Operating Systems - 2

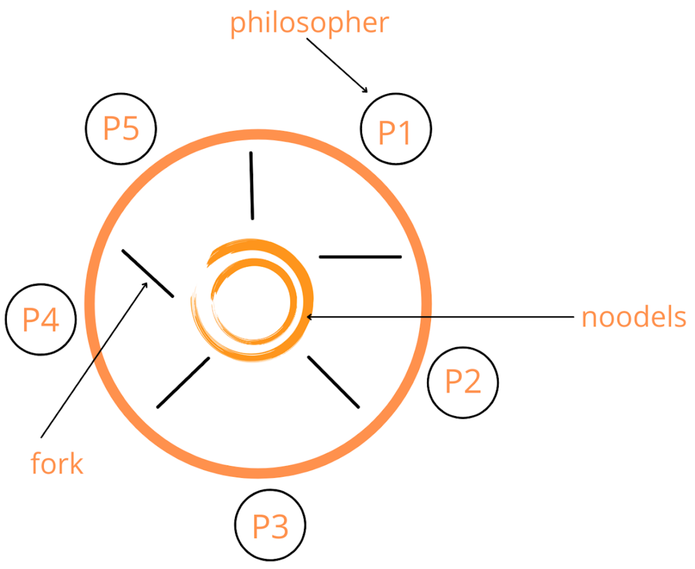
Dining Philosophers Problem

202000223 - تسنيم سامح سليمان محمود
202000945 - منة الله محمد سيد عيسوي
202001066 - يمنى محمد عبد القادر عبد الغني
201901009 - احمد عيد فوزي سيد
202000958 - مهند نصر الدين زغلول حسين
202000664 - كريم سلامه عبدالملك عثمان
202000469 - صلاح محمد صلاح محمد احمد

Problem:

N (we are making them 5 in our solution) philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the center of the table is a large plate of spaghetti.

- There is one chopstick between each philosopher.
- To eat, a philosopher must pick up their two nearest chopsticks.
- A philosopher must pick up one chopstick at a time, not both at once.



Solution Pseudocode:

Dining-Philosophers Solution Using Monitor

We demonstrate monitor ideas by proposing a deadlock and starvation free solution to the Dining-Philosophers problem. The monitor is used to control access to state variables and condition variables. It only notifies when to enter and exit the segment.
This approach imposes the limitation that a philosopher may only take up her forks if both the forks are available.
To code this solution, we must distinguish between three situations where we could see a philosopher. We introduce the following data structure for this purpose:

- Thinking: When the philosopher does not want to use either fork.
- Hungry: When a philosopher wishes to use the forks, i.e., he wants to go to the critical section.
- Eating: When the philosopher has both forks, i.e., he has entered the critical section.

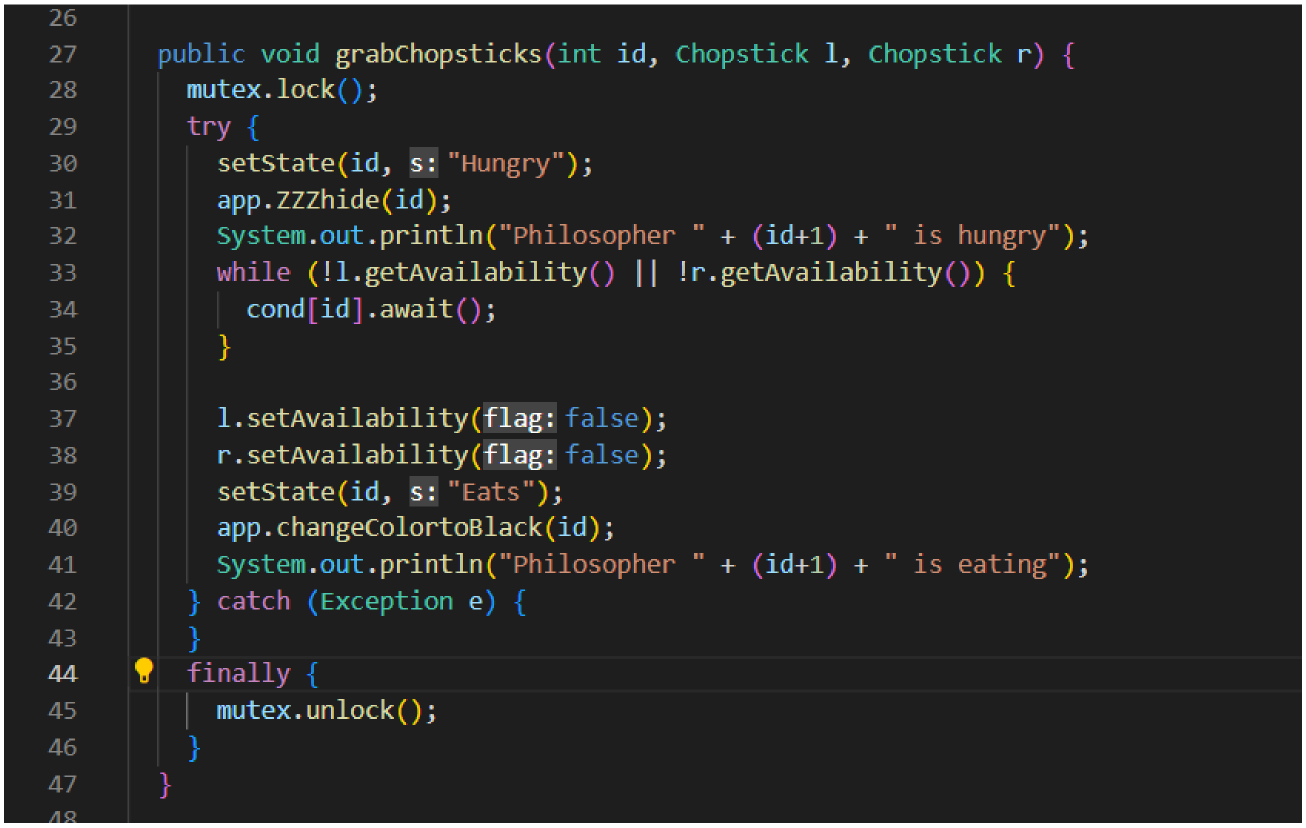
```
grabChopsticks( )
{
Set State of Philosopher = Hungry
While Left or Right Chopstick aren't available
{
Wait for the condition
}
Set Left and Right Chopsticks to Unavailable
Set State of Philosopher = Eats
}
eat( )
releaseChopsticks( )
{
Set State of Philosopher = Thinking
Set Left and Right Chopsticks to Available
Signal the philosophers on the right and left with the condition
}
think( )
```

Possibility of Deadlock:

If philosophers take one chopstick at a time, taking a chopstick from the left and then one from the right, there is a danger of deadlock.
This possibility of deadlock means that any solution to the problem must include some provision for preventing or otherwise dealing with deadlocks.

Solution of a Deadlock:

- **While Left or Right Chopstick aren't available**



```java

public void grabChopsticks(int id, Chopstick l, Chopstick r) {
  mutex.lock();
  try {
    setState(id, s: "Hungry");
    app.ZZZhide(id);
    System.out.println("Philosopher " + (id+1) + " is hungry");
    while (!l.getAvailability() || !r.getAvailability()) {
      cond[id].await();
    }

    l.setAvailability(flag: false);
    r.setAvailability(flag: false);
    setState(id, s: "Eats");
    app.changeColortoBlack(id);
    System.out.println("Philosopher " + (id+1) + " is eating");
  } catch (Exception e) {
  }
  finally {
    mutex.unlock();
  }
}
```

No philosopher can eat unless his left or right chopstick available, so this assures that no deadlock can happen as no two neighbouring philosophers can eat at the same time.

Possibility of Starvation

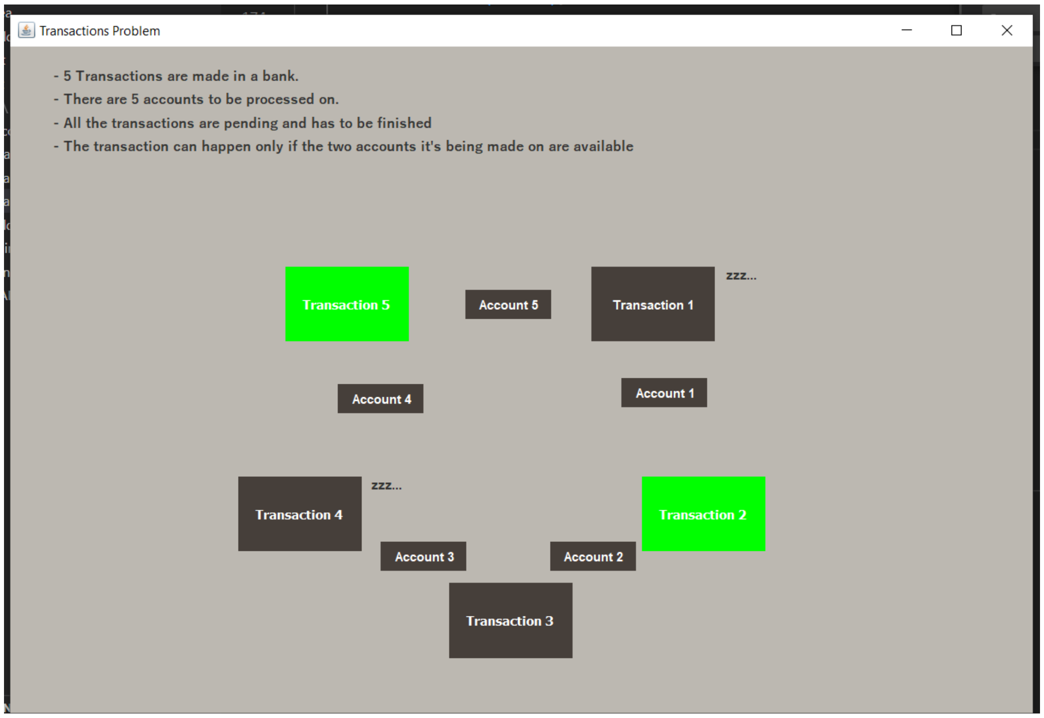
If philosophers take two chopsticks at a time, there is a possibility of starvation. Philosophers P2 & P5 and P1 & P3 can alternate in a way that starves out philosopher P4.
This possibility of starvation means that any solution to the problem must include some provision for preventing starvation.

Solution of a Starvation:

To prevent starvation in multithreading, it is important to ensure that each thread has a fair chance to make progress and access shared resources. This can be achieved by using a lock hierarchy that allows threads to wait for resources to become available without consuming all the resources of the system. The monitor used in the solution has a waiting queue that lets the last thread waiting to continue once the SignalAll() method is executed, which leads to no starvation.

Real World Example:

We have N transactions to be performed on N bank accounts, but the account must be used in only one transaction at the same time to avoid confusion. To execute the transaction the thread must lock both accounts to ensure the correct value is debited from one account and crediting to another.


Transactions Problem

- 5 Transactions are made in a bank.
- There are 5 accounts to be processed on.
- All the transactions are pending and has to be finished
- The transaction can happen only if the two accounts it's being made on are available