Deadlock: four necessary and sufficient conditions

- Serially reusable resources:
  the processes involved share resources which they use under mutual exclusion.

- Incremental acquisition:
  processes hold on to resources already allocated to them while waiting to acquire additional resources.

- No pre-emption:
  once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.

- Wait-for cycle:
  a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.
6.1 Deadlock analysis - primitive processes

- deadlocked state is one with no outgoing transitions
- in FSP: STOP process

\[ \text{MOVE} = (\text{north-}>(\text{south-}>	ext{MOVE}|\text{north-}>	ext{STOP})) \]

- animation to produce a trace.
- analysis using LTSA: Trace to DEADLOCK: (shortest trace to STOP) north

Can we distinguish between a desirable termination and deadlock?

deadlock analysis - parallel composition

- in systems, deadlock may arise from the parallel composition of interacting processes.

\[ \text{RESOURCE} = (\text{get-}>	ext{put-}>	ext{RESOURCE}). \]

\[ \text{P} = (\text{printer.get-}>	ext{scanner.get-}>	ext{copy-}>	ext{printer.put-}>	ext{scanner.put-}>	ext{P}). \]

\[ \text{Q} = (\text{scanner.get-}>	ext{printer.get-}>	ext{copy-}>	ext{scanner.put-}>	ext{printer.put-}>	ext{Q}). \]

\[ \text{SYS} = (\text{p:} \text{P} || \text{q:} \text{Q} || \{\text{p,q}\}::\text{printer:RESOURCE} || \{\text{p,q}\}::\text{scanner:RESOURCE} ). \]

Deadlock Trace? Avoidance?

deadlock analysis - avoidance

- acquire resources in the same order?
- timeout:

\[ \begin{align*}
\text{P} & = (\text{printer.get-}>	ext{GETSCANNER}), \\
\text{GETSCANNER} & = (\text{scanner.get-}>	ext{copy-}>	ext{printer.put-}>	ext{scanner.put-}>	ext{P} \\
& \quad \quad \quad | \text{timeout} -> \text{printer.put-}>	ext{P} )). \\
\text{Q} & = (\text{scanner.get-}>	ext{GETPRINTER}), \\
\text{GETPRINTER} & = (\text{printer.get-}>	ext{copy-}>	ext{printer.put-}>	ext{scanner.put-}>	ext{Q} \\
& \quad \quad \quad | \text{timeout} -> \text{scanner.put-}>	ext{Q} ).
\end{align*} \]

6.2 Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.
Each FORK is a shared resource with actions get and put.

When hungry, each PHIL must first get his right and left forks before he can start eating.

Dining Philosophers - model structure diagram

Dining Philosophers - model

FORK = (get -> put -> FORK).
PHIL = (sitdown ->right.get->left.get ->eat ->right.put->left.put
->arise->PHIL).

Table of philosophers:

||DINERS(N=5)= forall [i:0..N-1]
{phil[i].left,phil[((i-1)+N)%N].right}::FORK.

Can this system deadlock?

Dining Philosophers - model analysis

Trace to DEADLOCK:
phil.0.sitdown
phil.0.right.get
phil.1.sitdown
phil.1.right.get
phil.2.sitdown
phil.2.right.get
phil.3.sitdown
phil.3.right.get
phil.4.sitdown
phil.4.right.get

This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his right. The system can make no further progress since each philosopher is waiting for a fork held by his neighbour i.e. a wait-for cycle exists!

Dining Philosophers

Deadlock is easily detected in our model.

How easy is it to detect a potential deadlock in an implementation?
Dining Philosophers - implementation in Java

```
class Fork {
    private boolean taken=false;
    private PhilCanvas display;
    private int identity;
    Fork(PhilCanvas disp, int id)
        display = disp; identity = id;
    synchronized void put() {
        taken=false;
        display.setFork(identity,taken);
        notify();
    }
    synchronized void get()
        throws java.lang.InterruptedException {
            while (taken) wait();
            taken=true;
            display.setFork(identity,taken);
        }
}
```

taken encodes the state of the fork.

Dining Philosophers - Fork monitor

```
class Philosopher extends Thread {
    public void run() {
        try {
            while (true) {
                view.setPhil(identity,view.THINKING);
                sleep(controller.sleepTime()); // hungry
                view.setPhil(identity,view.HUNGRY);
                right.get(); // got right chopstick
                sleep(500);
                left.get(); // eating
                view.setPhil(identity,view.EATING);
                sleep(controller.eatTime());
                right.put();
                left.put();
            }
        } catch (java.lang.Interrupted Exception e){}
    }
}
```

Follows from the model (sitting down and leaving the table have been omitted).

Dining Philosophers - implementation in Java

```
for (int i =0; i<N; ++i)
    fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i)
    phil[i] =
        new Philosopher
            (this,i,fork[(i-1+N)%N],fork[i]);
    phil[i].start();
```

Code to create the philosopher threads and fork monitors:
Dining Philosophers

To ensure deadlock occurs eventually, the slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating. This "speedup" increases the probability of deadlock occurring.

Deadlock-free Philosophers

Deadlock can be avoided by ensuring that a wait-for cycle cannot exist. How?

Introduce an asymmetry into our definition of philosophers. Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

Other strategies?

Maze example - shortest path to "deadlock"

We can exploit the shortest path trace produced by the deadlock detection mechanism of LTS to find the shortest path out of a maze to the STOP process!

We first model the MAZE.

Each position is modelled by the moves that it permits. The MAZE parameter gives the starting position.

eg. \( \text{MAZE}(\text{Start}=8) = \text{P}[\text{Start}], \) \( \text{P}[0] = (\text{north} \rightarrow \text{STOP} | \text{east} \rightarrow \text{P}[1]), \ldots \)
Summary

Concepts
- **deadlock**: no further progress
- four necessary and sufficient conditions:
  - serially reusable resources
  - incremental acquisition
  - no preemption
  - wait-for cycle

Models
- no eligible actions (analysis gives shortest path trace)

Practice
- blocked threads

Aim: deadlock avoidance - to design systems where deadlock cannot occur.