Life in Deadlock II

Lecture 12

Alma Rahat

CS-210: Concurrency

3 March 2021

What did we do in the last session?

- Condition synchronisation: library example.
- Introduction to deadlock with dining philosophers.

In this lecture...

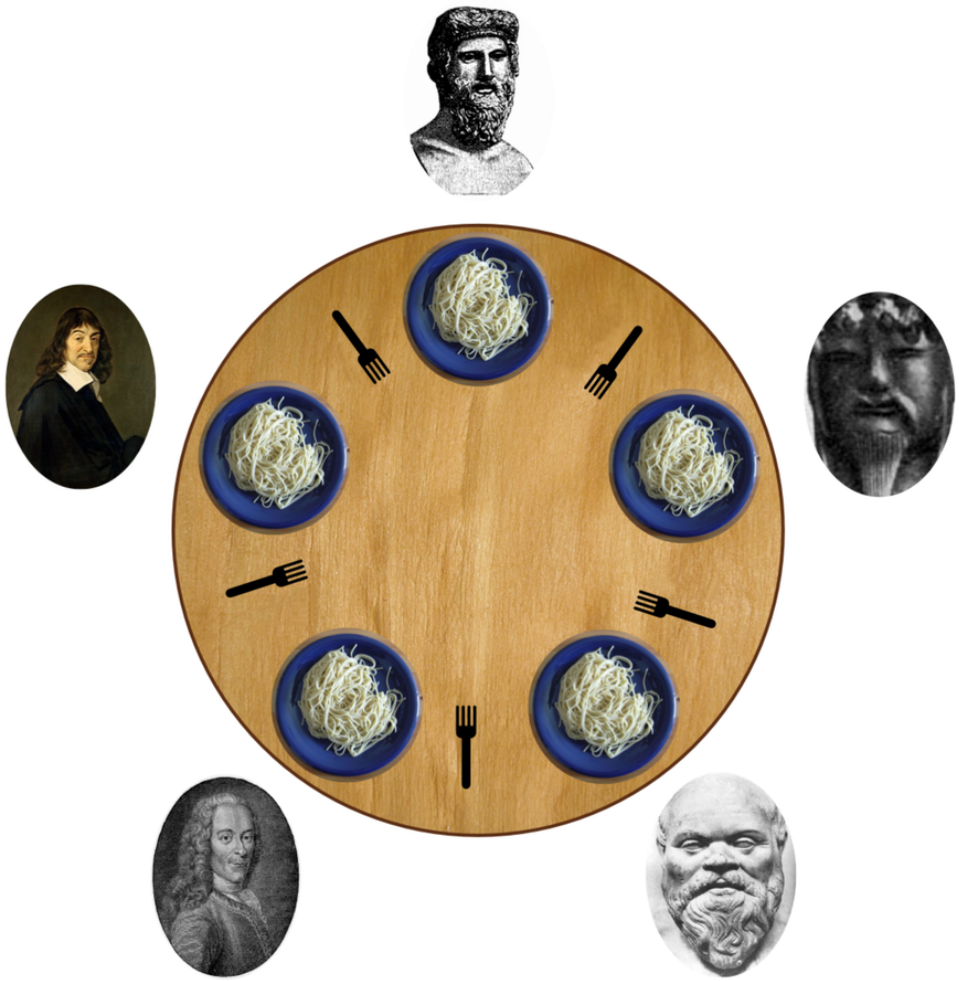
Learning outcomes.

1. To explain the necessary and sufficient conditions for deadlock.
2. To understand and apply Coffman conditions in order to analyse deadlocks in a given scenario.
3. To describe potential avenues to break deadlocks.
4. To apply resource allocation graphs to determine whether a system has deadlock or not.

Outline.

1. Review the necessary and sufficient conditions for deadlock.
2. Modelling dining philosophers.
3. Deadlock handling.

Dining Philosophers' Problem



source: wikipedia

Five philosophers are dining together, and they are sharing five forks to eat spaghetti. A philosopher arbitrarily sits down to eat, and then picks up the right fork first and then the left fork. This will lead to a deadlock situation, as in no progress can be made. How?

Everyone waits for the left fork to become available in the following scenario:

$P_1 \rightarrow$ right fork $\rightarrow P_2 \rightarrow$ right fork $\rightarrow P_3 \rightarrow$ right fork $\rightarrow P_4 \rightarrow$ right fork $\rightarrow P_5 \rightarrow$ right fork

Necessary and Sufficient Condition

Swansea University Prifysgol Abertawe

For deadlock to occur, **all** of the following must occur.

1. Serially reusable resources: the processes involved shared resources which they use under **mutual exclusion**. [Philosophers share forks.]
2. Incremental acquisition: processes **hold** on to resources already allocated to them while **waiting** to acquire additional resources. [If holds right fork, waits for left to become available.]
3. **No pre-emption**: once acquired by a process resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily. [Philosopher is responsible for letting go of the fork.]
4. Wait-for-cycle: a **circular** chain (or cycle) of processes holds a resource which its successor in the cycle is waiting to acquire. [One waits for another to release fork and *vice-versa*.]

These conditions were first proposed by Coffman in 1971. This is why they are often referred to as the Coffman conditions.

Any questions?

Exercise

Source: www.reddit.com/r/Wellthatsucks/comments/a8ffke/traffic_deadlock/

Consider the traffic deadlock scenario above. Can you analyse how the necessary and sufficient conditions of deadlock apply here?

Please go to www.menti.com and use the code 30 19 32 9.

Solution

1. Serially reusable resources (or mutual exclusion): Only a car can occupy a particular part of the road.
2. Incremental acquisition (or hold and wait): A car is waiting to move on to the next section.
3. No pre-emption: We cannot just pick up and move a car.
4. Wait-for-cycle: Each car is waiting on the other to move.

Another Simple Example

Timeline	Thread 1	Thread 2
t_0	lock(x);	lock(y);
$t_1 - t_m$	// do something	// do soemthing
$t_m - t_\infty$	lock(y); ✗	lock(x); ✗
✗	// do something	// do something
✗	unlock(y);	unlock(x);
✗	unlock(x);	unlock(y);

One is waiting on the other to let go!

Any questions?

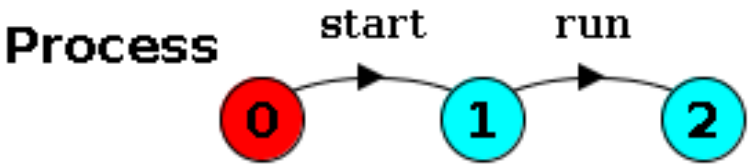
Review: Modelling and Coding Lifecycle

1. Identify component processes.
2. Identify states and actions.
3. Identify variables and logic.
4. FSP code and verification.
5. Java code.

Deadlock Analysis: Primitive Processes

When we mean to end a program, the LTSA tool will still report that there is a deadlock, but it is intentional, and we can safely ignore this. We will try to identify problems that are unintentional.

```
Process = (start -> run -> STOP).
```




```
Trace to DEADLOCK:
        start
        run
```

Deadlock Analysis: Three Philosophers

Swansea University
Prifysgol Abertawe

Think about the dining philosophers, but with three philosophers instead.

Deadlock Analysis: Three Philosophers

Swansea University
Prifysgol Abertawe

Think about the dining philosophers, but with three philosophers instead.

```
Fork = (acquire -> release -> Fork).
Philosopher = (sit -> right.acquire -> left.acquire
-> eat -> left.release -> right.release -> stand ->
Philosopher).
||ThreePhil = ({a,b,c}:Philosopher).
||Fork1 = ({a.right, b.left}::Fork).
||Fork2 = ({b.right, c.left}::Fork).
||Fork3 = ({c.right, a.left}::Fork).
||Table = (ThreePhil || Fork1 || Fork2 || Fork3).
```

Deadlock Analysis: Three Philosophers

Think about the dining philosophers, but with three philosophers instead.

All three philosophers holding their right fork.

```
  a.sit
✓ a.right.acquire
  c.right.acquire
  c.left.acquire
  c.eat
  c.left.release
  c.right.release
  c.stand
  b.stand
  b.sit
  c.sit
✓ c.right.acquire
✓ b.right.acquire
```

Any questions?

Deadlock Handling

There are two groups of developers that should take this responsibility:

- Operating System (OS) Developer
- Application Developer

Approaches for handling deadlocks:

- Ignorance (is bliss!) – Both OS and app. dev.
- Prevention – Primarily app. dev.
- Avoidance – Primarily OS dev.
- Detection and Recovery – Primarily OS dev.

Many current OSs cannot prevent deadlocks. Different OS adopt different strategies for handling potential deadlocks.

Prevention: Breaking Conditions

Swansea University
Prifysgol Abertawe

Avoid deadlock by design: ensure that the system breaks at least one of the Coffman conditions (usually people target the final condition of circular wait).

Prevention: Breaking Conditions

Swansea University Prifysgol Abertawe

Mutual exclusion. Difficult to eliminate completely, when we are dealing with shared resources. We may allow some processes to be non-exclusive (e.g. file read), but for others (e.g. file write) must have exclusion by design.

Hold and wait. Request all resources at the beginning, and block processes until requests can be granted. Must know all required resources in advance, which is difficult.

No pre-emption. Design processes such that it lets go of any resources it is holding if request for another resource is denied, timed waiting on resources, or force a required resource to be released by processes (more difficult due to priority). Important to consider rollback.

Circular wait. Impose strict ordering between resources for acquisition.

These methods allow a safe system by design, which is our main aim.

Any questions?

Code for Dining Philosophers

```
public class Fork {
    private boolean isTaken = false;
    public synchronized void acquire()
            throws InterruptedException{
        while (isTaken == true)
            wait();
        this.isTaken = true;
        notifyAll();
    }
    public synchronized void release()
            throws InterruptedException{
        while (isTaken == false)
            wait();
        this.isTaken = false;
        notifyAll();
    }
}
```

Code for Dining Philosophers

```
public class Philosopher implements Runnable {
    private Fork leftFork;
    private Fork rightFork;
    private String name;
    private double scaler = 1000;
    Philosopher(String name, Fork left, Fork right){
        this.name = name;
        leftFork = left;
        rightFork = right;
    }
    public void setLeftFork(Fork leftFork) {
        this.leftFork = leftFork;
    }
    public void setRightFork(Fork rightFork) {
        this.rightFork = rightFork;
    }
```

Code for Dining Philosophers

```
public void sit()
        throws InterruptedException{
    double random = Math.random();
    System.out.println(name + " is trying to sit down.");
    Thread.sleep((long) (random*scaler));
    System.out.println(name + " has sat down.");
}
private void eat()
        throws InterruptedException {
    double random = Math.random();
    System.out.println(name + " is trying to eat.");
    Thread.sleep((long) (random*scaler));
    System.out.println(name + " has eaten.");
}
private void stand(){
    System.out.println(name + " has stood up.");
}
```

Code for Dining Philosophers

```
@Override
public void run() {
    while(true){
        try {
            sit();
            leftFork.acquire();
            System.out.println(name + " has taken left fork.");
            rightFork.acquire();
            System.out.println(name + " has taken right fork.");
            eat();
            rightFork.release();
            System.out.println(name + " has released right fork.");
            leftFork.release();
            System.out.println(name + " has released left fork.");
            stand();
        } catch (InterruptedException ex) {
            System.out.println(name + " was interrupted!");
            break;
        }
    }
}
```

Code for Dining Philosophers

```
public static void main(String[] args) throws InterruptedException {
    int numberOfPhil = 3;
    Philosopher[] philosophers = new Philosopher[numberOfPhil];
    Fork[] forkSet = new Fork[numberOfPhil];
    Thread[] threads = new Thread[numberOfPhil];
    for(int i = 0; i < numberOfPhil; i++){
        forkSet[i] = new Fork();
    }
    for (int i = 0; i<numberOfPhil; i++){
        int leftForkInd = i;
        int rightForkInd = (i+1)%forkSet.length;
        philosophers[i] = new Philosopher(Integer.toString(i),
                                          forkSet[leftForkInd],
                                          forkSet[rightForkInd]);
        threads[i] = new Thread(philosophers[i]);
        threads[i].start();
    }
    System.out.println("Simulation started!");
    System.out.println("Main thread is sleeping!");
    Thread.sleep(5000);
```

Code for Dining Philosophers

```
1 has sat down.
1 has taken left fork.
1 has taken right fork.
1 is trying to eat.
2 has sat down.
1 has eaten.
1 has released right fork.
1 has released left fork.
1 has stood up.
2 has taken left fork.
1 is trying to sit down.
2 has taken right fork.
2 is trying to eat.
0 has sat down.
1 has sat down.
1 has taken left fork.
2 has eaten.
2 has released right fork.
0 has taken left fork.
```

A Solution to Dining Philosophers

Swansea University
Prifysgol Abertawe

Let's aim to break the *wait-for-cycle*. How?

A Solution to Dining Philosophers

Swansea University Prifysgol Abertawe

Let's aim to break the *wait-for-cycle*. How?

If the even numbered Philosopher takes the right fork first and the odd numbered Philosopher takes the left fork first, the cycle is broken!

```
for (int i = 0; i<numberOfPhil; i++){
    int leftForkInd = i;
    int rightForkInd = (i+1)%forkSet.length;
    if (i%2 != 0){
        // flip the fork allocation
        leftForkInd = (i+1)%forkSet.length;
        rightForkInd = i;
    }
```

Exercise

Swansea University
Prifysgol Abertawe

Given what we know about condition synchronisation and Coffman conditions, can you suggest a condition that would break the deadlock here?

Please go to `www.menti.com` and use the code 20 83 84 4.

Solution

Only allow $N - 1$ philosophers to sit down at a time.

```
const N = 3
set Names = {a,b,c}
Butler(Capacity=2) = ChairFull[0],
ChairFull[i:0..Capacity] = (when i<Capacity
Names.sit -> ChairFull[i+1] | when i>0 Names.stand ->
ChairFull[i-1]).
||ButleredTable = (Butler || Table).
```

There are no deadlocks in this case!

Other possible solutions: pick the lower order fork first, use timed waiting on a fork, etc.

You should try and implement some of these solutions in Java at home.

Butlered Dining Philosophers Code

```
  private Semaphore butler;
public void sit()
        throws InterruptedException{
    butler.acquire();
    double random = Math.random();
    System.out.println(name + " is trying to sit down.");
    Thread.sleep((long) (random*scaler));
    System.out.println(name + " has sat down.");
}
private void stand(){

    System.out.println(name + " has stood up.");
    butler.release();
}
```

Butlered Dining Philosophers Code

```
Simulation started!
Main thread is sleeping!
1 is trying to sit down.
0 is trying to sit down.
0 has sat down.
0 has taken left fork.
0 has taken right fork.
0 is trying to eat.
0 has eaten.
0 has released right fork.
0 has released left fork.
0 has stood up.
0 is trying to sit down.
1 has sat down.
1 has taken left fork.
1 has taken right fork.
1 is trying to eat.
```

Here, there is a fairness issue, sometimes philosopher 2 never gets a go. this is because release only wakes up one of the waiting threads.

Butlered Dining Philosophers Code

Constructors

Constructor and Description
Semaphore(int permits) Creates a Semaphore with the given number of permits and nonfair fairness setting.
Semaphore(int permits, boolean fair) Creates a Semaphore with the given number of permits and the given fairness setting.

```
Simulation started!
Main thread is sleeping!
0 is trying to sit down.
1 is trying to sit down.
0 has sat down.
0 has taken left fork.
0 has taken right fork.
0 is trying to eat.
0 has eaten.
0 has released right fork.
0 has released left fork.
0 has stood up.
2 is trying to sit down.
2 has sat down.
2 has taken left fork.
2 has taken right fork.
2 is trying to eat.
```

Any questions?

Summary

- We should aim to design deadlock free systems.
- Four main methods to break deadlocks:
  - Ignorance.
  - Prevention by design.
  - Avoidance.
  - Detection and Recovery.