philosopher spends his time alternating between thinking & eating.
- A philosopher $P_i$ needs both forks ($f_i \& f_{i+1}$) to be able to eat.
- If a philosopher grabs one fork, it does not put it down till getting the second fork and eating (Hold and wait property).
- A philosopher is not allowed to take a fork from another philosopher’s hands (No preemption property).
- Parallelism increases with the number of philosophers.

There exist multiple solutions, we discuss some of them.
Dinning Philosophers problem

**Solution 1:** Pick up a fork on the right side then pick up a fork on the left side.

Semaphore Fork [5]; /* ∀i ∈ {0, 1, 2, 3, 4} Fork[i]=1 */

**Philosopher(integer i)**

```c
while(1)
{
    Think();
    acquire(Fork[i]);
    acquire(Fork[(i+1) mod 5]);
    Eat();
    release(Fork[i]);
    release(Fork[(i+1) mod 5]);
}
```

Dinning Philosophers problem

**Solution 1:** Pick up a fork on the right side then pick up a fork on the left side.

Semaphore Fork [5]; /* \( \forall i \in \{0,1,2,3,4\} \) Fork\[i\]=1 */

**Philosopher(integer i)**

```c
{  
    while(1)  
    {  
        Think();  
        acquire(Fork[i]);  
        acquire(Fork[(i+1) mod 5]);  
        Eat();  
        release(Fork[i]);  
        release(Fork[(i+1) mod 5]);  
    }  
}
```
Dinning Philosophers problem

**Solution 2:** A mix of left handed and right handed philosophers.

Semaphore Fork [5]; /* ∀i ∈ {0, 1, 2, 3, 4} Fork[i]=1 */

Philosopher(integer i)
{
    while(1)
    {
        Think();
        acquire(Fork[Min(i, (i+1) mod 5)]);
        acquire(Fork[Max(i, (i+1) mod 5)]);
        Eat();
        release(Fork[Min(i, (i+1) mod 5)]);
        release(Fork[Max(i, (i+1) mod 5)]);
    }
}
Dinning Philosophers problem

Solution 2: A mix of left handed and right handed philosophers.

Semaphore Fork [5]; /* ∀i ∈ {0, 1, 2, 3, 4} Fork[i]=1 */

Philosopher(integer i)
{
    while(1)
    {
        Think();
        acquire(Fork[Min(i, (i+1) mod 5)]);
        acquire(Fork[Max(i, (i+1) mod 5)]);
        Eat();
        release(Fork[Min(i, (i+1) mod 5)]);
        release(Fork[Max(i, (i+1) mod 5)]);
    }
}

This solution is a deadlock-free and starvation-free solution.
Dinning Philosophers problem

Solution 3: Allows at most 4 philosophers to attempt access to the forks at a given time (uses a Semaphore room=4;).

Solution 4: Two forks are requested at a time.
Using semaphores for synchronization

The following steps describe an informal procedure of writing codes with semaphores:

1. Understand the problem to solve.

   “Understanding a question is half an answer” - Socrates 470-399 BJC

2. Determine which processes or threads are needed.

   e.g., Readers and writers

3. Write down informally, conditions under which a process must wait for another:

   e.g., A reader waits if a writer is writing

4. Figure out what counters and semaphores are needed.

5. Code and debug.
Producer & Consumers problem
Producer & Consumers problem

This problem consists of: (1) A set of producers threads $P = \{p_1, \ldots, p_n\}$ & (2) a set of consumer threads $C = \{c_1, \ldots, c_m\}$. Each producer thread $p_i$ produces an item and places it into a buffer $B$, and each consumer $c_i$ consumes one item from that buffer such that:

- A producer $p_i \in P$ produces an item if the buffer is not full.
- A consumer $c_i \in C$ consumes an item if the buffer is not empty.
- Producers and consumers must have exclusive access to the buffer.
- The buffer $B$ is bounded i.e., has a limited size $N$.
- A.k.a., Bounded buffer problem.

This principle is used in Unix pipes: $\$ netstat -natp | grep chrome$
Producers & Consumers problem

Solution: One consumer and one producer.

Semaphore m = N; Semaphore e = 0; Semaphore b = 1;

Producer() {

while(true) {
    acquire(m);
    item = Produce();
    acquire(b);
    Place(item, B);
    release(b);
    release(e);
}
}

Consumer() {

while(1) {
    acquire(e);
    acquire(b);
    Take(B, item);
    release(b);
    Consume(item);
    release(m);
}
}
End.