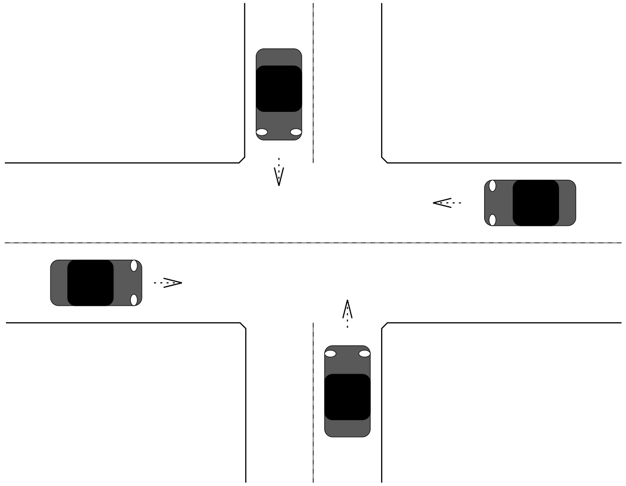
MIMD Programming problems

Interaction Of Processes

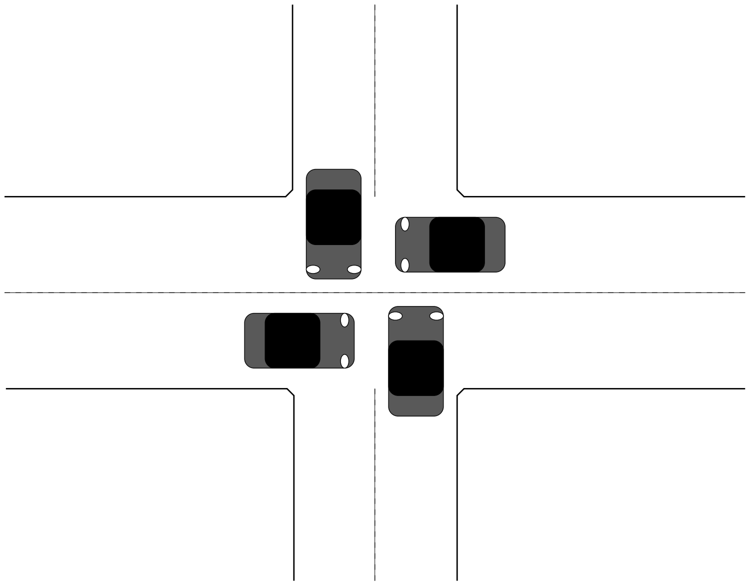
- A Simple Situation
  - A road junction in France



- A simple Set of Rules
  - Drive until you have to *Give Way* to traffic from the right.
  - Wait until the way is clear, then continue.

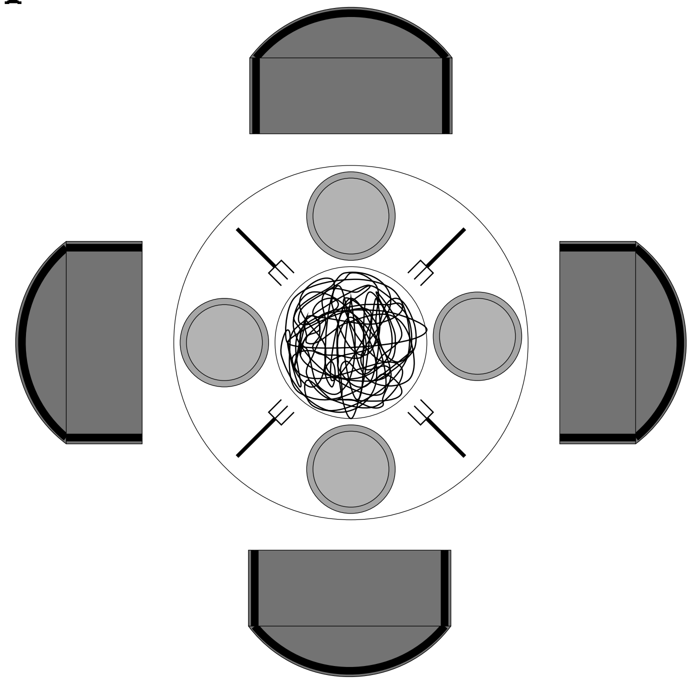
Deadlock

The state in which two or more processes are deferred indefinitely because each is awaiting another process to make progress, and no process is able to make progress.



- By some fluke all four cars have arrived at the junction together.
- We have deadlock.

Dining Philosophers



- One Table - One Bowl of Spaghetti.
- Four Philosophers - Four Chairs - Four Plates - Four Forks.

Dining Philosophers

- The Situation:
  - Philosophers Think & Eat.
  - Thinking and Eating are Exclusive Tasks.
- The Catch:
  - A Philosopher requires two forks in order to eat.
  - There are only four forks in all.
- The Problem:
  - We must write code to model the behaviour of one philosopher.
  - We will then examine the group behaviour.

OCCAM Processes

- An OCCAM program can be considered as hierarchy of processes.
- Most processes perform actions and then terminate.

Process Construction

```
SEQ
  Process_A
  Process_B
```

- This compound process is the sequence of the two processes `Process_A` and `Process_B`.
- `Process_A` is executed to termination before `Process_B` is begun.
- The compound process terminates when `Process_B` terminates.

OCCAM for Dining Philosophers

- Loop

```
WHILE condition
  Process_A
```

- This process executes `Process_A` repetitively while `condition` is true.

OCCAM for Dining Philosophers

- Choice

```
IF
  condition_a
    Process_A
  condition_b
    Process_B
```

- This process executes `Process_A` if `condition_a` is true.
- Else it executes `Process_B` if `condition_b` is true.
- Else it executes nothing at all and *doesn't terminate*.

OCCAM for Dining Philosophers

- Parallel Processes

```
PAR
  Process_A
  Process_B
```

- This compound process executes `Process_A` and `Process_B` in parallel.
- `Process_A` need not terminate before `Process_B` is begun.
- The compound process terminates when both `Process_A` and `Process_B` have terminated.

OCCAM for Dining Philosophers

- Declarations

```
INT i:
Process_A
```

- Declares i to be an integer within `Process_A`.

- Procedures

```
PROC fred()
  Process_B
:
Process_A
```

- Defines `fred()` to represent `Process_B` within `Process_A`.

Approach

- We will code the problem in OCCAM.
- A number of pre-defined functions are available for our use. Thus we do not have to worry about the intricacies of philosophical thought or the winding of spaghetti.
- We are not initially provided with a function allowing our philosophers to talk to each other.

Dining Philosophers

```
PROC Think()
  --- Think until hungry - unspecified duration.
:
PROC Eat()
  --- Eat until full - unspecified duration.
:
PROC Pick_Fork_If_Possible( FORK f )
  --- Pick up fork  f  if it is there.
:
BOOL FUNCTION Got_Fork( FORK f )
  --- Returns TRUE if fork  f  has been picked up.
:
PROC Pick_Fork_Always( FORK f )
    WHILE NOT Got_Fork( f )
      Pick_Fork_If_Possible( f )
:
```

Solution 1

Let us take the simple approach:

- Our philosopher will Think first.
- When hungry our philosopher will pick up the fork to his left and then the fork to his right.
- Our philosopher will then Eat.
- When full our philosopher will put down the fork to his right and then the fork to his left.

Dining Philosophers

Solution 1:

```
PROC Try_Eat()
  SEQ
    Pick_Fork_Always( left )
    Pick_Fork_Always( right )

    Eat()

    Put_Fork( right )
    Put_Fork( left )
:

WHILE TRUE
  SEQ
    Think()
    Try_Eat()
```

Group Behaviour

- Unfortunately by some fluke all the philosophers happen to finish thinking together.
- Each philosopher picks up the fork to his left.
- Each philosopher must wait for his right hand neighbour to finish eating.
- None of the philosophers can make progress.
- We have *deadlock*.

  The state in which two or more processes are deferred indefinitely because each is awaiting another process to make progress, and no process is able to make progress.

Solution 2

To prevent deadlock we must modify the behaviour of our philosopher:

- The deadlock arises because our philosopher stubbornly holds onto one fork while awaiting the other.
- If he *must wait* for a second fork, he should put down the first while he does so.
- Thus a waiting philosopher holds no forks. We can have no deadlock.

Dining Philosophers

Solution 2:

```
PROC Try_Eat()

  SEQ
    Pick_Fork_Always( left )
    Pick_Fork_If_Possible( right )

    WHILE NOT ( Got_Fork( left ) AND Got_Fork( right ))
      Swap_and_Retry()

    Eat()

    Put_Fork( right )
    Put_Fork( left )
:
```

Dining Philosophers

Where Swap_and_Retry() has been defined as:

```
PROC Swap_and_Retry()

  IF
    Got_Fork( left )
      SEQ
        Put_Fork( left )
        Pick_Fork_Always( right )
        Pick_Fork_If_Possible( left )

    Got_Fork( right )
      SEQ
        Put_Fork( right )
        Pick_Fork_Always( left )
        Pick_Fork_If_Possible( right )
:
```

Group Behaviour

- By fluke each philosopher picks up the fork from his left.
- No philosopher can pick up the fork on his right.
- All philosophers put down their left forks and pick up their right forks.
- No philosopher can now pick up the fork on his left.
- The process swaps and repeats.
- By some further fluke the philosophers remain synchronized.
- No food is consumed.
- We have *livelock*.

Livelock

- *A state in which the actions of two or more concurrently executing processes prevent computation from proceeding. No useful work is done by the interacting processes.*

- *The state may arise from a quirk of timing and may disappear for a similar reason. Unlike deadlock, livelock is not inherently stable.*

Solution 3

We shall try a different approach:

- Our problems are still caused by the state where the philosophers each have one fork.
- Let us assume that we can add another procedure to our library:

```
PROC Pick_Both_Forks_If_Possible()
  --- Pick up both forks if both are on the table.
:
```

- Are all our problems solved?

Solution 3:

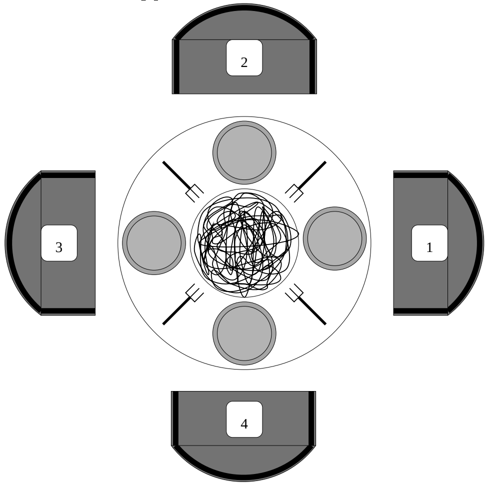
```
PROC Pick_Both_Forks_Always()
    WHILE NOT ( Got_Fork(left) AND Got_Fork(right))
      Pick_Both_Forks_If_Possible()
:

PROC Try_Eat()
  SEQ
    Pick_Both_Forks_Always()

    Eat()

    Put_Fork( right )
    Put_Fork( left )
:
```

Group Behaviour



- Let us assume that philosopher number 1 Eats while philosopher 3 Thinks and vice versa.
- Philosophers 2 and 4 will never see two available forks and will never Eat.
- We have *Indefinite Postponement*

Indefinite Postponement

- *A state in which the progress of one group of (one or more) processes is indefinitely postponed awaiting the release of resources by another group.*
- *The problem is essentially one of fairness in the allocation of resources.*
- *Like livelock, indefinite postponement is not inherently stable. It is possible for a timing quirk to return the system to normal operation.*

Deadlock, Livelock & Indefinite Postponement

- All of these problems are timing dependent.
- When we find our code behaving strangely we add extra debugging in order to track down the cause.
- The system timings are changed by this examination.
- Frequently we find that a problem disappears when we try to chase it.
- It is even possible for the this examination to expose new problems to confuse the issue further.

Programming with Concurrent Processes is Difficult.