Applying ...

Peter Welch (p.h.welch@kent.ac.uk)
Computing Laboratory, University of Kent at Canterbury

Co631 (Concurrency)

Applying ...

The dining philosophers ...

Compiling …

Real-time inference engine …

Fast fourier transform …

Computing on global data …

Neural nets …

Microprocessor design …

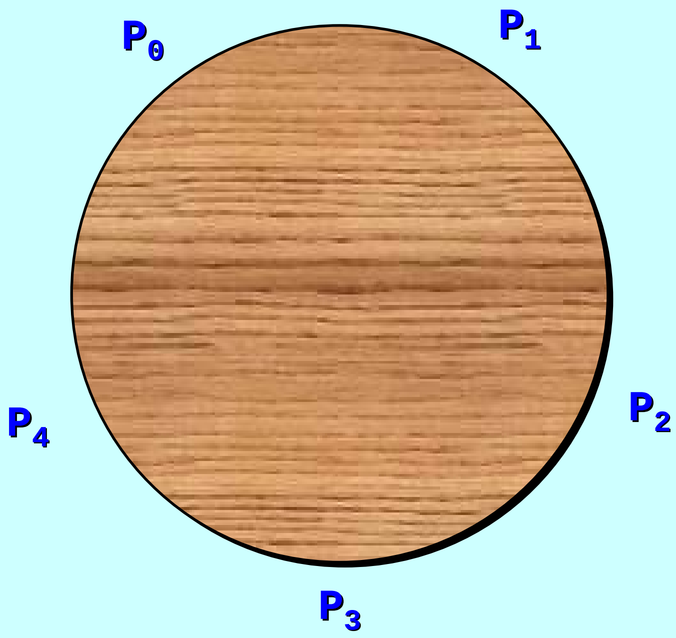
Autonomous robots ...

The Dining Philosophers

Once upon a time, five philosophers lived in the same college. They were proud, independent philosophers who thought independent thoughts and never communicated with each other (or with anyone else, for that matter) what these thoughts might have been.

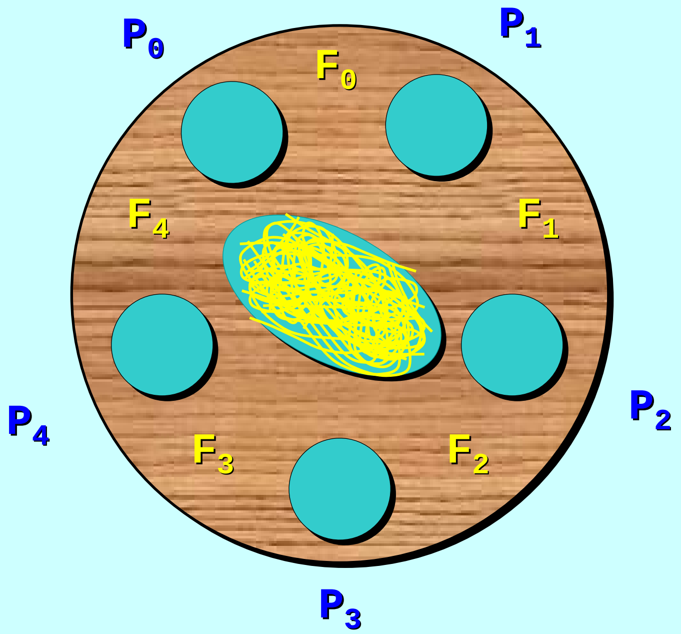
From time to time, each philosopher would get hungry. At such times, she (or he) would stop thinking and go to the single dining room in the college – shared by all the philosophers.

The Dining Philosophers



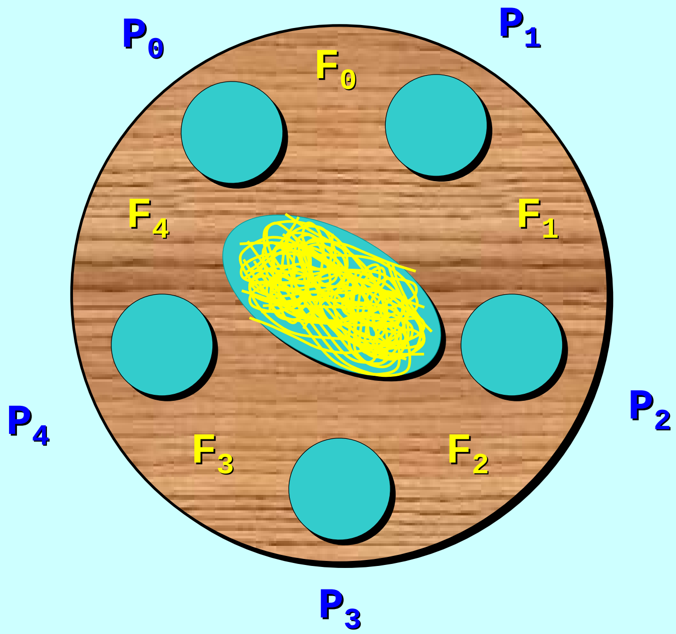
The dining room contained one circular table, around which were symmetrically placed five chairs. Each chair was labelled with the name of one of the philosophers and each philosopher was only allowed to sit in her/his own chair.

The Dining Philosophers



Opposite each chair was a plate and, between the plates, was laid a single golden fork. In the centre of the table was a large bowl of spaghetti, which was constantly replenished.

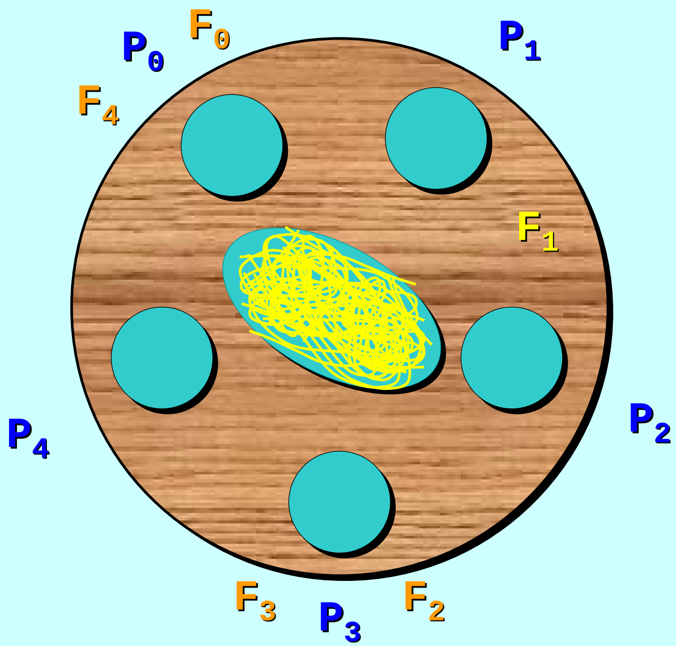
The Dining Philosophers



The philosophers never managed to master the art of serving, or indeed, eating the spaghetti with a single fork.

To eat, they had to pick up *two* forks – one from each side of their plates.

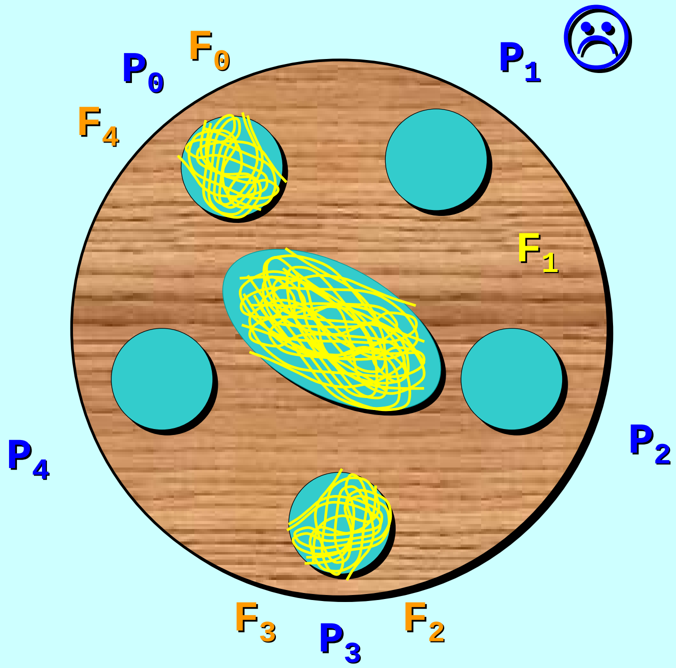
The Dining Philosophers



The philosophers never managed to master the art of serving, or indeed, eating the spaghetti with a single fork.

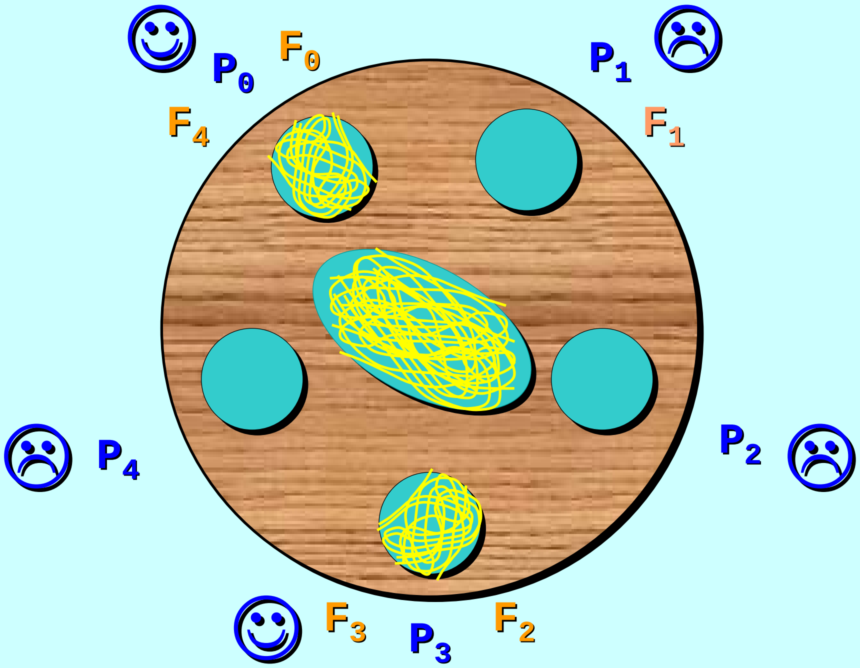
To eat, they had to pick up *two* forks – one from each side of their plates.

The Dining Philosophers



If a fork was being used by a neighbouring philosopher, a hungry philosopher anxiously waited for the neighbour to finish eating.

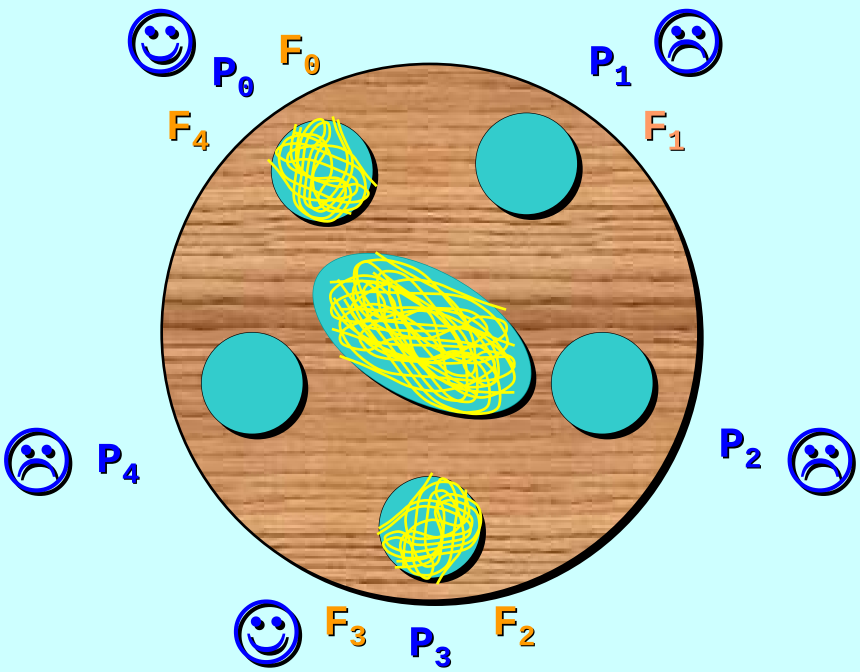
The Dining Philosophers



If a fork was being used by a neighbouring philosopher, a hungry philosopher anxiously waited for the neighbour to finish eating.

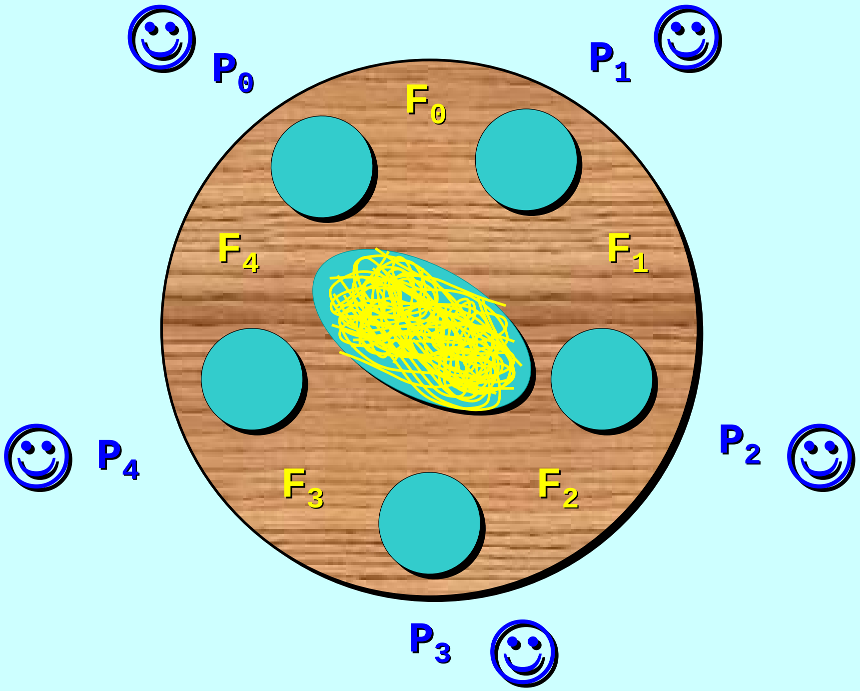
This was the only occasion when the existence of one philosopher had an impact on the life of another ...

The Dining Philosophers



The philosophers lived like this for years and years until, one day, something most unfortunate happened.

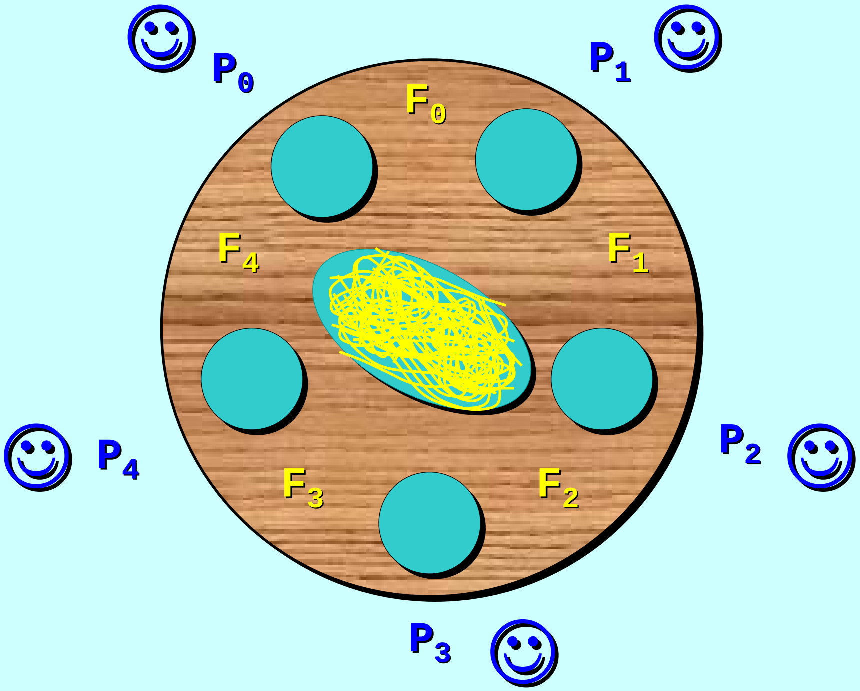
The Dining Philosophers



The philosophers lived like this for years and years until, one day, something most unfortunate happened.

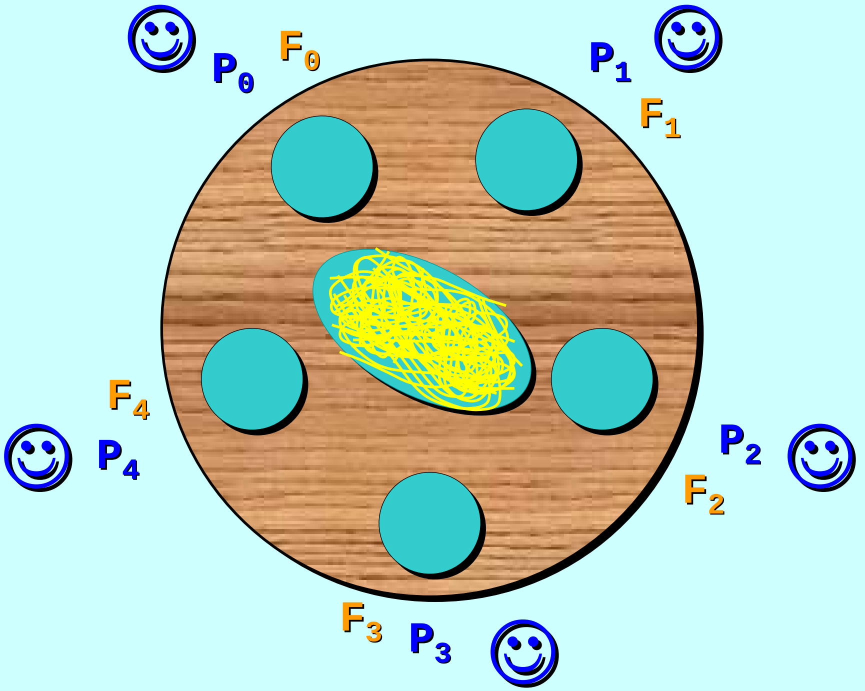
By chance, all the philosophers got hungry at the same time, went to the dining room, sat down and reached for the forks.

The Dining Philosophers



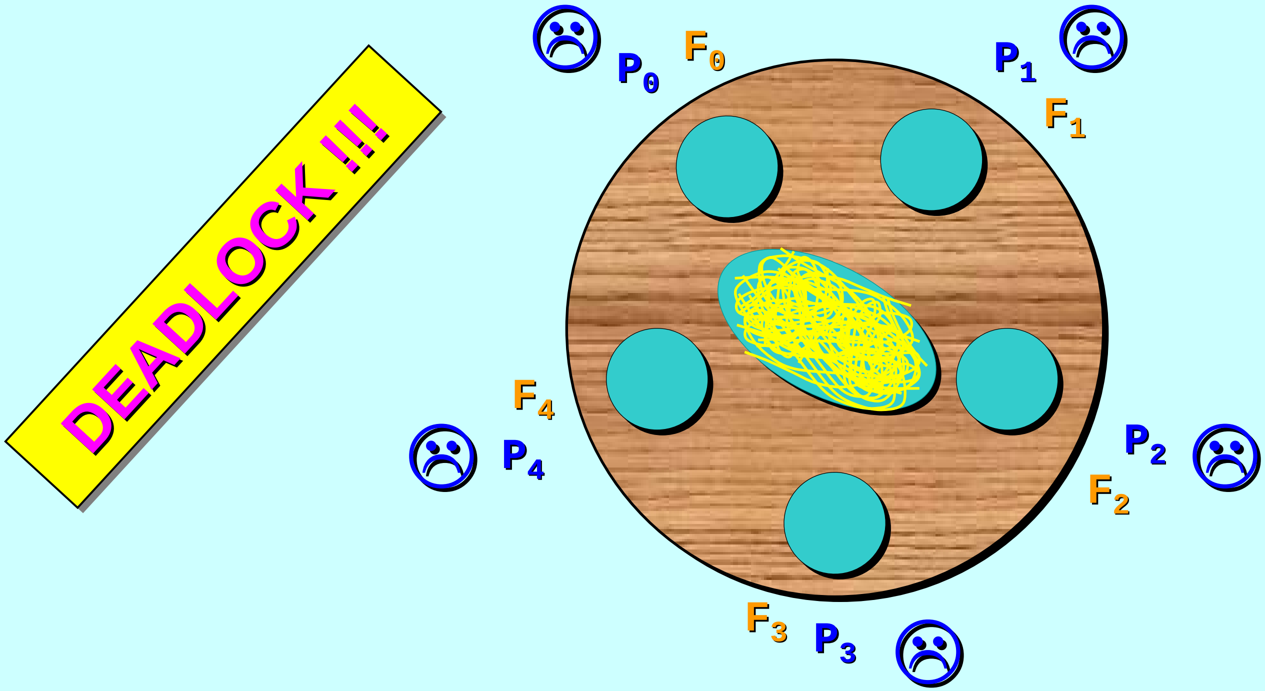
By further chance, each philosopher picked up the fork on her/his left.

The Dining Philosophers

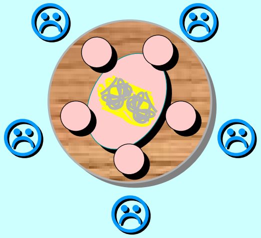


By further chance, each philosopher picked up the fork on her/his left. Noticing that the other fork was being used, all philosophers waited for their neighbours to finish .... and waited ... and waited ... and waited ....

The Dining Philosophers



By further chance, each philosopher picked up the fork on her/his left. Noticing that the other fork was being used, all philosophers waited for their neighbours to finish .... and waited ... and waited ... and waited .... and starved to death!



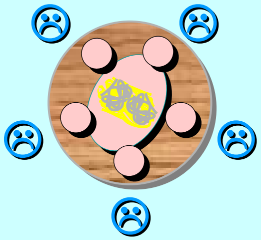
The Dining Philosophers

The story of *The Dining Philosophers* is due to Edsger Dijkstra – one of the founding fathers of Computer Science.

It illustrates a classic problem in concurrency: how to share resources safely between competing consumers.

http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF

Historical document



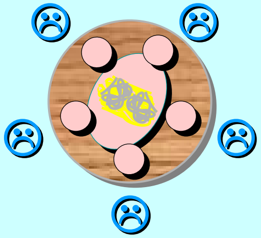
The Dining Philosophers

In this example, the resources are the forks and the consumers are the philosophers.

Problems arise because of the limited nature of the resources (only 5 forks) and because each consumer (5 of them) needs 2 forks at a time.

The spaghetti is an infinite resource in this tale – so, plays no role in the catastrophe.

Similarly, the college provides exclusive facilities for *thinking* (rooms) and *eating* (chairs and plates) for each philosopher – so, these also play no role.



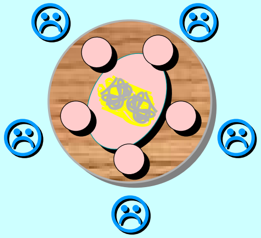
The Dining Philosophers

The source of the story was a deadlock that would mysteriously arise from time to time in an early multiprocessing operating system.

The philosophers are user processes that need file I/O.

To read or write a file, a process has to acquire a data buffer (to smooth data transfer and make it fast). If 2 files need to be open at the same time, 2 buffers are needed.

In those days, memory was scarce – so the number of buffers was limited. The forks are the buffers.

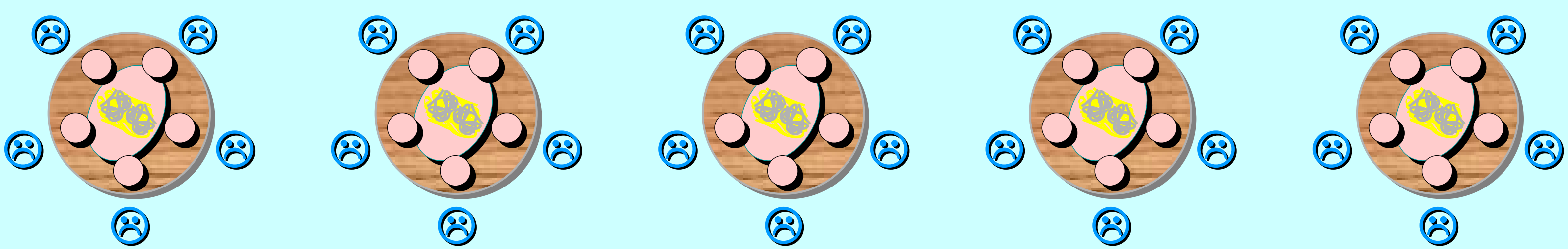


The Dining Philosophers

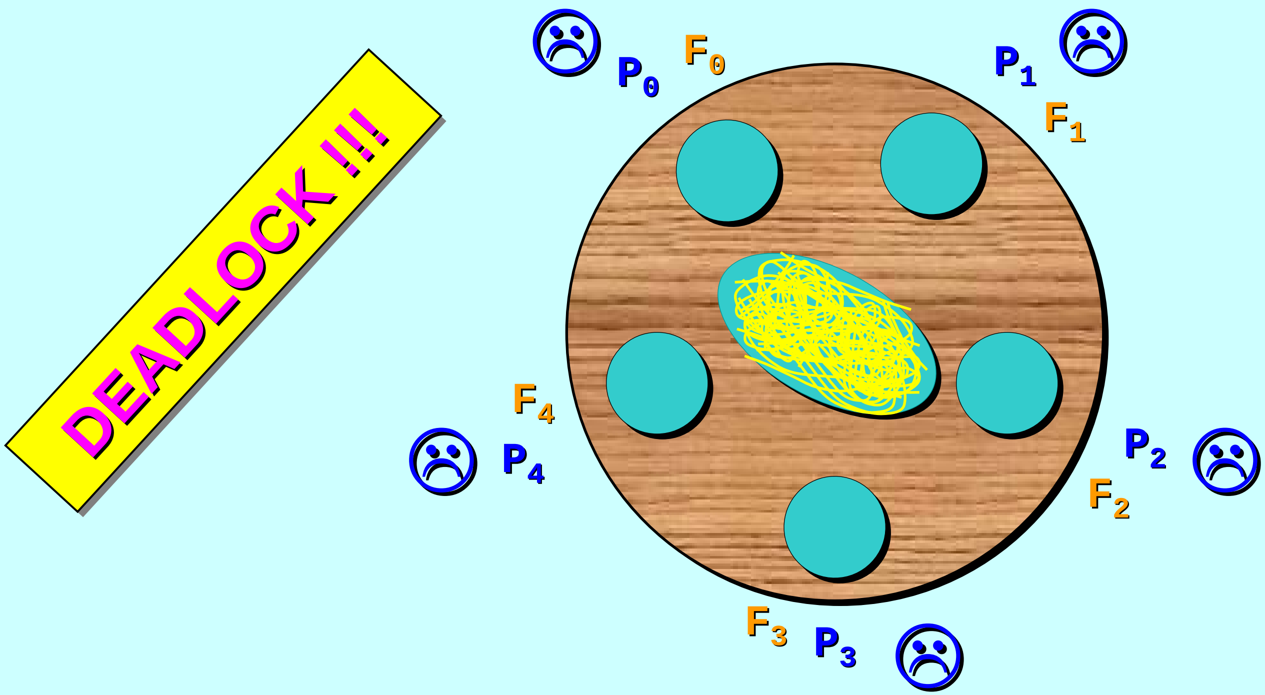
Today – some 34 years later – memory is not so scarce!

Yet, operating system (or specific application) deadlock is rampant. How often does your whole PC (or one of its applications) lock up on you?

We have been, and still are, making the same mistakes again and again and again ...



The Dining Philosophers



We'll modify the system to eliminate deadlock presently. First, let's model the system as it stands. Then, we can start reasoning about it!

As discussed, the only significant players are the forks and the philosophers. We'll start with the philosophers.


```
PROC philosopher (CHAN BOOL left!, right!)






  :
```

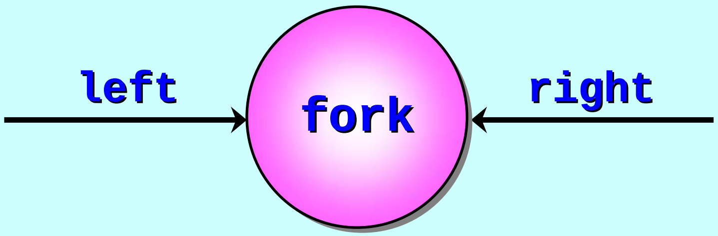
This philosopher's only point of contact with the rest of the world is when picking up the forks …


```
PROC philosopher (CHAN BOOL left!, right!)
  WHILE TRUE
    SEQ
      ...  think
      PAR                        -- pick up forks
        left ! TRUE
        right ! TRUE
      ...  eat
      PAR                        -- put down forks
        left ! TRUE
        right ! TRUE
:
```

This philosopher's only point of contact with the rest of the world is when picking up the forks … the philosopher will be blocked if one or both are not there …

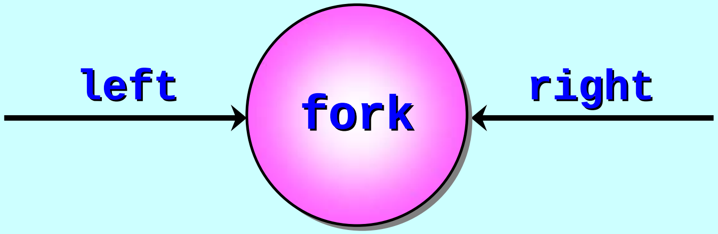

```
PROC philosopher (CHAN BOOL left!, right!)
  WHILE TRUE
    SEQ
      ...  think
      PAR                        -- pick up forks
        left ! TRUE
        right ! TRUE
      ...  eat
      PAR                        -- put down forks
        left ! TRUE
        right ! TRUE
:
```

This philosopher's only point of contact with the rest of the world is when picking up the forks … the philosopher will never be blocked putting down the forks …



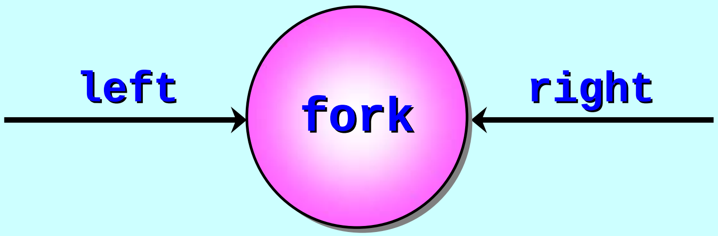

```
PROC fork (CHAN BOOL left?, right?)
  WHILE TRUE
    BOOL any:
    ALT
      left ? any              -- left phil picks up
        left ? any            -- left phil puts down
      right ? any             -- right phil picks up
        right ? any           -- right phil puts down
:
```

Once a fork has been *picked up* by a philosopher (say the one on its left), it waits to be *put down* by that philosopher (the one on its left). While it is being held by one, it cannot be *picked up* by another.




```
PROC fork (CHAN BOOL left?, right?)
  WHILE TRUE
    BOOL any:
    ALT
      left ? any                -- left phil picks up
        left ? any              -- left phil puts down
      right ? any               -- right phil picks up
        right ? any             -- right phil puts down
:
```

Once a fork has been *picked up* by a philosopher (say the one on its left), it waits to be *put down* by that philosopher (the one on its left). While it is being held by one, it cannot be *picked up* by another.




```
PROC fork (CHAN BOOL left?, right?)
  WHILE TRUE
    BOOL any:
    ALT
      left ? any                -- left phil picks up
        left ? any              -- left phil puts down
      right ? any               -- right phil picks up
        right ? any             -- right phil puts down
:
```

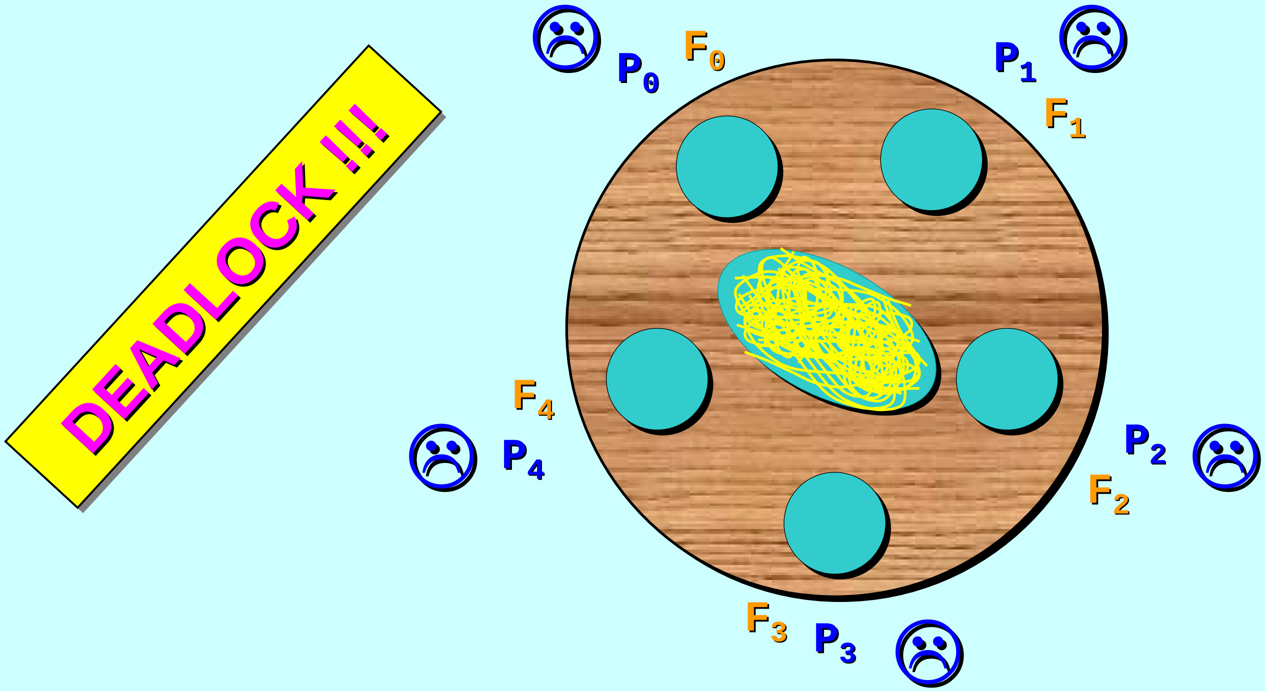
Note: this **fork** process provides a *mutual exclusion* lock – commonly known as a '*mutex*'. If two processes must not engage in a particular activity at the same time, program them to acquire a *mutex* first ... and release it afterwards*.

* NB: there are other ways ...


```
PROC philosopher (CHAN BOOL left!, right!)
  WHILE TRUE
    SEQ
      ...  think
      PAR                          -- pick up forks
        left ! TRUE
        right ! TRUE
      ...  eat
      PAR                          -- put down forks
        left ! TRUE
        right ! TRUE
:
```

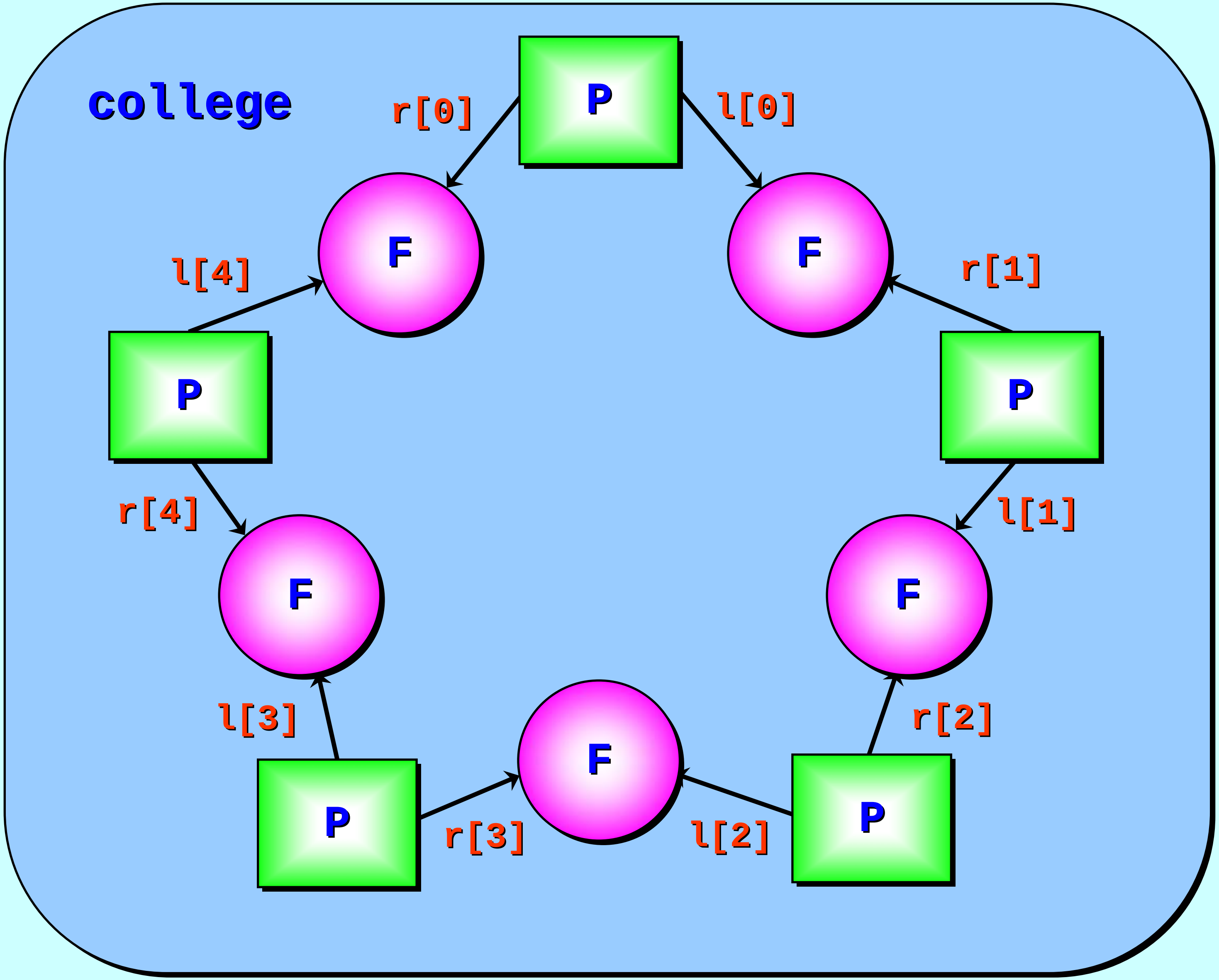
For our **philosopher** processes, the competitive activity is using a particular **fork**. Only one may use it at a time ... Hence, it must acquire the **fork** before eating ... and release it afterwards.

The Dining Philosophers

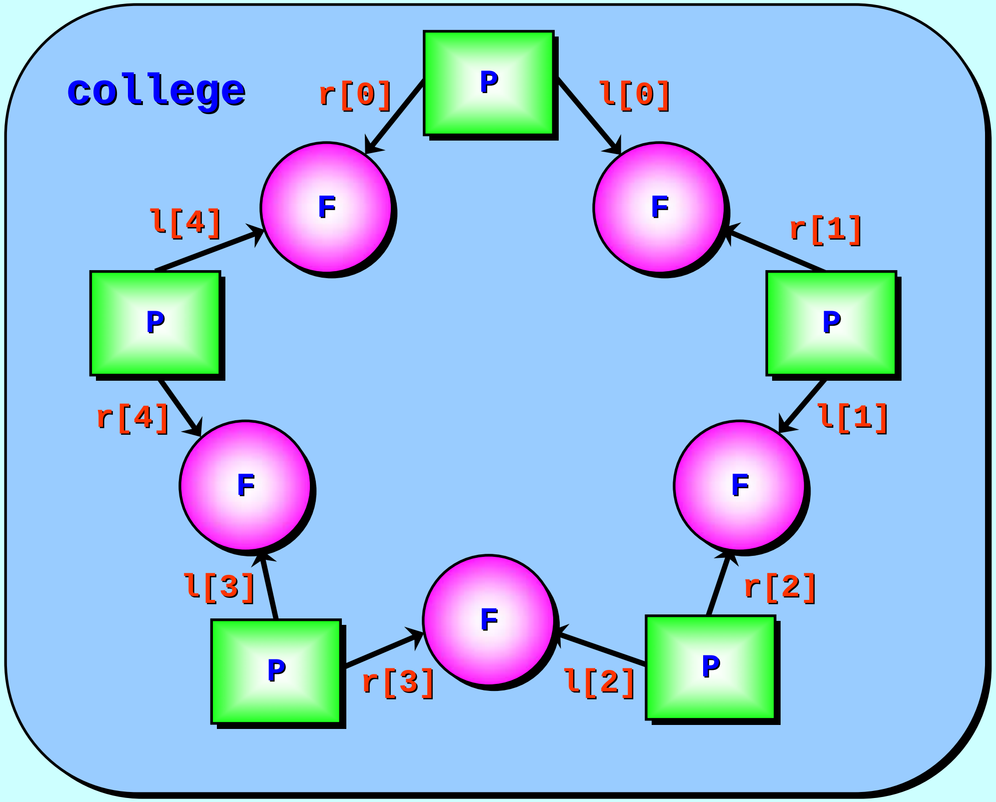


Now, let's build the **college** system ... which is simply all the philosophers and all the forks ... connected together correctly.

The **college** is just a process. It is a closed system, currently, with no connections to the outside world.

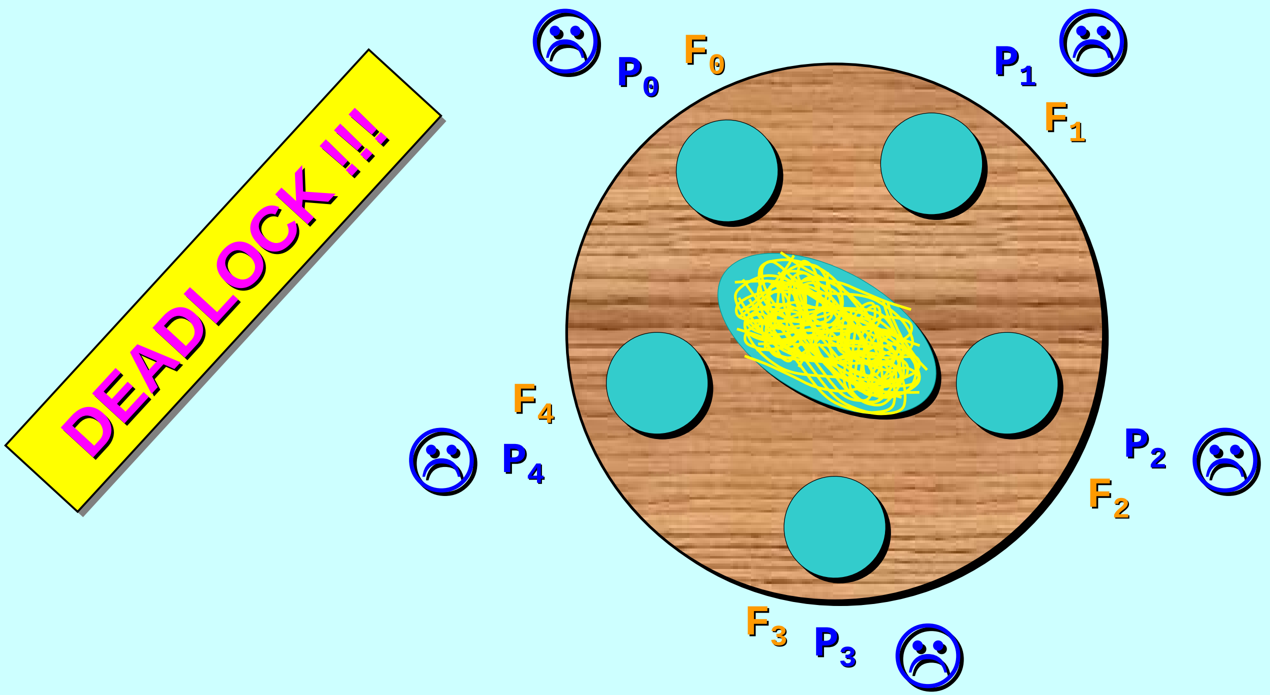


"**l**" = "**left**", "**r**" = "**right**" channels (from the philosophers' points of view)




```
PROC college ()
  [5]CHAN BOOL left, right:
  PAR i = 0 FOR 5
    PAR
      philosopher (left[i]!, right[i]!)
      fork (left[i]?, right[(i+1)\5]?)
:
```

The Dining Philosophers



Now, let's eliminate the potential for deadlock in this system ... and prove it!

Ways to avoid this deadlock ...

- **Buy one extra fork:**

  Asymmetric solution. Also, the philosophers are very jealous and would not tolerate one of their number having more resources (an extra fork) than the others!

- **Buy five extra forks:**

  Too expensive!! The college is suffering from government cut-backs and the forks are made of gold.

- **One of the philosophers picks up the right fork first:**

  Asymmetric solution. Each philosopher would need to be told whether to go for the right or left first. Also, it forces the fork pick-ups to be done in sequence. Philosophers have two hands and want to use them in parallel.

Ways to avoid this deadlock ...

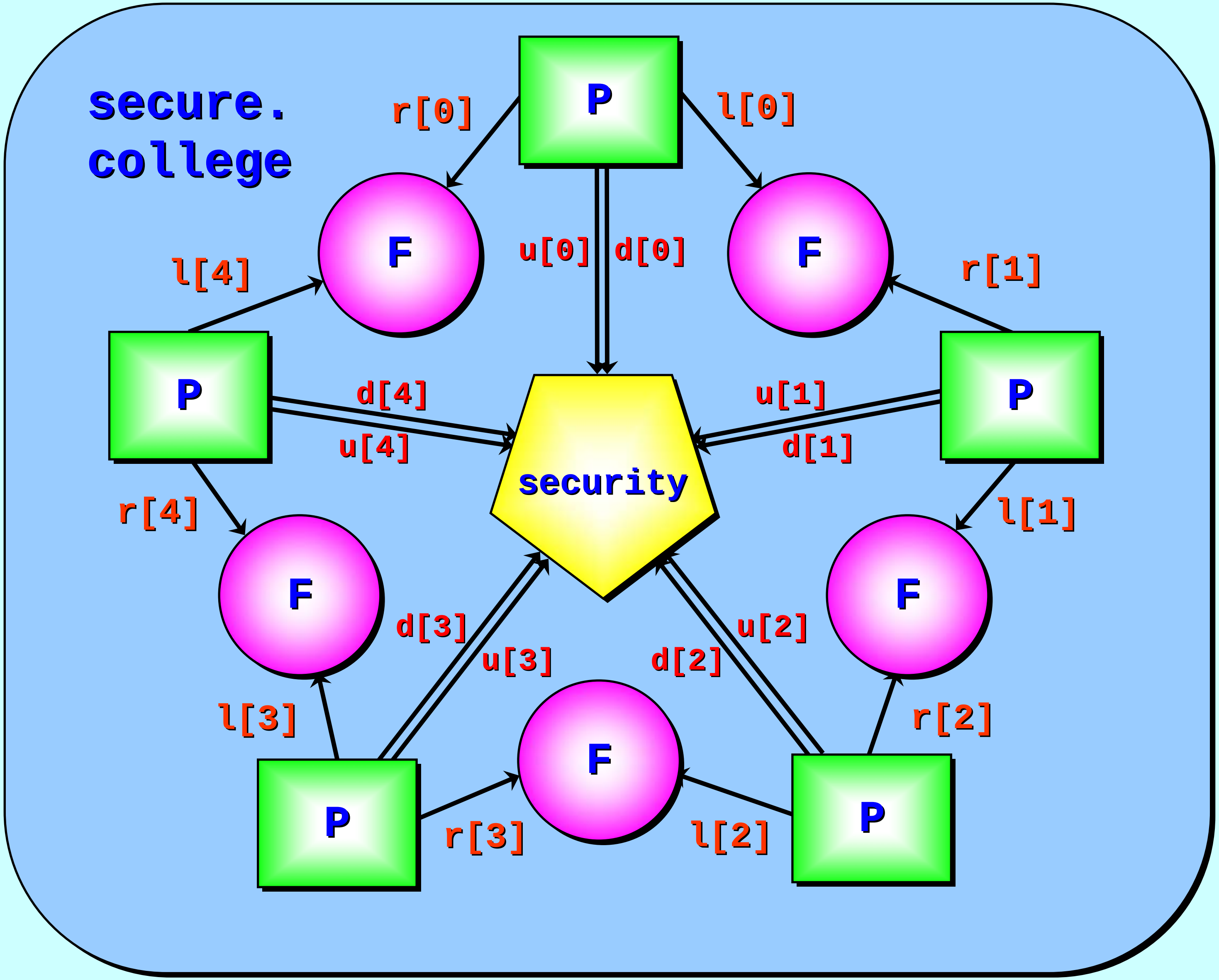
- **External authority:**

  College hires a `security` guard to whom each philosopher has to report when she wants to *sit down* at or *stand up* from the table.

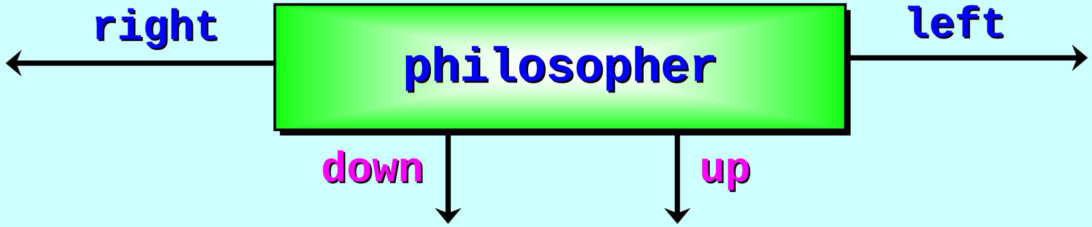
  The `security` guard has instructions *not to allow more than four philosophers at a time to sit down*.

  This solution is *symmetric* (the philosophers still have equal, though reduced, rights), *does not reduce concurrency* (in the fork pick-ups) and is *cheap* (salaries are peanuts compared with the cost of extra forks).

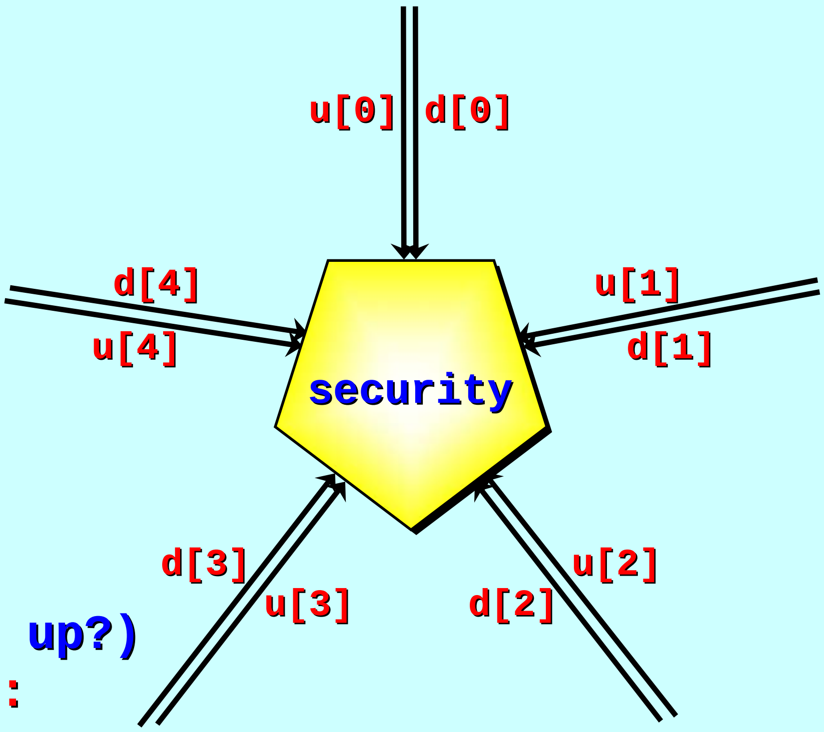
We'll go for this one ...



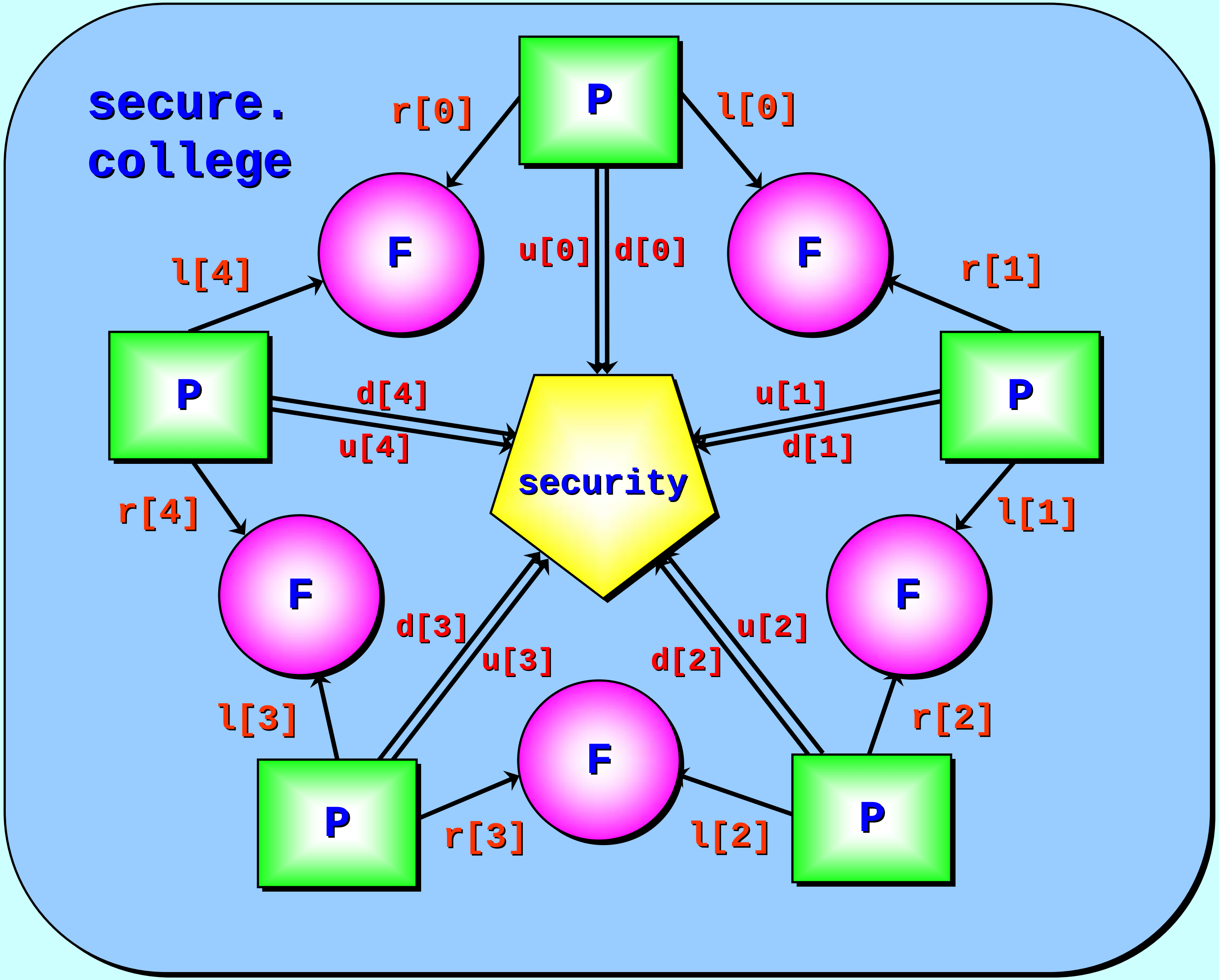
"**d**" = "**down**", "**u**" = "**up**" channels (for indicating wish to sit down or stand up)



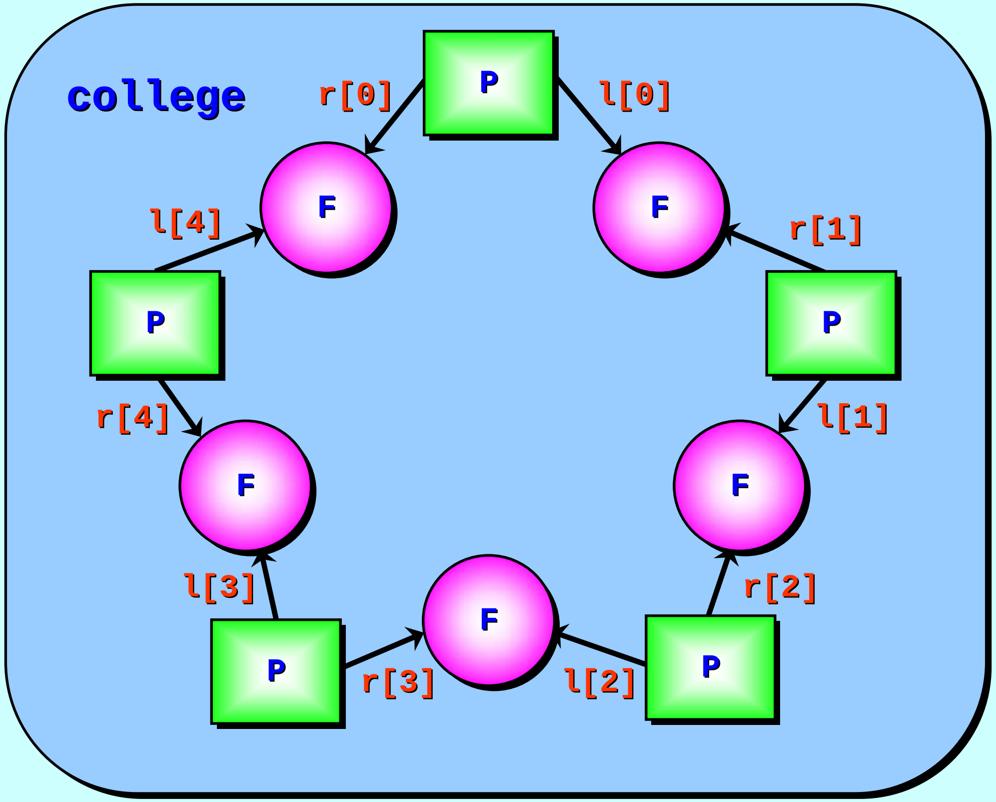

```
PROC philosopher (CHAN BOOL left!, right!, down!, up!)
  WHILE TRUE
    SEQ
      ...  think
      down ! TRUE              -- get permission to sit down
      PAR                      -- pick up forks
        left ! TRUE
        right ! TRUE
      ...  eat
      PAR                      -- put down forks
        left ! TRUE
        right ! TRUE
      up ! TRUE                -- notify security that
                               -- you have finished
:
```



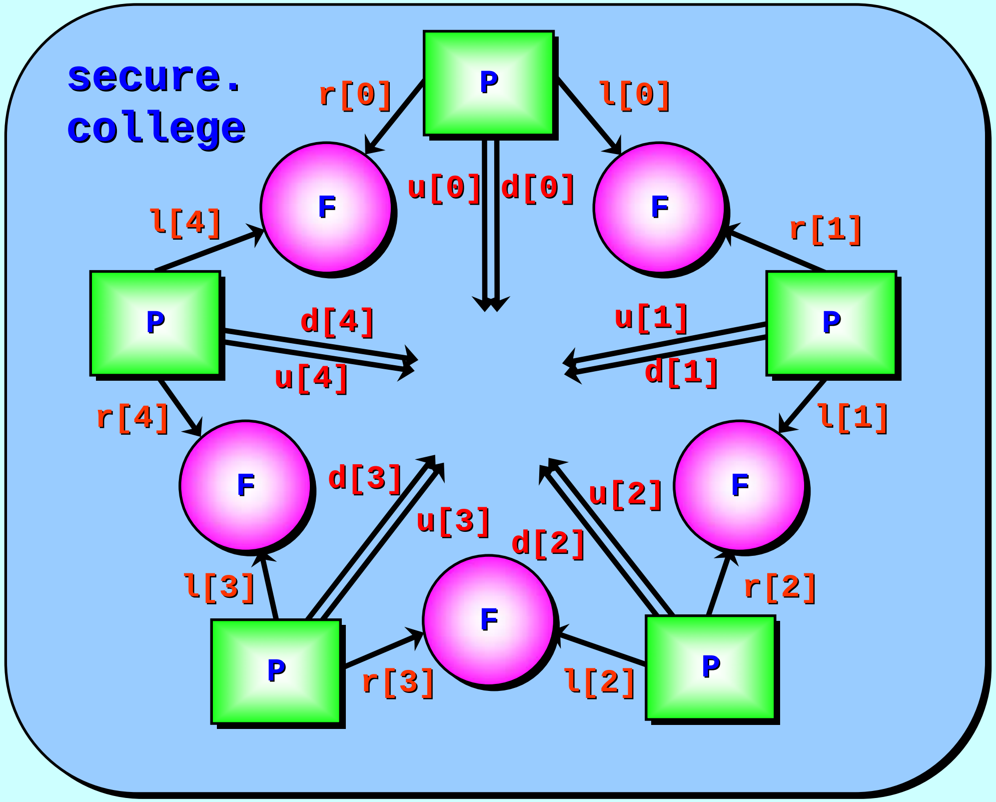

```
PROC security ([]CHAN BOOL down?, up?)
  VAL INT max IS (SIZE down?) - 1:
  INITIAL INT n.sat.down IS 0:
  WHILE TRUE
    BOOL any:
    ALT i = 0 FOR SIZE down?
      ALT
        (n.sat.down < max) & down[i] ? any
          n.sat.down := n.sat.down + 1
        up[i] ? any
          n.sat.down := n.sat.down - 1
:
```



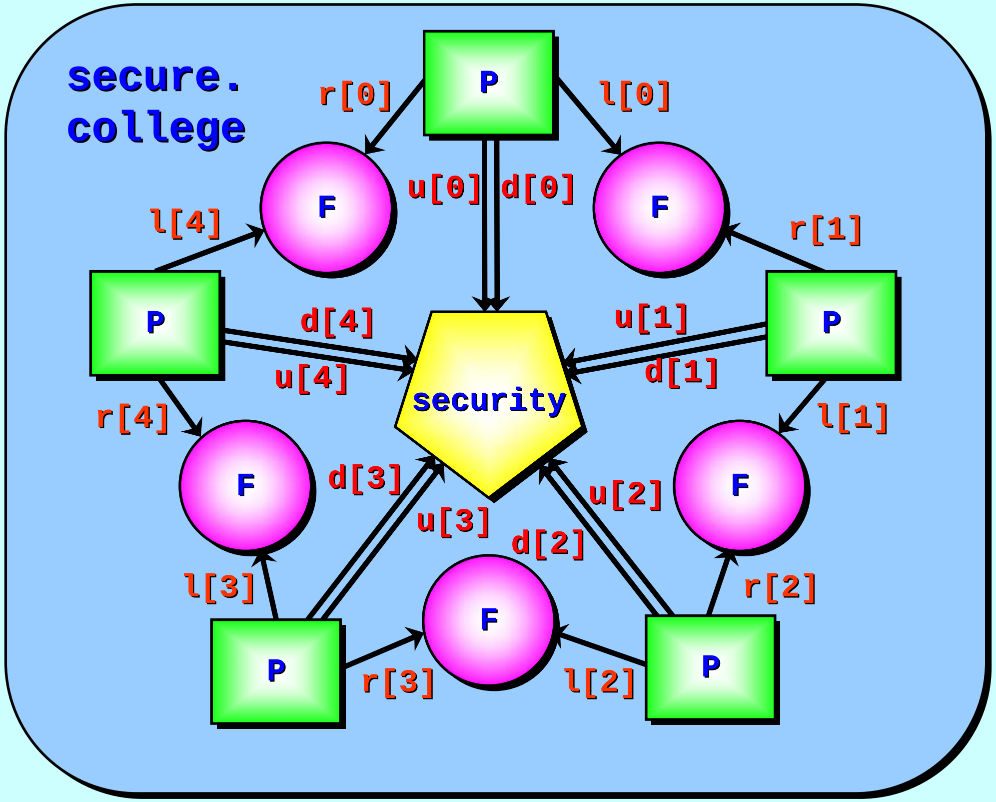
"**d**" = "**down**", "**u**" = "**up**" channels (for indicating wish to sit down or stand up)




```
PROC college ()
  [5]CHAN BOOL left, right:



  PAR i = 0 FOR 5
    PAR
      philosopher (left[i]!, right[i]!)
      fork (left[i]?, right [(i+1)\5]?)
:
```




```
PROC secure.college ()
  [5]CHAN BOOL left, right:
  [5]CHAN BOOL up, down:


    PAR i = 0 FOR 5
      PAR
        philosopher (left[i]!, right[i]!, down[i]!, up[i]!)
        fork (left[i]?, right [(i+1)\5]?)
:
```




```
PROC secure.college ()
  [5]CHAN BOOL left, right:
  [5]CHAN BOOL up, down:
  PAR
    security (down?, up?)
    PAR i = 0 FOR 5
      PAR
        philosopher (left[i]!, right[i]!, down[i]!, up[i]!)
        fork (left[i]?, right [(i+1)\5]?)
:
```

The potential for deadlock in **`college`** was not obvious to its designers.

The claim that there is no such potential within **`secure.college`** should not be accepted lightly.

We must provide a *(formal) proof* of the absence of deadlock in any safety-critical application.

Systematic validation through “exhaustive” testing is unacceptable ... *been there* ... *doesn’t work* ... *!!!*

DEF (informal): DEADLOCK

A network of processes is deadlocked when every process is blocked trying to communicate with other processes *within that network*.

If any process within the network is blocked waiting for *an external communication*, its environment may eventually offer that communication – and the network would proceed. It is not deadlocked.

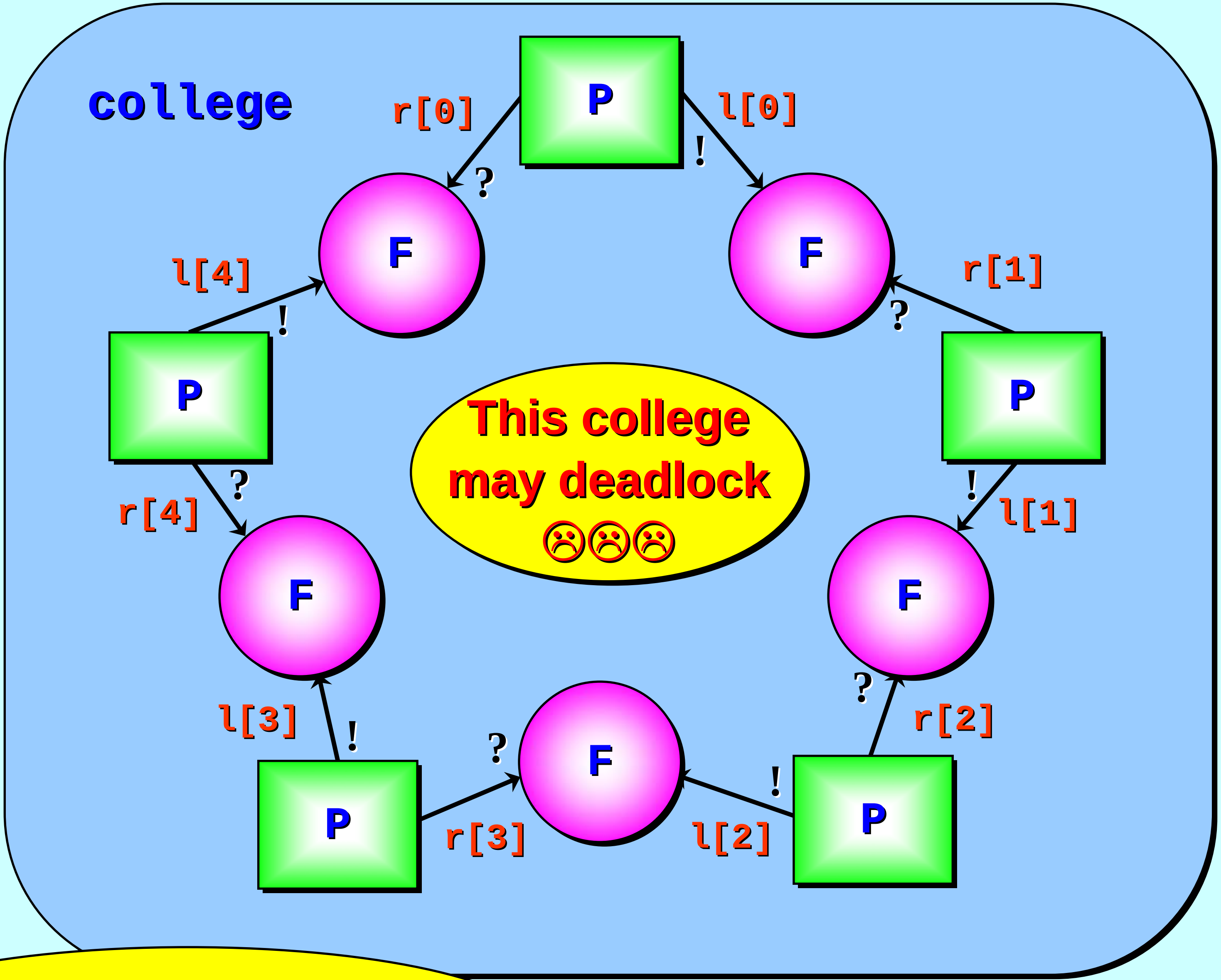
If any process within the network is blocked *on a timeout*, that process will eventually continue – and the network is not deadlocked.

A deadlocked network *refuses* all external events (communications, the passing of time, ...), as well as all internal activity.

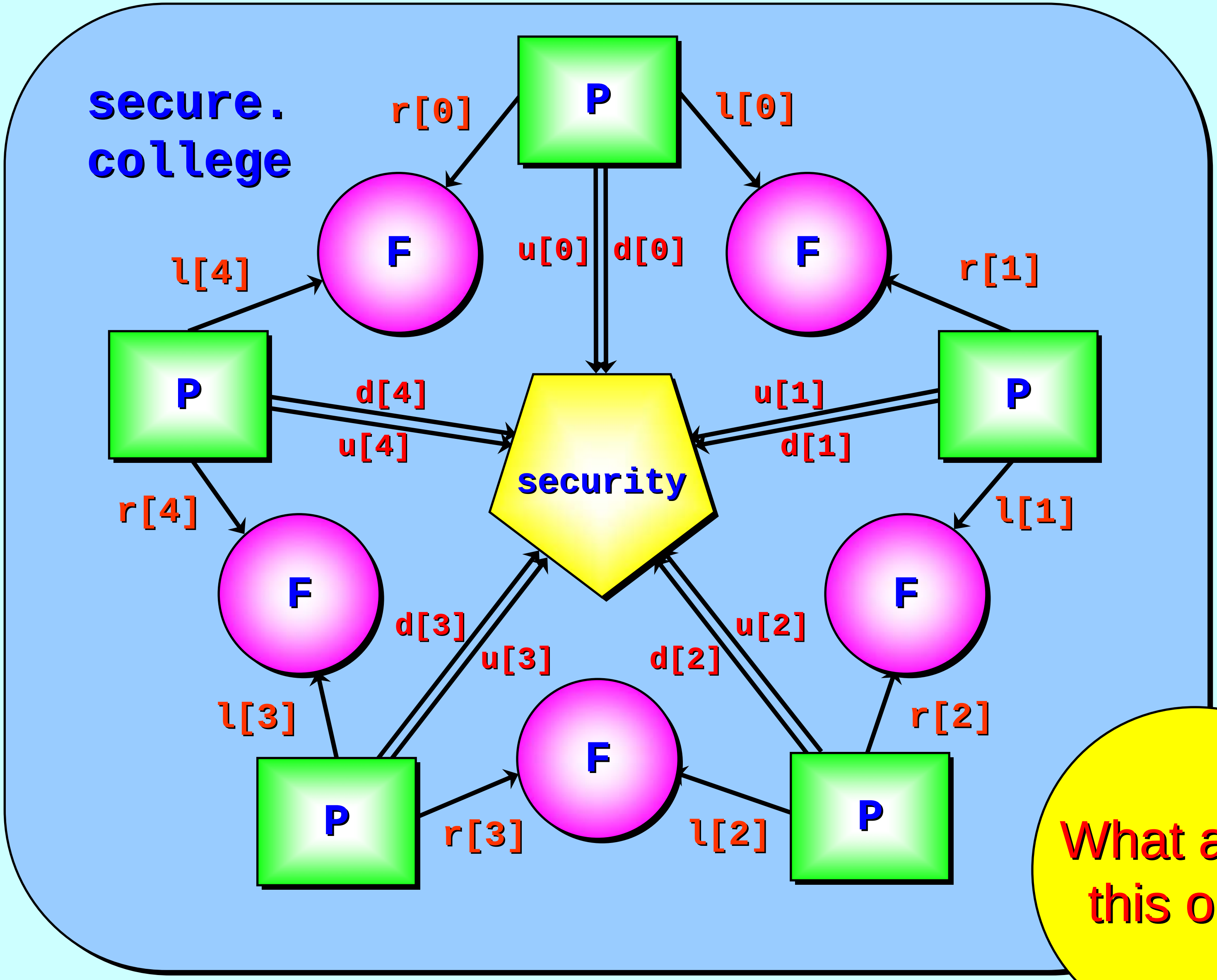
DEF (informal): DEADLOCK

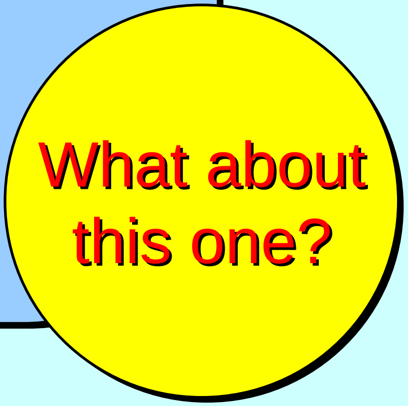
A network of processes is deadlocked when every process is blocked trying to communicate with other processes *within that network*.

Theorem: a deadlocked network will contain a cycle of processes with each process in the cycle blocked trying to communicate with the next node in the cycle.



Note the cycle of blocked communications


secure.
college
P
P
P
P
P
F
F
F
F
F
security
r[0]
l[0]
u[0]
d[0]
r[1]
l[1]
u[1]
d[1]
r[2]
l[2]
u[2]
d[2]
r[3]
l[3]
u[3]
d[3]
r[4]
l[4]
u[4]
d[4]


What about this one?

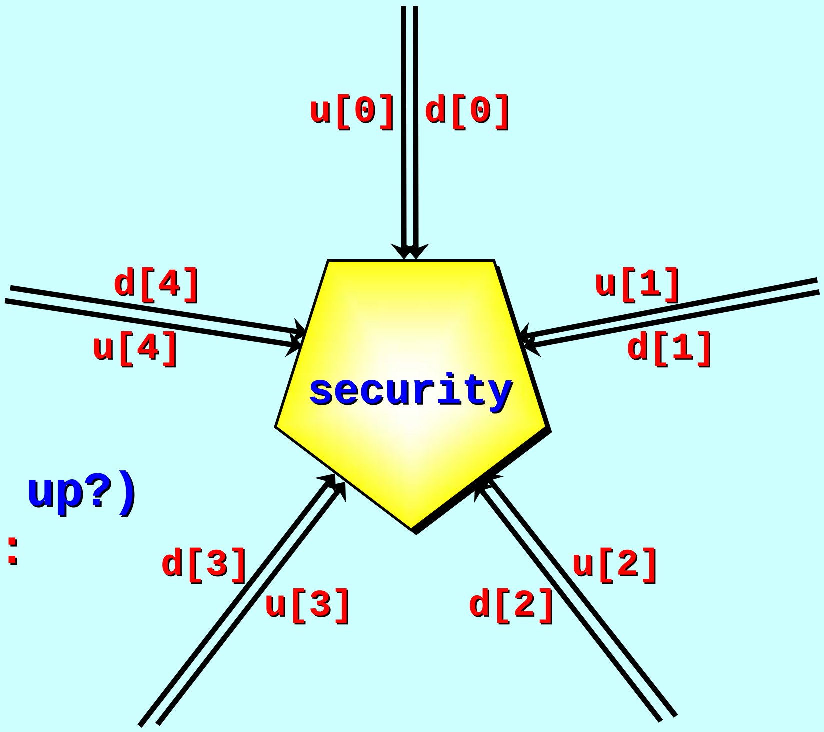
The claim that there is no deadlock within **`secure.college`** should not be accepted lightly.

ASSUME: **`secure.college`** is deadlocked …

In that case, all its processes – each **`philosopher`**, each **`fork`** and the **`security`** guard are blocked. Where might they be?

The **`security`** guard can only be in one place – blocked on its **`ALT`**, waiting for a **`philosopher`** to enter/leave the dining room.

Waits for signals on up or down channels ...




```
PROC security ([]CHAN BOOL down?, up?)
  VAL INT max IS (SIZE down?) – 1:
  INT n.sat.down:
  SEQ
    n.sat.down := 0
    WHILE TRUE
      BOOL any:
      ALT i = 0 FOR SIZE down?
        ALT
          (n.sat.down < max) & down[i] ? any
            n.sat.down := n.sat.down + 1
          up[i] ? any
            n.sat.down := n.sat.down – 1
:
```

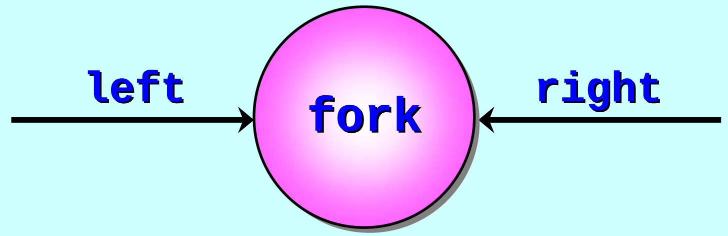
The claim that there is no deadlock within **`secure.college`** should not be accepted lightly.

ASSUME: **`secure.college`** is deadlocked …

In that case, all its processes – each **`philosopher`**, each **`fork`** and the **`security`** guard are blocked. Where might they be?

The **`security`** guard can only be in one place – blocked on its **`ALT`**, waiting for a **`philosopher`** to enter/leave the dining room.

Each **`fork`** is either on the table or in the hands of one of its neighbouring philosophers.




```
PROC fork (CHAN BOOL left?, right?)
  WHILE TRUE
    BOOL any:
    ALT
      left ? any                -- left phil picks up
        left ? any              -- left phil puts down
      right ? any               -- right phil picks up
        right ? any             -- right phil puts down
:
```

On table – waiting to be picked up ...

Held by ‘right’ philosopher – waiting to be put down ...

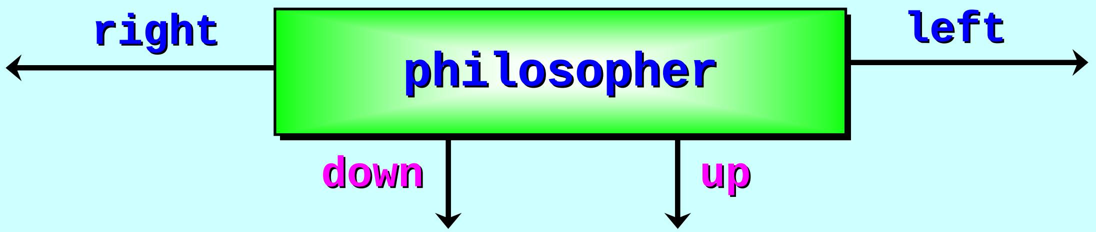
Held by ‘left’ philosopher – waiting to be put down ...

The claim that there is no deadlock within **`secure.college`** should not be accepted lightly.

ASSUME: **`secure.college`** is deadlocked …

In that case, all its processes – each **`philosopher`**, each **`fork`** and the **`security`** guard are blocked. Where might they be?

Each **`philosopher`** could be in one of several places – thinking, trying to get past **`security`**, trying to pick up its **`fork`**s, eating, trying to put down its forks or trying to leave the dining room (i.e. telling **`security`** that it's leaving).




```
PROC philosopher (CHAN BOOL left!, right!, down!, up!)
  WHILE TRUE
    SEQ
      ...  think
      down ! TRUE                 -- get permission to sit down
      PAR                         -- pick up forks
        left ! TRUE
        right ! TRUE
      ...  eat
      PAR                         -- put down forks
        left ! TRUE
        right ! TRUE
      up ! TRUE                   -- notify security that
                                  -- you have finished
:
```

Can’t get stuck here!

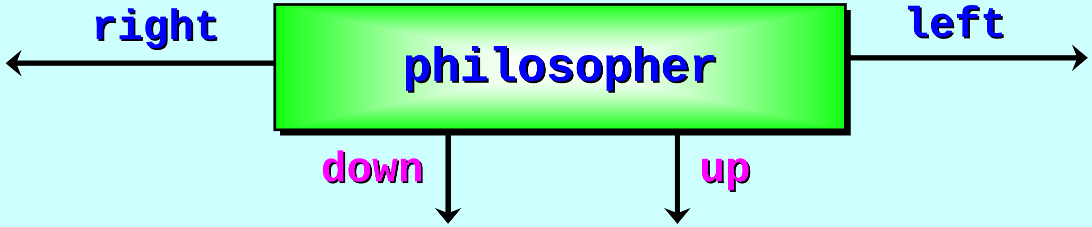
Four must get past here …

The claim that there is no deadlock within **`secure.college`** should not be accepted lightly.

ASSUME: **`secure.college`** is deadlocked …

In that case, all its processes – each **`philosopher`**, each **`fork`** and the **`security`** guard are blocked. Where might they be?

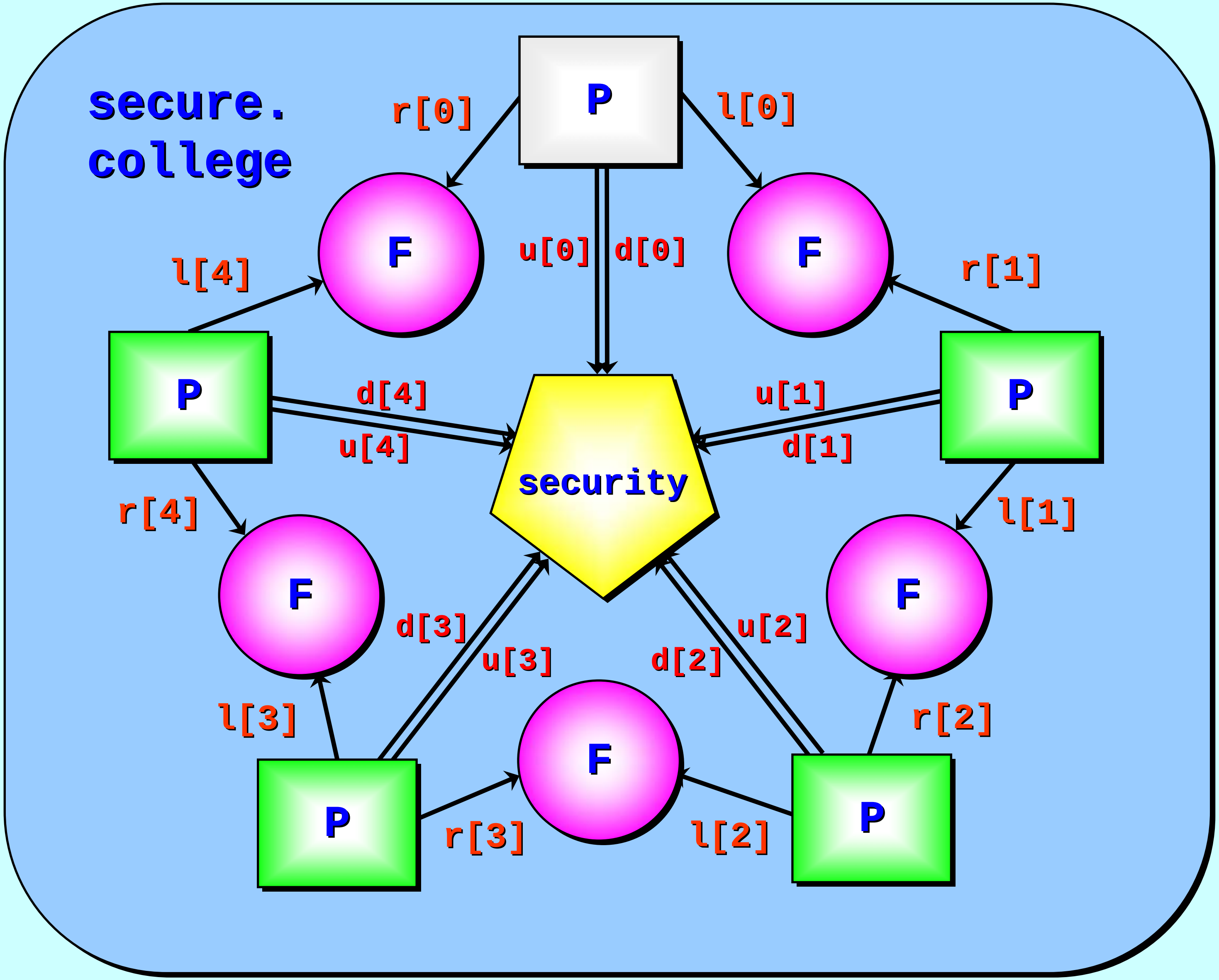
Therefore, one **`philosopher`** must be stuck trying to get past **`security`**. The other four must be in the dining room, trying to pick up their **`fork`**s. No **`philosopher`** can have picked up *both* **`fork`**s (else s/he would be eating – which is in the non-stuck region).



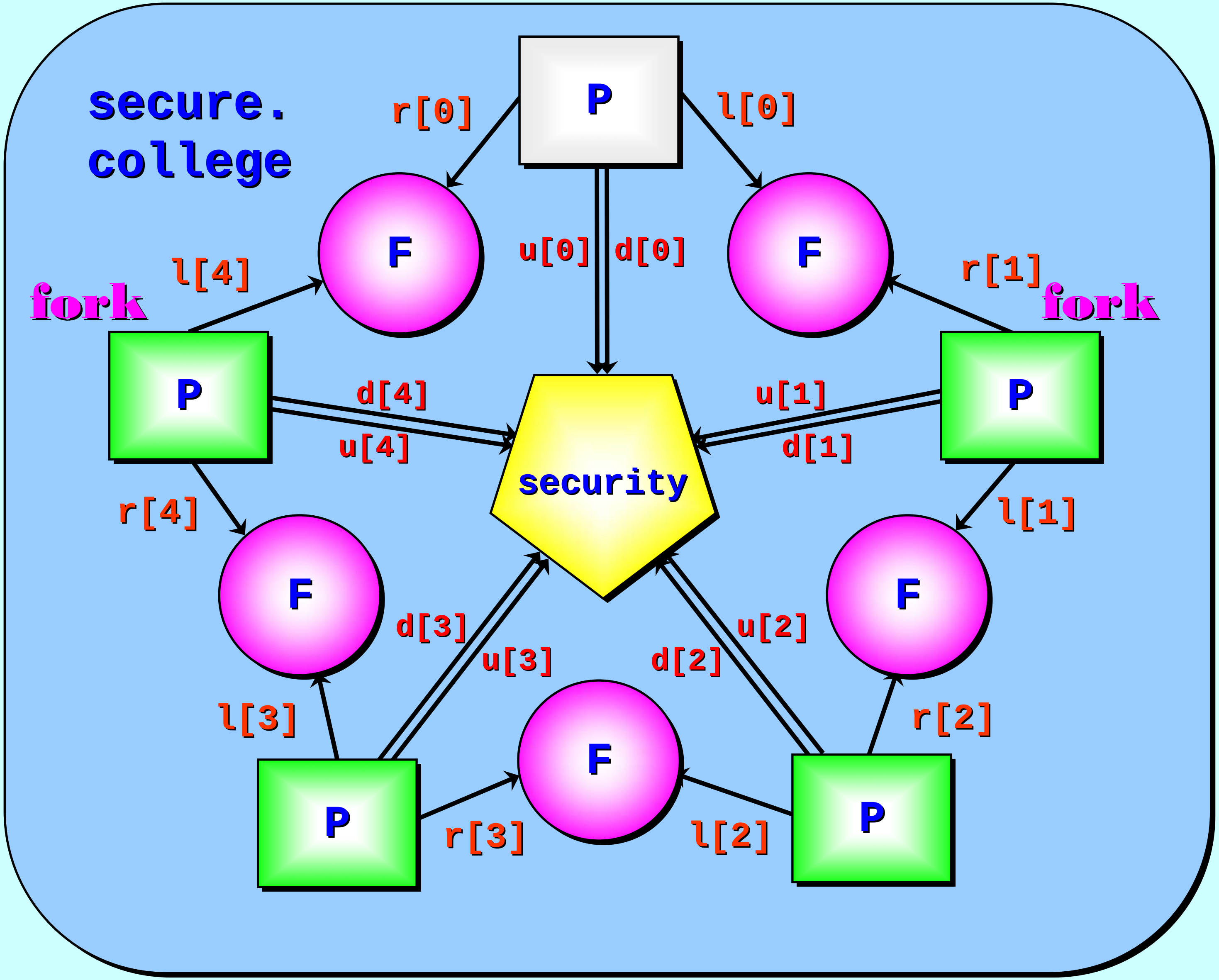

```
PROC philosopher (CHAN BOOL left!, right!, down!, up!)
  WHILE TRUE
    SEQ
      ...  think
      down ! TRUE            -- get permission to sit down
      PAR                    -- pick up forks
        left ! TRUE
        right ! TRUE
      ...  eat
      PAR                    -- put down forks
        left ! TRUE
        right ! TRUE
      up ! TRUE              -- notify security that
                             -- you have finished
:
```

Can't get stuck here!

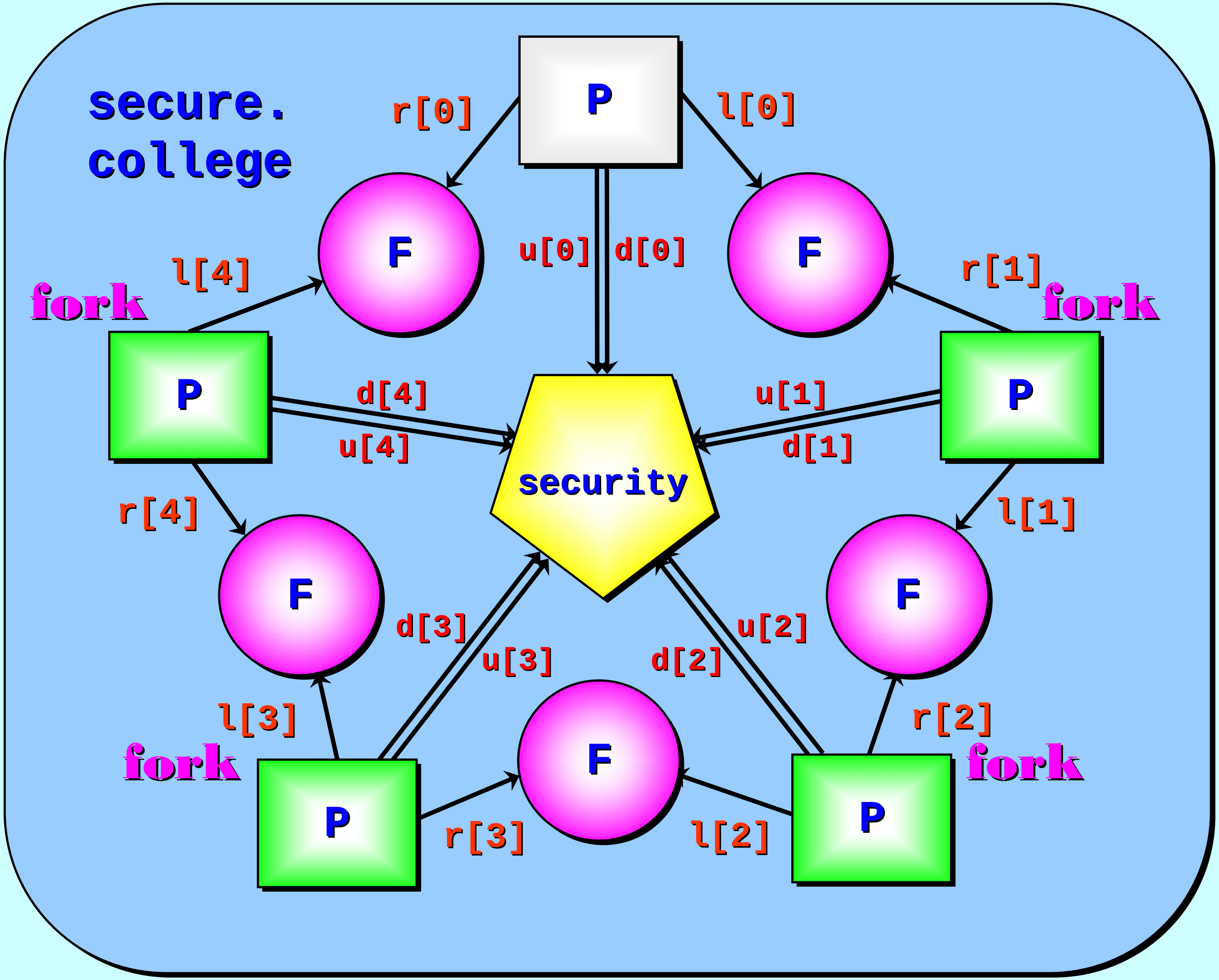
Four must get past here …



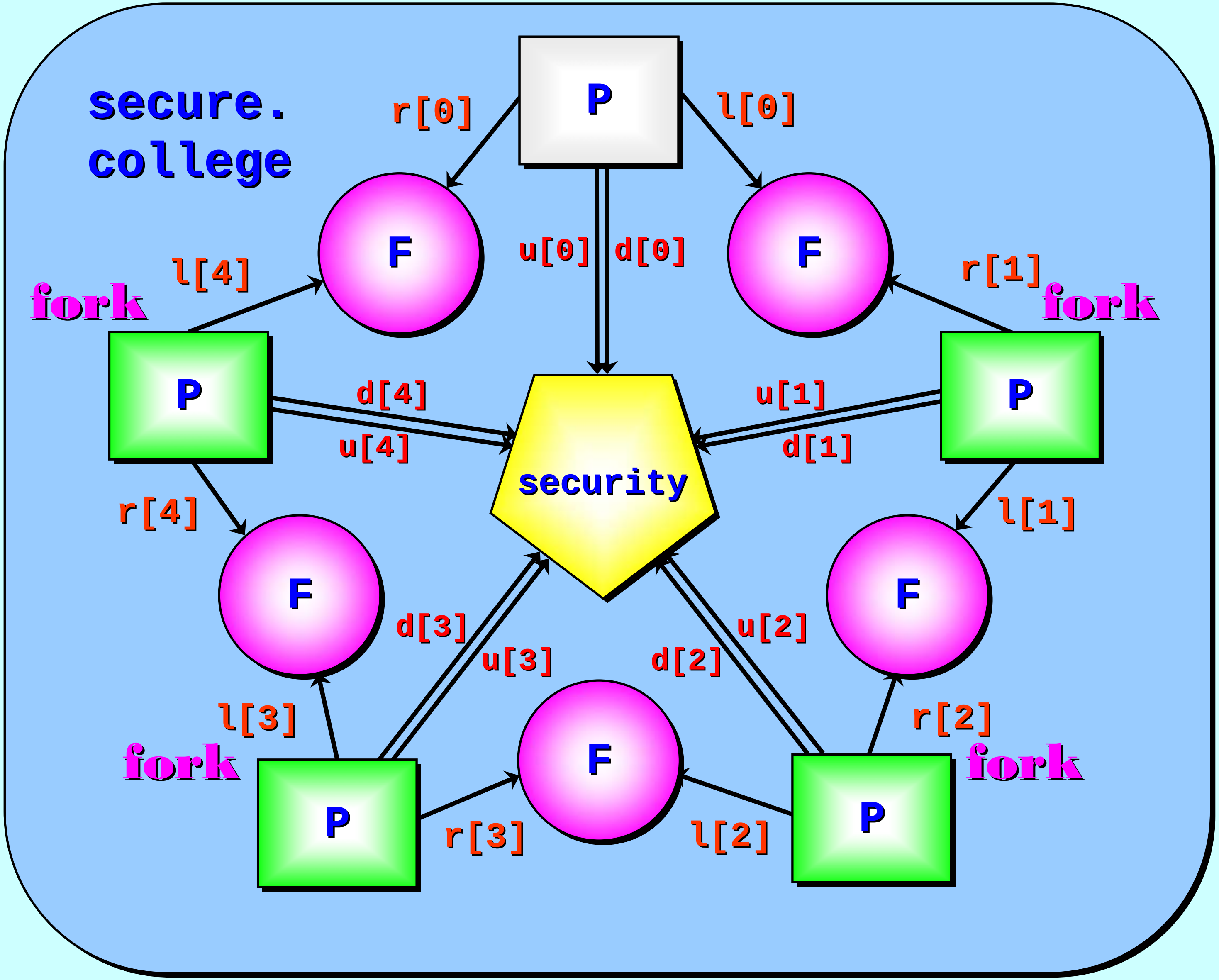
Without loss of generality, suppose it's the top philosopher who is not there.



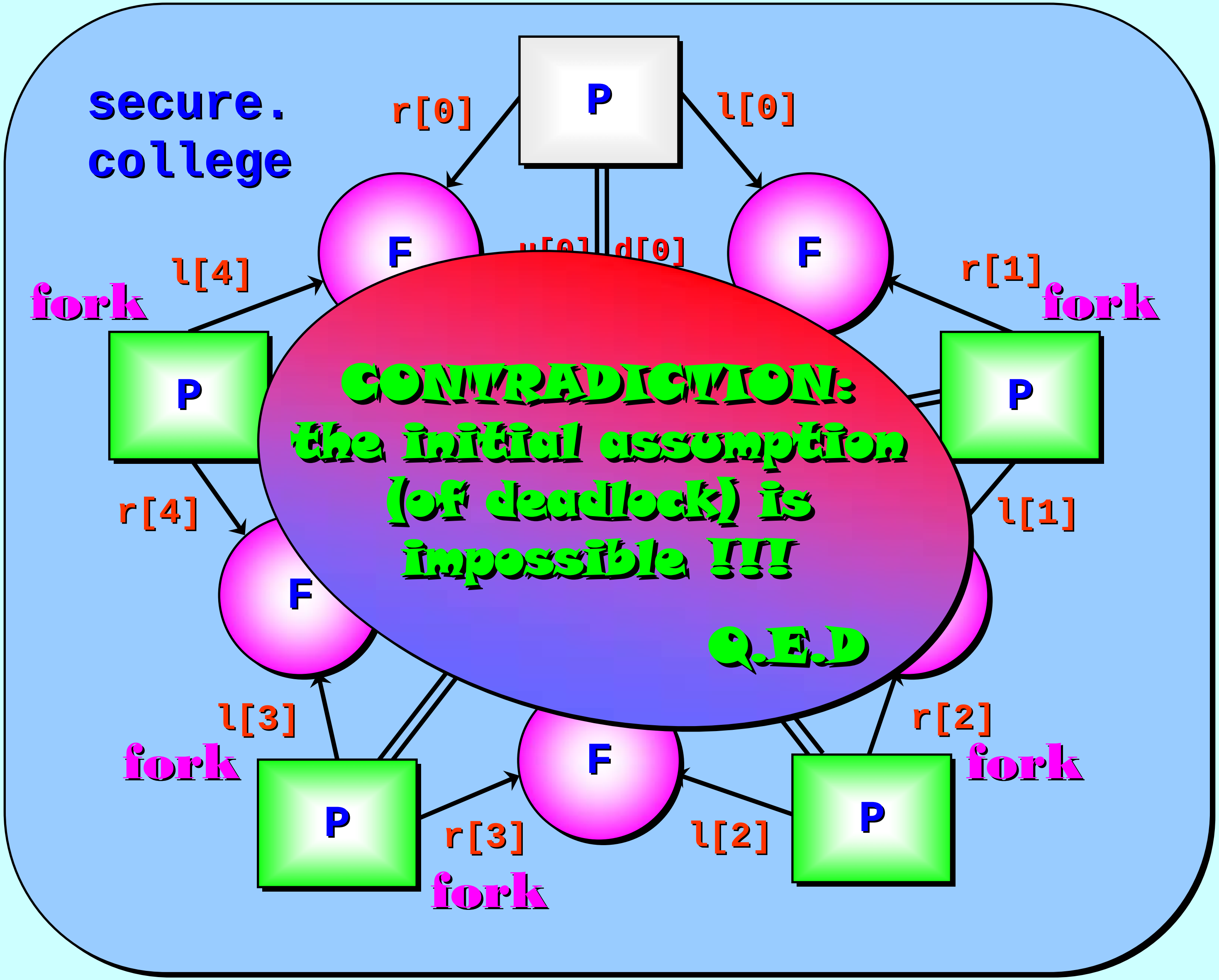
Philosophers 1 and 4 must get the top forks …



They can't have both their forks – so philosophers 2 and 3 must have them …



Philosophers 2 and 3 can't have both their forks …



... but ONE WILL GET BOTH !!!

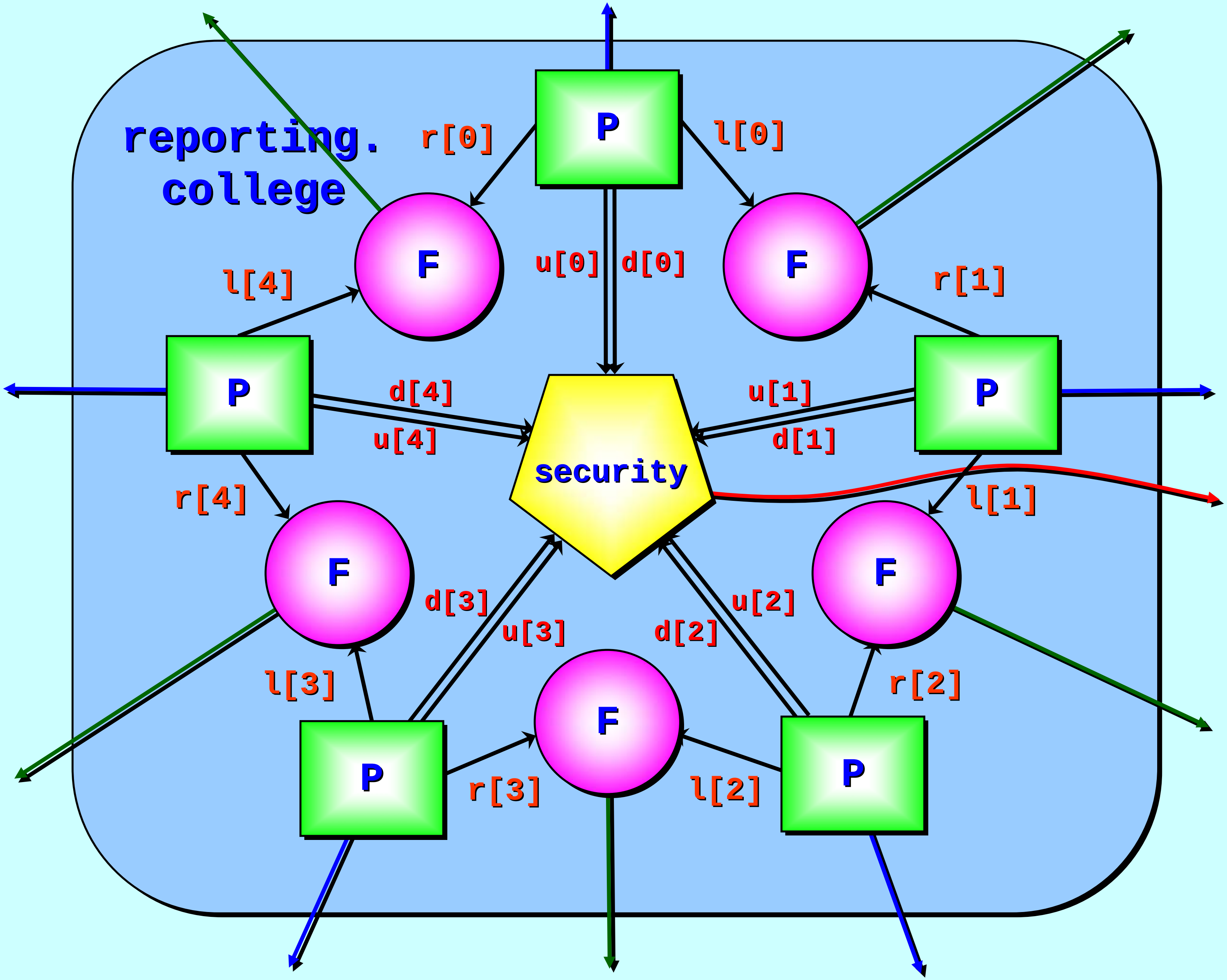
Exercise:

Provide a similar (informal) proof that the other methods (slide 31) for ensuring freedom from deadlock do just that!

Exercise:

Provide some links from **`secure.college`** to the outside world and animate an interactive demonstration of life inside.

Modify the **`fork`** process so that it guarantees service on each input – even if one of them is perpetually busy. No philosopher must starve because of greedy colleagues!


reporting.
college
P
r[0]
l[0]
F
F
u[0]
d[0]
l[4]
r[1]
P
P
d[4]
u[1]
u[4]
d[1]
security
r[4]
l[1]
F
F
d[3]
u[2]
u[3]
d[2]
l[3]
r[2]
F
P
P
r[3]
l[2]

Applying ...

The dining philosophers ...

Compiling …

Real-time inference engine …

Fast fourier transform …

Computing on global data …

Neural nets …

Microprocessor design …

Autonomous robots ...

To appear ...

Applying ...

- The dining philosophers ...
- Compiling …
- Real-time inference engine …
- Fast fourier transform …
- Computing on global data …
- Neural nets …
- Microprocessor design …
- Autonomous robots ...

To appear ...

Applying ...

The dining philosophers ...

Compiling …

Real-time inference engine …

Fast fourier transform …

Computing on global data …

Neural nets …

Microprocessor design …

Autonomous robots ...

To appear ...

Applying ...

The dining philosophers ...

Compiling …

Real-time inference engine …

Fast fourier transform …

Computing on global data …

Neural nets …

Microprocessor design …

Autonomous robots ...

To appear …

Applying ...

The dining philosophers ...

Compiling …

Real-time inference engine …

Fast fourier transform …

Computing on global data …

Neural nets …

Microprocessor design …

Autonomous robots ...

To appear …

Applying ...

The dining philosophers ...

Compiling …

Real-time inference engine …

Fast fourier transform …

Computing on global data …

Neural nets …

Microprocessor design …

Autonomous robots ...

To appear ...

Applying ...

The dining philosophers ...

Compiling …

Real-time inference engine …

Fast fourier transform …

Computing on global data …

Neural nets …

Microprocessor design …

Autonomous robots ...

To appear ...