

DEADLOCK

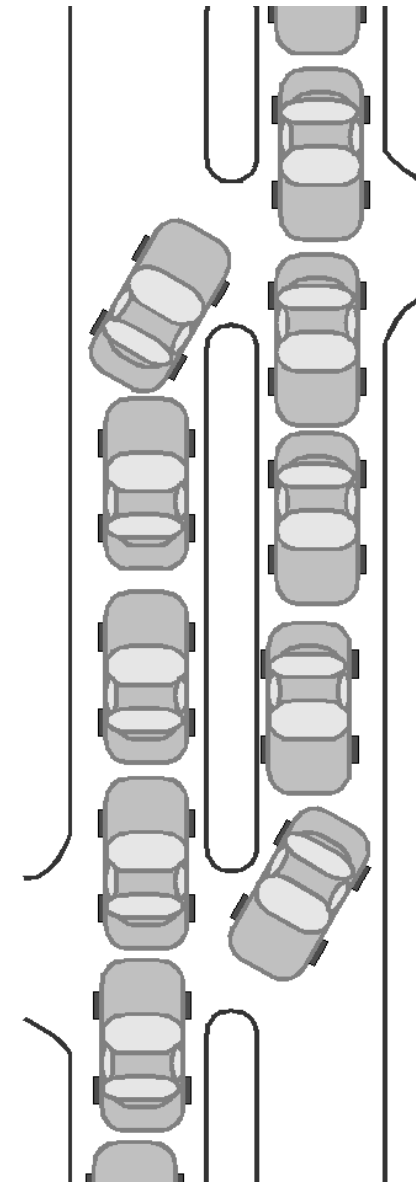
In previous lecture notes we looked at the problem of allowing multiple processes to perform operations on shared data. Each of the solutions – flag variables, semaphores, monitors, transactions – share the key idea of a *lock*.

In this context, lock is something that is *requested* by a process that wants to access some shared data, and *held* by a single process while the data is being accessed.

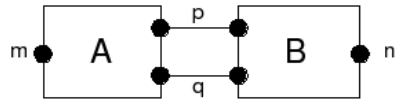
Locks can be used to enforce correct manipulation of shared data: but in a concurrent system with multiple processes and multiple locks, they can create problems.

Any concurrent system in which processes may simultaneously hold one or more locks while requesting *other* locks is potentially prone to *deadlock*.

Deadlock is not confined to computer systems: it can occur whenever there is a cycle of resources that are simultaneously *held* and *requested*.



XCIRCAL DEADLOCK EXAMPLE



Process A, A1, B, B1

Event m, n, p, q

A \leftarrow m A1

A1 \leftarrow p q A

B \leftarrow n B1

B1 \leftarrow q p B

display $\sim(A*B)$

Which results in:

S0 == ((n S1 + (m n) S2) + m S3)

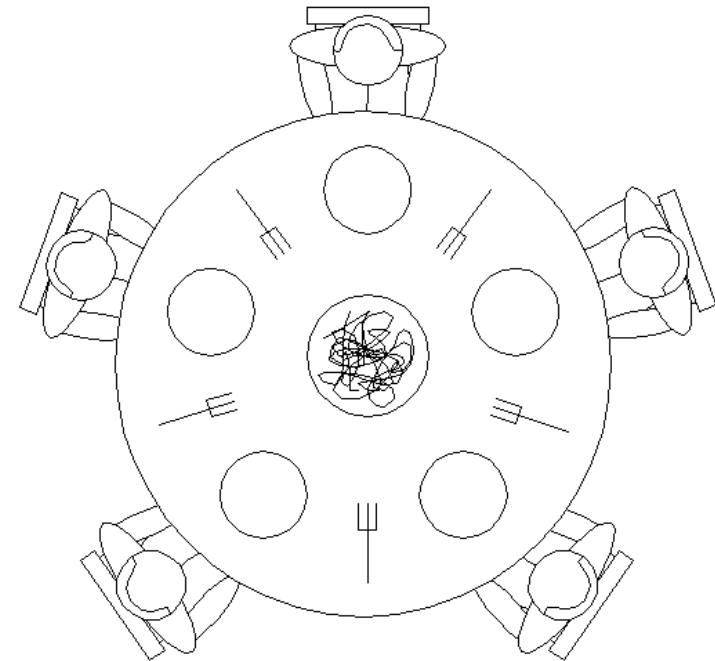
S1 == m S2

S2 == /\

S3 == n S2

THE DINING PHILOSOPHERS PROBLEM

The so called “Dining Philosophers” problem is a piece of computer science folklore (used by Edsger Dijkstra in 1971). It illustrates classic issues of concurrency and contention for shared resources.



A number of philosophers sit around a circular table. The philosophers only do two things: they alternately *think* and *eat*. Between each philosopher is a fork. Because of the tangled supply of noodles in the middle of the table, each philosopher requires two forks in order to eat. Each philosopher thus executes the following process:

Think
Obtain forks
Eat
Release forks
Think
...

The interesting part is the protocol whereby philosophers obtain and release their forks.

Compare this system to the mutual exclusion example in Lecture Notes 4. There, we had 2 worker processes contending for 1 shared resource (the right to enter a critical section). Here, we have N processes worker processes (the philosophers), contending for N shared resources (the forks).

CIRCAL MODEL OF THE FORKS

The various processes are linked (and thus *communicate*) in a pattern. Philosopher I requires forks (I) and (I+1), while fork (I) is shared between philosopher (I-1) and philosopher (I).

This system is more complex, but it does have a regular pattern. We can use XCircal to automatically construct all the processes for a problem of size N and link them correctly together.

First let us model a single fork. It can be picked up, then dropped, from either the left or right.

```
Frk <- lpt lpd Frk + rpt rpd Frk
```

where the actions have the following meanings:

lpt	“left philosopher takes” The philosopher to the <i>fork’s left</i> picks up the fork
lpd	“left philosopher drops” The philosopher to the <i>fork’s left</i> drops the fork
rpt	“right philosopher takes” The philosopher to the <i>fork’s right</i> picks up the fork
rpd	“right philosopher drops” The philosopher to the <i>fork’s right</i> drops the fork

XCIRCAL MODEL OF THE FORKS

Now, each of our forks has the same behaviour, but with different events.

We can write a `XCircal` function that (a) defines a *prototype fork* and (b) makes a copy of this prototype with 4 specified event names.

```
Process Fork(Event leftPhilosopherTakes,  
             leftPhilosopherDrops,  
             rightPhilosopherTakes,  
             rightPhilosopherDrops) {  
  static Event lpt, lpd, rpt, rpd  
  static Process Frk {  
    Frk <- lpt lpd Frk + rpt rpd Frk  
  }  
  return Frk[leftPhilosopherTakes/lpt,  
             leftPhilosopherDrops/lpd,  
             rightPhilosopherTakes/rpt,  
             rightPhilosopherDrops/rpd]  
}
```

This “process factory” function is explained as follows.

```
Process Fork(Event leftPhilosopherTakes,  
             leftPhilosopherDrops,  
             rightPhilosopherTakes,  
             rightPhilosopherDrops) {
```

`XCircal` uses a C-like function definitions. Here, a function called `Fork` is defined, which takes four *event names* as arguments, and returns a `Process` object.

```
static Event lpt, lpd, rpt, rpd  
static Process Frk {  
  Frk <- lpt lpd Frk + rpt rpd Frk  
}
```

The `static` keyword inside a function body works has a similar meaning as in C: it declares that only one copy of an object is to be created and it is to be shared in all invocations of the function.

These lines define the prototype fork (called `Frk`), with its own set of event names (`lpt`, `lpd`, `rpt`, `rpd`).

```
return Frk[leftPhilosopherTakes/lpt,  
           leftPhilosopherDrops/lpd,  
           rightPhilosopherTakes/rpt,  
           rightPhilosopherDrops/rpd]  
}
```

When we call our fork-constructing function, we want a new process that has the same behaviour as the prototype fork, but with action names that we have supplied instead of the prototype action names.

This is done with the renaming operator: we replace each of the prototype event names with the corresponding event name from the function argument list.

We can then use this function to create `N` fork processes.

```
Process F[n]  
Event lpt[n], lpd[n], rpt[n], rpd[n]  
  
for(i=0;i<n;i++) // instantiate n forks  
  F[i] <- Fork(lpt[i],lpd[i],rpt[i],rpd[i])
```

You may notice that each fork acts like a semaphore: the ‘grab’ events act as the semaphore ‘wait’ operation and then ‘free’ events act as the semaphore ‘signal’ operation. This suggests the following protocol for the philosopher.

- 1) Wait for left fork to become free and pick it up
- 2) Wait for right fork to become free and pick it up
- 3) Eat
- 4) Put down left fork
- 5) Put down right fork

In XCircal, a prototypical philosopher can thus be written:

```
Ph <- think tlf trf eat dlf drf Ph
```

where the actions have the following meanings:

tlf	“take left fork” The philosopher picks up <i>his left</i> fork.
trf	“take right fork” The philosopher picks up <i>his right</i> fork
dlf	“drop left fork” The philosopher drops <i>his left</i> fork
drf	“drop right fork” The philosopher drops <i>his right</i> fork

We will study this model of the dining philosophers problem in the second lab exercise.