

# Life in Deadlock III & Amdahl's Law

## Lecture 13

Alma Rahat

CS-210: Concurrency

09 March 2021



# What did we do in the last session?

---

- Introduction to deadlock.
- Introduction to necessary and sufficient conditions.
- Deadlock handling.

## Learning outcomes.

- 1 To explain how to break deadlocks.
- 2 To explain how to use resource allocation graphs for deadlock detection.
- 3 To apply Amdahl's law in determining the potential speed up from program optimisation.
- 4 To describe avenues to define safety properties in FSP.

## Outline.

- 1 Code example of dining philosophers and breaking deadlock.
- 2 Deadlock detection: Resource Allocation Graphs (RAGs).
- 3 Amdahl's law and its applications.
- 4 Introduction to properties.

```
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).
```

Three key classes: Fork, Philosopher and Table.

```
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();  
    }  
}
```

```
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;  
    }  
    ...  
}
```

```
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);
}
```

```
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.
```

# Recap: Deadlock Prevention by Design

---

- 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.

Two twins fight over goalkeeper gloves. Sometimes this results in each one getting one glove. To stop the fight their big sister takes the right hand glove from whoever is holding it and gives it to the other. Does this lead to a deadlock?

Please go to [www.menti.com](https://www.menti.com) and use the code **1410 9896**.

Two twins fight over goalkeeper gloves. Sometimes this results in each one getting one glove. To stop the fight their big sister takes the right hand glove from whoever is holding it and gives it to the other. Does this lead to a deadlock?

Please go to [www.menti.com](https://www.menti.com) and use the code **1410 9896**.

Solution: No deadlock!

**Mutual exclusion.** Yes, sharing a pair of gloves.

**Hold and wait.** Yes, if one of the brothers gets one part of the pair, reluctant to let it go.

**No pre-emption.** No, the sister forces one of them to let go.

**Circular wait.** Potentially.

# A Solution to Dining Philosophers

---

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

# A Solution to Dining Philosophers

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;  
    }  
}
```

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](http://www.menti.com) and use the code **6494 0688**.



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: use timed waiting and release on a fork, etc.

Some of the solutions are on github repository. You should try and implement some of these solutions in Java on your own.

# Any questions?

---



## Deadlock Handling: An Operating System Perspective

A dynamic (run-time) scheme.

- Do not start a process if its demands might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation may lead to deadlock.

Essentially, the following apply:

- Assume we know about the maximum resources required.
- Track allocations in real-time.
- When a request is made, only grant if guaranteed no deadlock even if all others take maximum resources.

Not very useful in practice, as we need to know a lot about the processes, and their needs a-priori.

Another dynamic scheme: probe programs regularly to construct a resource allocation graph, and see if there is at least one way to progress. If not, then we are deadlocked.

## Recovery

Once detected, we have the following options:

- Abort all processes (common in many OSs).
- Back up to a predefined check point.
- Abort processes successively (i.e. one-by-one) until deadlock no longer exists.

Selection of processes for recovery may be based: processor time consumed, estimated time remaining, amount of output produced so far, etc.

# Resource Allocation Graph (RAG)

Resource Allocation Graphs allow us to see the state of a system, and investigate whether deadlock is a possibility or not.

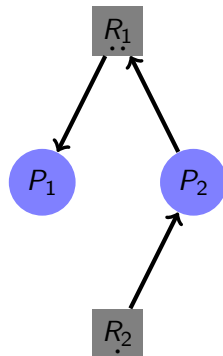
$P_i$  Process  $P_i$

$R_j$  Resource  $R_j$  with one instance (dot).

$P_k \rightarrow R_k$   $P_k$  requesting  $R_k$  (with two instances – two dots)

$P_l \leftarrow R_l$   $P_l$  holding  $R_l$

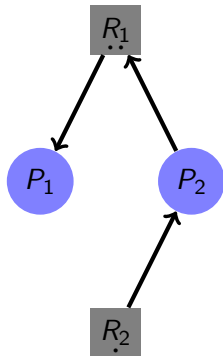
A process  $P_1$  is holding  $R_1$ , and hopefully it would finish, and simultaneously  $P_2$  can hold  $R_1$  (two instances) and  $R_2$  both and complete its tasks.



How to determine if deadlock exists?

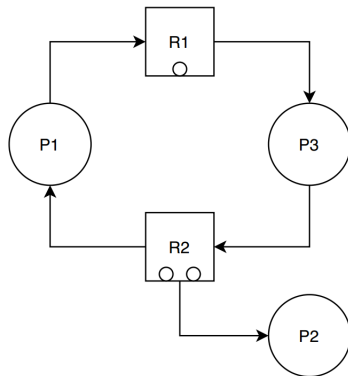
- No cycles  $\equiv$  No deadlock.
- If there are cycles:
  - If only one instance per resource type, then deadlock.
  - If several instances per resource type, possibility of deadlock.

A process  $P_1$  is holding  $R_1$ , and hopefully it would finish, and simultaneously  $P_2$  can hold  $R_1$  (two instances) and  $R_2$  both and complete its tasks.



# Example

Consider the scenario: there are three processes  $P_1$ ,  $P_2$  and  $P_3$ , and two resources  $R_1$  (with one instance) and  $R_2$  (with two instances).  $P_1$  has  $R_2$ , but waiting on  $R_1$  which is held by  $P_3$ .  $P_3$  is waiting on  $R_2$ , which is held by both  $P_1$  and  $P_2$ .

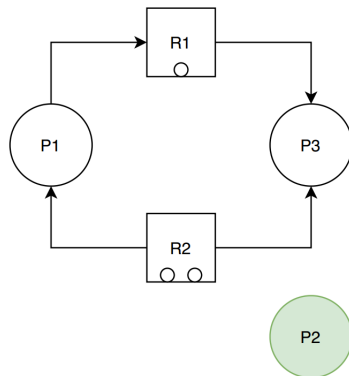


Does deadlock occur here?



# Example

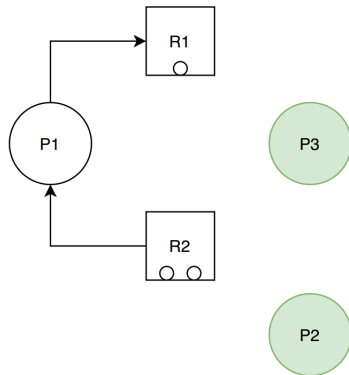
Consider the scenario: there are three processes  $P_1$ ,  $P_2$  and  $P_3$ , and two resources  $R_1$  (with one instance) and  $R_2$  (with two instances).  $P_1$  has  $R_2$ , but waiting on  $R_1$  which is held by  $P_3$ .  $P_3$  is waiting on  $R_2$ , which is held by both  $P_1$  and  $P_2$ .



$P_2$  completes its tasks and lets  $R_2$  go.  $P_3$  therefore gets the resource, and now can finish its tasks.

# Example

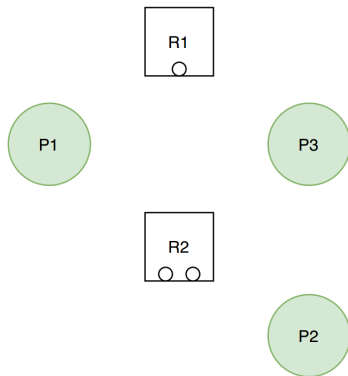
Consider the scenario: there are three processes  $P_1$ ,  $P_2$  and  $P_3$ , and two resources  $R_1$  (with one instance) and  $R_2$  (with two instances).  $P_1$  has  $R_2$ , but waiting on  $R_1$  which is held by  $P_3$ .  $P_3$  is waiting on  $R_2$ , which is held by both  $P_1$  and  $P_2$ .



$P_3$  completes its tasks and lets  $R_1$  go.  $P_1$  therefore has all the resources it needs, and can complete its tasks.

# Example

Consider the scenario: there are three processes  $P_1$ ,  $P_2$  and  $P_3$ , and two resources  $R_1$  (with one instance) and  $R_2$  (with two instances).  $P_1$  has  $R_2$ , but waiting on  $R_1$  which is held by  $P_3$ .  $P_3$  is waiting on  $R_2$ , which is held by both  $P_1$  and  $P_2$ .



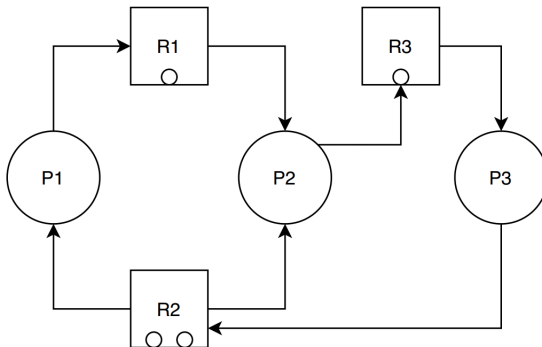
Each process finishes its tasks: no deadlock.

# Any questions?

---

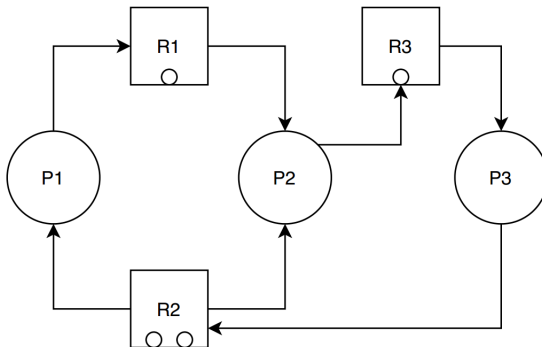


Is this system going to be deadlocked?



Please go to [www.menti.com](http://www.menti.com) and use the code 7232 1306

Is this system going to be deadlocked?



Please go to [www.menti.com](http://www.menti.com) and use the code 7232 1306

*P3* never gets *R2*, *P2* never gets *R3*, and *P1* never gets *R1*. The system is going to be deadlocked.

# Any questions?

---



Program or software optimisation is a process of modifying a software system to make some aspect of it to work more efficiently (i.e. executes more rapidly) or to use fewer resources (e.g. memory, energy, etc.). There is usually a trade-off between efficiency and resource usages.

Levels of optimisation: Design, algorithms and data structures, source code, build, compile, assembly, run-time, and platform-based.

*We should forget about small efficiencies, say about 97% of the time: premature optimization (prioritising performance over design) is the root of all evil. Yet we should not pass up our opportunities in that critical 3%*

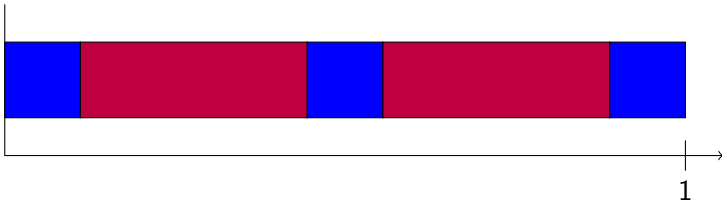
– Prof. Donald Knuth.



It takes effort and time to optimise a program. Often we can only spend finite time on a part of the program. How should we decide where to concentrate our efforts?



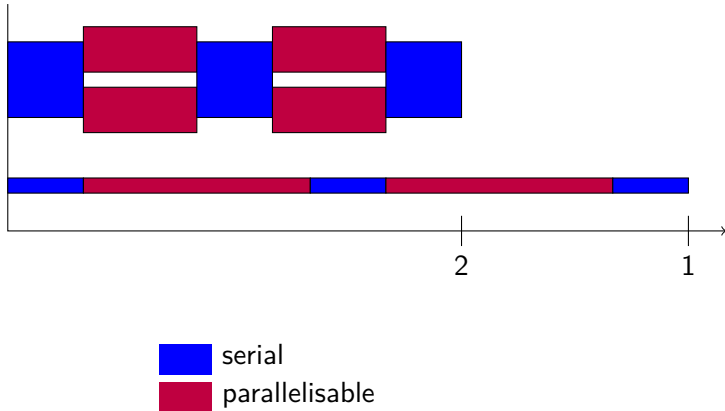
# Execution time: single processor

---

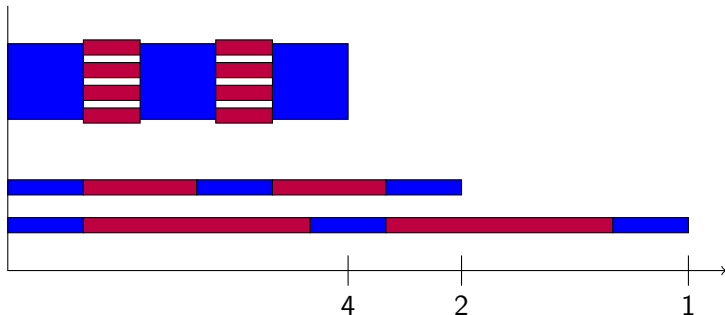


 serial  
 parallelisable

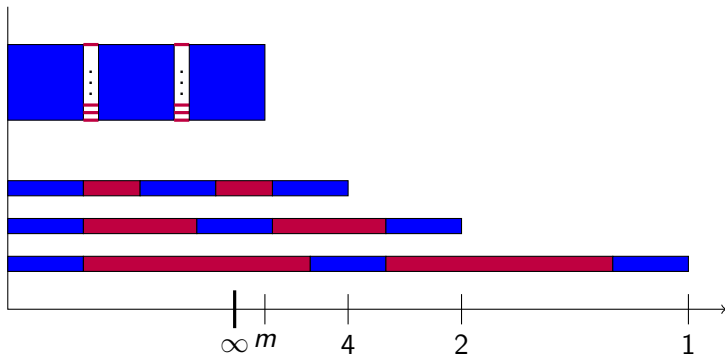
# Execution time: two processors



# Execution time: four processors



# Execution time: many processors



Optimising code, either through parallelisation or code improvement, may lead to improving overall execution time of a program. There is, however, a limit to how much performance gain is achievable. Amdahl's law helps us compute such theoretical limits for tasks with **fixed workload**.

$$L(k, n) \leq \frac{t_i}{t_o(k, n)},$$

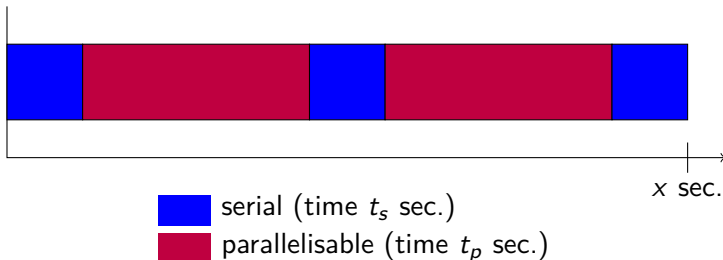
where,  $L$  = upper bound of speed up in latency (proportional improvement in execution time),

$k$  = Improvement factor in the sequential part,

$n$  = Improvement factor in the parallelisable part,

$t_o$  = Optimised execution time,

$t_i$  = Execution time before improvement  $t_i = t_o(1, 1)$ .



Initial total time for execution,  $t_i$   
= time for serial part + time for parallelisable part  
=  $t_s + t_p$

Therefore,  $t_p = t_i - t_s$ .

If we use,  $n$  cores for the parallelisable part, then the execution time will be:

$$t'_p = \frac{t_p}{n} = \frac{t_i - t_s}{n}.$$

Now  $k$  times improvement in the sequential part means, the sequential execution time will be:

$$t'_s = \frac{t_s}{k}.$$

Therefore, the combined execution time is:

$$\begin{aligned} t_0(k, n) &= t'_s + t'_p \\ &= \frac{t_s}{k} + \frac{t_p}{n} \end{aligned}$$

Now, the speed up bound is given by:

$$\begin{aligned} L(k, n) &\leq \frac{t_i}{t_o(k, n)} \\ &= \frac{t_s + t_p}{\frac{t_s}{k} + \frac{t_p}{n}} \\ &= \frac{1}{\frac{1}{k} \frac{t_s}{t_s + t_p} + \frac{1}{n} \frac{t_p}{t_s + t_p}} \quad \text{;divide both numerator} \\ &\quad \text{and denominator by } (t_s + t_p) \\ &= \frac{1}{\frac{1}{k} \tilde{t}_s + \frac{1}{n} \tilde{t}_p} \end{aligned}$$

Note that  $\tilde{t}_s = \frac{t_s}{t_s + t_p}$  is the proportion of the time spent in executing the sequential part, and  $\tilde{t}_p = \frac{t_p}{t_s + t_p}$  is the proportion of the time spent in executing the parallelisable part in the original program. Thus,  $\tilde{t}_p = 1 - \tilde{t}_s$ .



# Any questions?

---



A property is an attribute of a program that is true for every possible execution of that program.

Properties of interest for concurrent programs fall into two categories:

**Safety** property asserts that nothing bad (e.g. interference or deadlock) happens during execution. This must be true for all states.

**Liveness** property asserts that something good (e.g. a result, fairness and no restriction to progress) eventually happens. This is eventually true. We will generally concern ourselves with Liveness.

To check the safety property of a process  $P$ , we define a deterministic process property  $Q$ , and do the following:

- 1 Create a composite process  $R = (P \parallel Q)$ .
- 2 Check whether  $R$  has paths to an error state or not (much like testing).

# Car Park Example Revisited

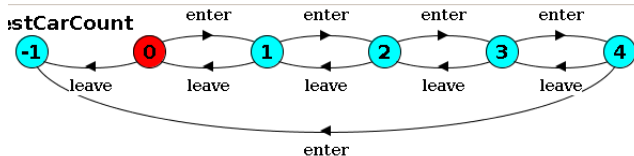
```
Entrance = (enter -> Entrance).  
Exit = (leave -> Exit).  
Controller(Capacity=4) = Spaces[Capacity],  
Spaces[spaceLeft:0..Capacity] =  
  (when spaceLeft>0 enter -> Spaces[spaceLeft-1]  
   |when spaceLeft<Capacity leave -> Spaces[spaceLeft+1]).  
||CarPark = (Entrance || Controller || Exit).  
// Property for checking the number of total cars  
property TotalCars = TotalCars[0],  
TotalCars[i:0..4] = (enter -> TotalCars[i+1] | leave ->  
TotalCars[i-1]).  
||TestCarCount = (CarPark || TotalCars).
```

In this case, there will be no errors, as we have used conditions to guard against these. What would happen if the guards were not there?

# Car Park Example Revisited

```

Entrance = (enter -> Entrance).
Exit = (leave -> Exit).
Controller(Capacity=4) = Spaces[Capacity],
Spaces[spaceLeft:0..Capacity] =
(when spaceLeft>0 enter -> Spaces[spaceLeft-1]
| when spaceLeft<Capacity leave -> Spaces[spaceLeft+1]).
||CarPark = (Entrance || Controller || Exit).
// Property for checking the number of total cars
property TotalCars = TotalCars[0],
TotalCars[i:0..4] = (enter -> TotalCars[i+1] | leave ->
TotalCars[i-1]).
||TestCarCount = (CarPark || TotalCars).
  
```



# Any questions?

---



- Resource Allocation Graphs are a neat tool to identify whether the system will approach deadlock dynamically.
- Safety and liveness are important properties.
- In FSP, you can use the `property` keyword to define a safety property that must hold for all states.
- Amdahl's law helps us quickly determine the potential performance gain for our optimisation efforts, and this is defined as the proportion between original program execution time and the execution time of the optimised program.