Embedded Software

Thread synchronization II

AARHUS UNIVERSITY

Agenda

- Mutexes/Semaphores
  - Pitfalls
  - Priority Inversion
    - Problem
    - Solution
      - Inversion
      - Inheritance
      - Ceiling
  - Deadlocks

AARHUS UNIVERSITY

Mutex & Semaphore pitfalls

Mutexes & Semaphores: Pitfalls

- It is extremely easy to get in trouble with mutexes!
- ***Example 1: Find and explain the problem***

```
unsigned int shared;
Mutex m = MUTEX_INITIALIZER;

threadFunc()
{
   while(true)
   {
    lock(m);
    shared++;
    sleep(ONE_SECOND);
    unlock(m);
   }
}


main()
{
   shared = 0;
   createThread(threadFunc);
   createThread(threadFunc);
   for(;;) sleep(100);
}
```

Mutexes & Semaphores: Pitfalls

- It is extremely easy to get in trouble with mutexes!

- ***Example 1: Find and explain the problem***

```
unsigned int shared;
Mutex m = MUTEX_INITIALIZER;

threadFunc()
{
   while(true)
   {
    lock(m);
    shared++;
    sleep(ONE_SECOND);
    unlock(m);
   }
}


main()
{
   shared = 0;
   createThread(threadFunc);
   createThread(threadFunc);
   for(;;) sleep(100);
}
```

m is held for a **full second**, ***blocking*** the other thread

Mutexes & Semaphores: Pitfalls

- It is extremely easy to get in trouble with mutexes!
- ***Example 2: Find and explain the problem***

```
unsigned int shared;
Mutex m = MUTEX_INITIALIZER;

threadFunc()
{
   lock(m);
   while(true)
   {
    shared++;
    sleep(ONE_SECOND);
   }
   unlock(m);
}

main()
{
   shared = 0;
   createThread(threadFunc);
   createThread(threadFunc);
   for(;;) sleep(100);
}
```

Mutexes & Semaphores: Pitfalls

- It is extremely easy to get in trouble with mutexes!
- ***Example 2: Find and explain the problem***

```
unsigned int shared;
Mutex m = MUTEX_INITIALIZER;

threadFunc()
{
   lock(m);
   while(true)
   {
    shared++;
    sleep(ONE_SECOND);
   }
   unlock(m);
}

main()
{
   shared = 0;
   createThread(threadFunc);
   createThread(threadFunc);
   for(;;) sleep(100);
}
```

You're in a world of pain!

Mutexes & Semaphores: Pitfalls

- It is extremely easy to get in trouble with mutexes!

- ***Example 3: Find and explain the problem***

```
unsigned int shared;
SEM_ID s;

threadFunc()
{
   while(true)
   {
    take(s);
    shared++;
    release(s);
    sleep(ONE_SECOND);
   }
}

main()
{
   shared = 0;
   s = createSem(0);
   createThread(threadFunc);
   createThread(threadFunc);
   for(;;) sleep(100);
}
```

Mutexes & Semaphores: Pitfalls

- It is extremely easy to get in trouble with mutexes!

- ***Example 3: Find and explain the problem***

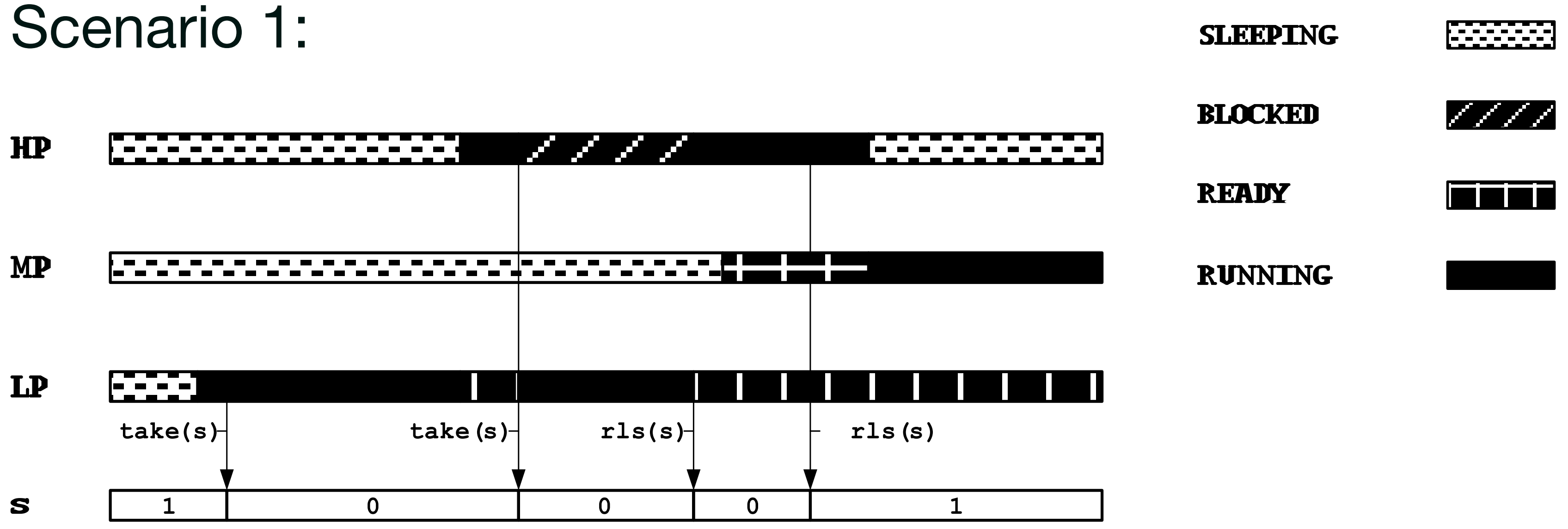
```
unsigned int shared;
SEM_ID s;

threadFunc()
{
   while(true)
   {
    take(s);
    shared++;
    release(s);
    sleep(ONE_SECOND);
   }
}

main()
{
   shared = 0;
   s = createSem(0);
   createThread(threadFunc);
   createThread(threadFunc);
   for(;;) sleep(100);
}
```

s is initialized to 0 – no one can pass **take()** before someone calls **release()**

Mutex priority

Mutexes & Semaphores: Pitfalls

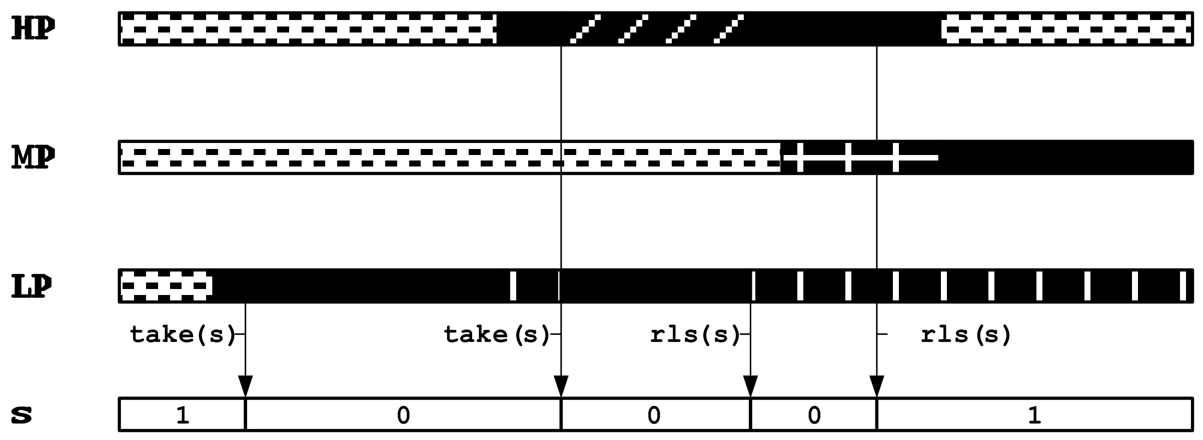
- Scenario 1:



- Scenario 2 (MP arrives a little earlier):

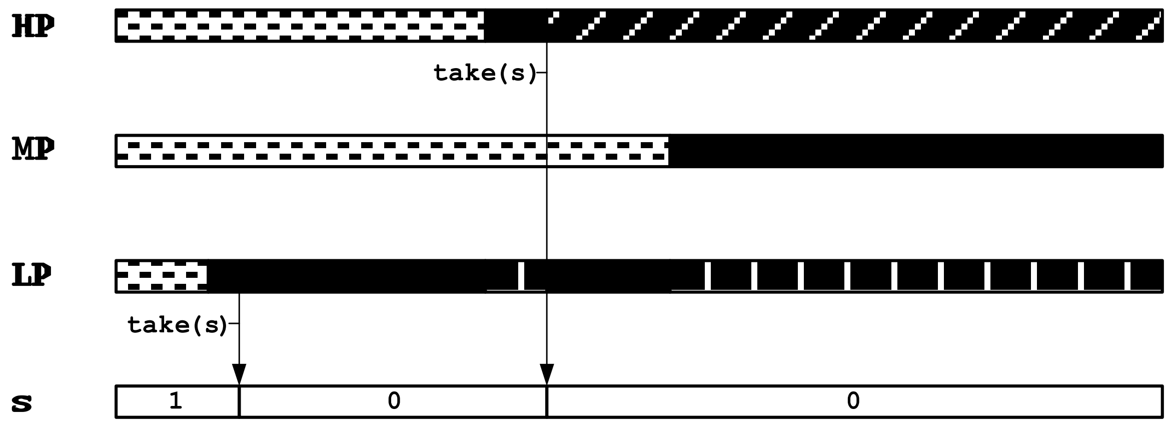
Mutexes & Semaphores: Pitfalls

- Scenario 1:



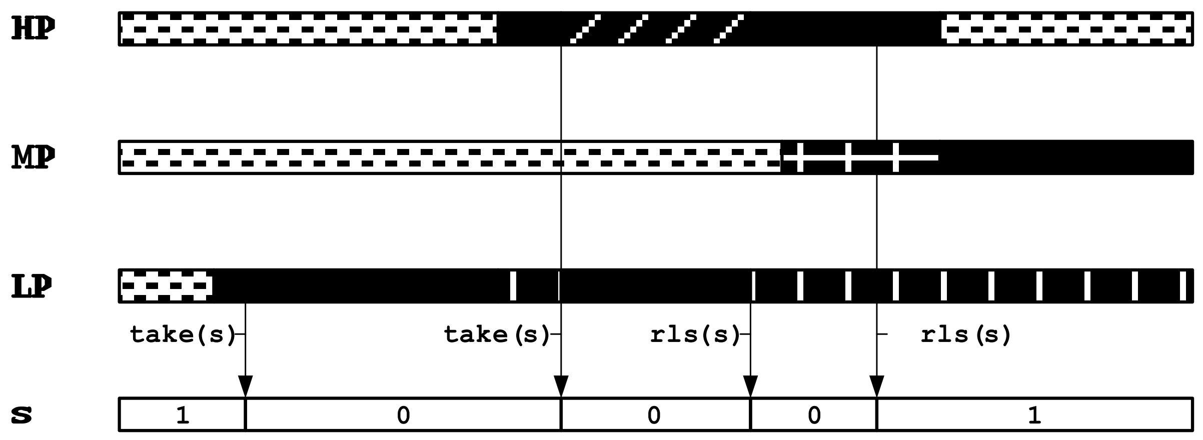
SLEEPING	▒
BLOCKED	▨
READY	▥
RUNNING	■

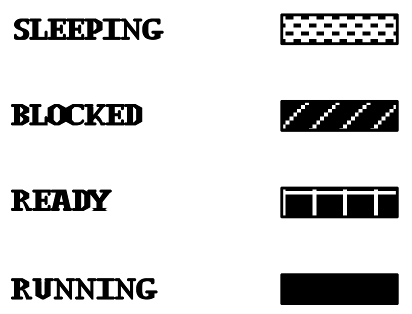
- Scenario 2 (MP arrives a little earlier):



Mutexes & Semaphores: Pitfalls

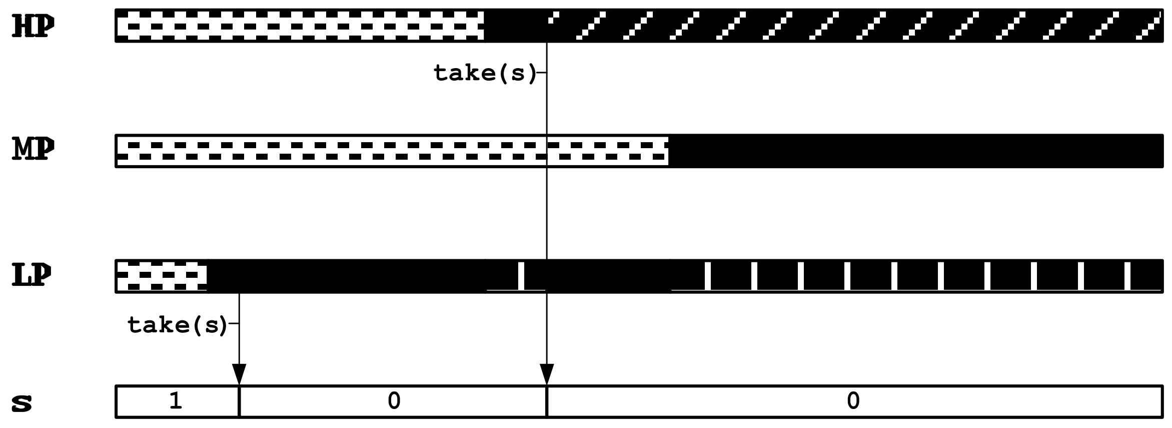
- Scenario 1:





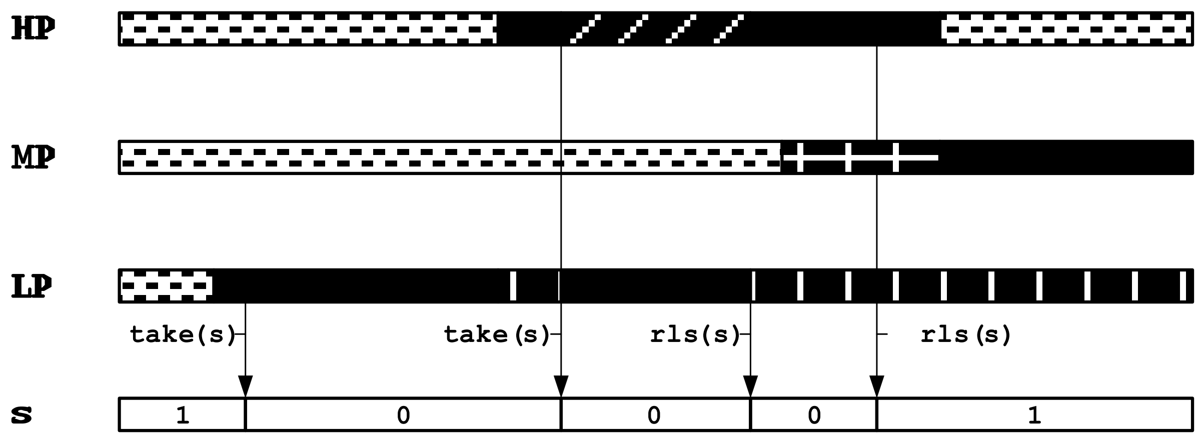
HP: High priority thread
MP: Medium priority thread
LP: Low priority thread

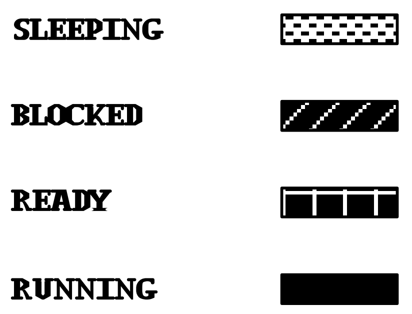
- Scenario 2 (MP arrives a little earlier):



Mutexes & Semaphores: Pitfalls

- Scenario 1:

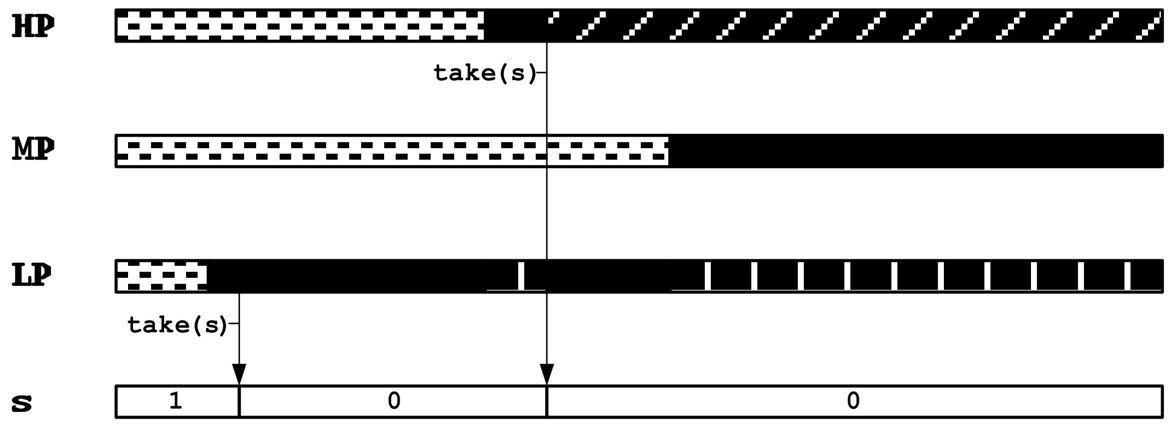




HP: High priority thread
MP: Medium priority thread
LP: Low priority thread

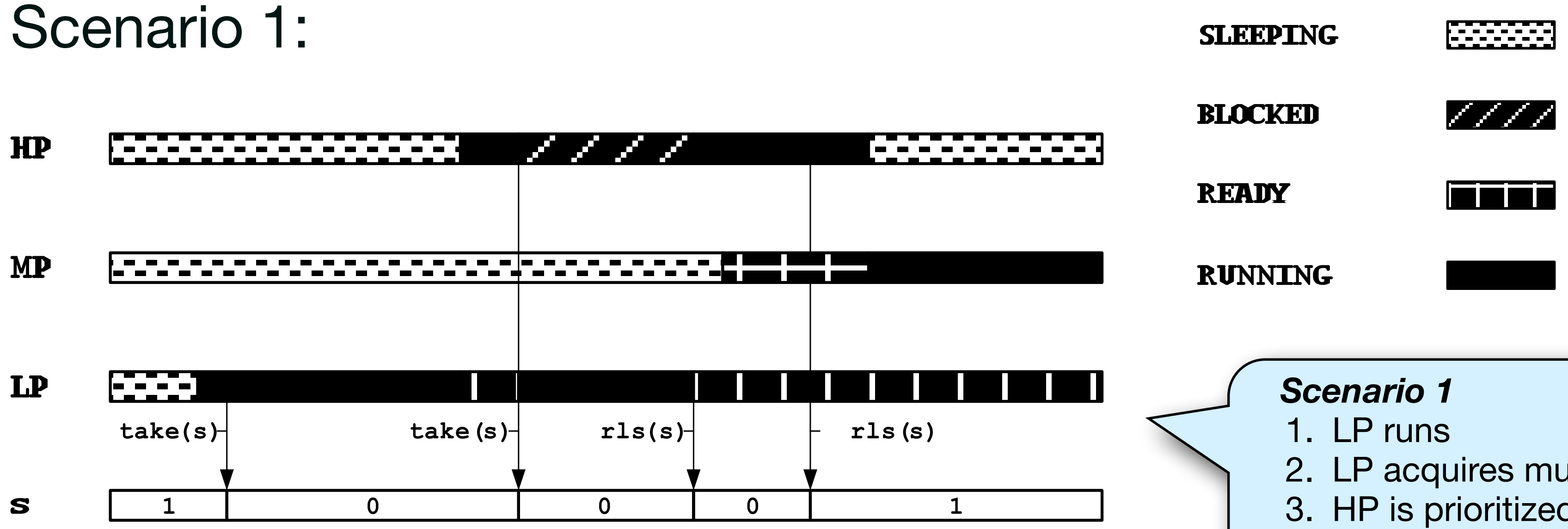
Scenario 1
1. LP runs
2. LP acquires mutex
3. HP is prioritized to run, LP on waiting queue (WQ)
4. HP blocked due to mutex taken
5. LP runs until mutex release
6. HP runs until done, LP on WQ
7. MP is ready but due to lower priority -> WQ
8. LP waits until both HP and MP done and then run until

- Scenario 2 (MP arrives a little earlier):



Mutexes & Semaphores: Pitfalls

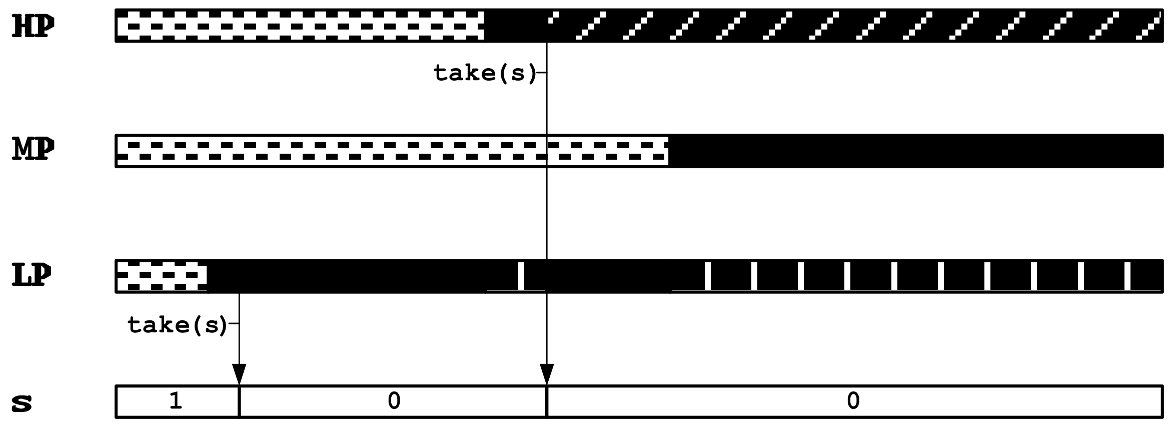
- Scenario 1:



HP: High priority thread
MP: Medium priority thread
LP: Low priority thread

Scenario 1
1. LP runs
2. LP acquires mutex
3. HP is prioritized to run, LP on waiting queue (WQ)
4. HP blocked due to mutex taken
5. LP runs until mutex release
6. HP runs until done, LP on WQ
7. MP is ready but due to lower priority -> WQ
8. LP waits until both HP and MP done and then run until

- Scenario 2 (MP arrives a little earlier):



Scenario 2 - Priority inversion
1. LP runs
2. LP acquires mutex
3. HP is prioritized to run, LP on WQ
4. HP blocked due to mutex taken - LP continues
5. **MP is prioritized to run (over LP) until done**
   MP is thus scheduled *ahead* of HP - priority inversion
6. *LP runs until mutex release, HP is blocked*
7. *HP runs until done*
8. *LP waits until HP done*

Priority inversion

Priority inversion

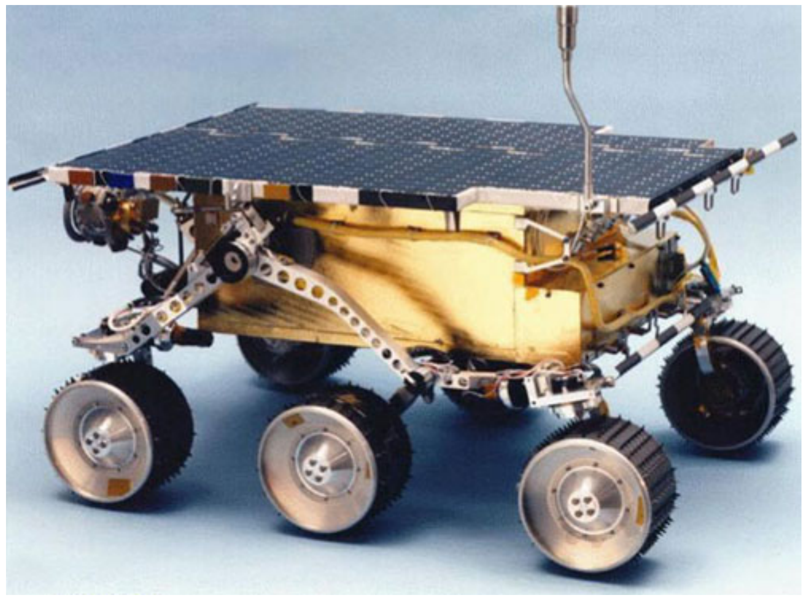
- Priority inversion is a nasty error – especially in RT systems
  - System does not deadlock forever – it just responds slower sometimes
  - “Slower”…”sometimes”…not words the RT system engineer likes!!!

Priority inversion

- Priority inversion is a nasty error – especially in RT systems
  - System does not deadlock forever – it just responds slower sometimes
  - “Slower”…”sometimes”…not words the RT system engineer likes!!!
- The error may go unnoticed or not happen at all, until…
  - Final customer demonstration
  - Your thingy has landed on Mars

Priority inversion

- Priority inversion is a nasty error – especially in RT systems
  - System does not deadlock forever – it just responds slower sometimes
  - "Slower"…"sometimes"…not words the RT system engineer likes!!!
- The error may go unnoticed or not happen at all, until…
  - Final customer demonstration
  - Your thingy has landed on Mars



Mars Pathfinder
Problem: *Ground communications terminated abruptly ($$$!)*
Cause: *HW/SW reset by watchdog*
Cause: *HP data distribution (DD) task not completed on time*
Cause: *DD-task waited for mutex held by LP ASI/MET task, which was preempted by several MP tasks*

AARHUS UNIVERSITY

Priority inversion

Priority inversion

- Priority inversion can be solved by one of two methods:

Priority inversion

- Priority inversion can be solved by one of two methods:
  - **Priority inheritance**: When a thread holds a mutex it is temporarily assigned the priority of the highest-priority thread waiting for the mutex.

Priority inversion

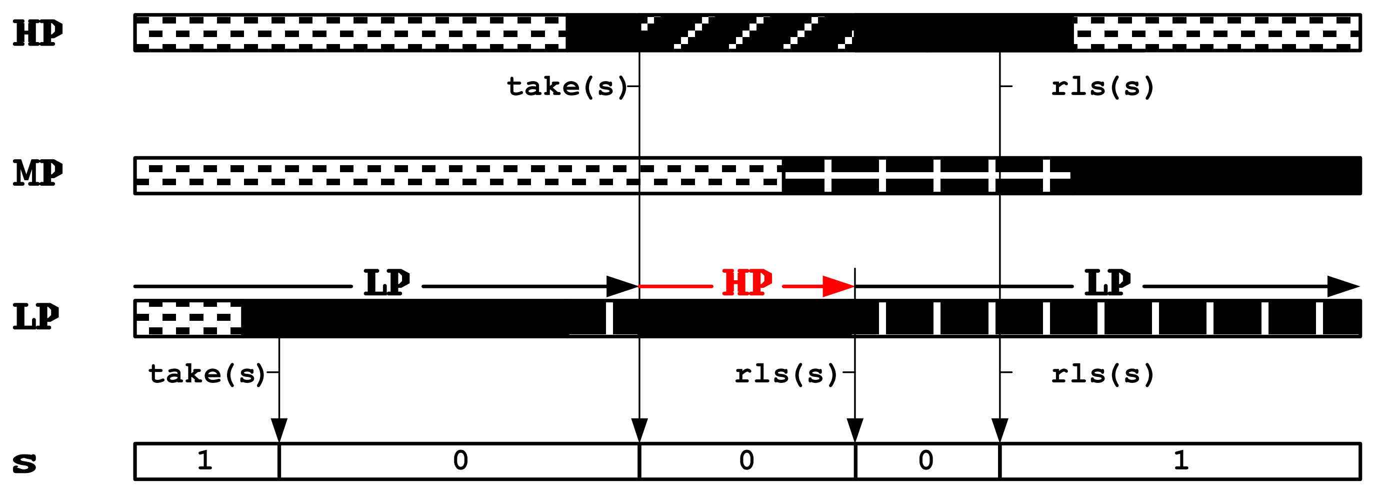
- Priority inversion can be solved by one of two methods:
  - **Priority inheritance**: When a thread holds a mutex it is temporarily assigned the priority of the highest-priority thread waiting for the mutex.
  - **Priority ceiling**: All mutexes are assigned a (high) priority (the priority ceiling) which the owner of the mutex is assigned while it holds the mutex

Priority inversion

- Priority inversion can be solved by one of two methods:
  - **Priority inheritance**: When a thread holds a mutex it is temporarily assigned the priority of the highest-priority thread waiting for the mutex.
  - **Priority ceiling**: All mutexes are assigned a (high) priority (the priority ceiling) which the owner of the mutex is assigned while it holds the mutex
- Note semaphores do ***NOT*** support the above

Priority inheritance

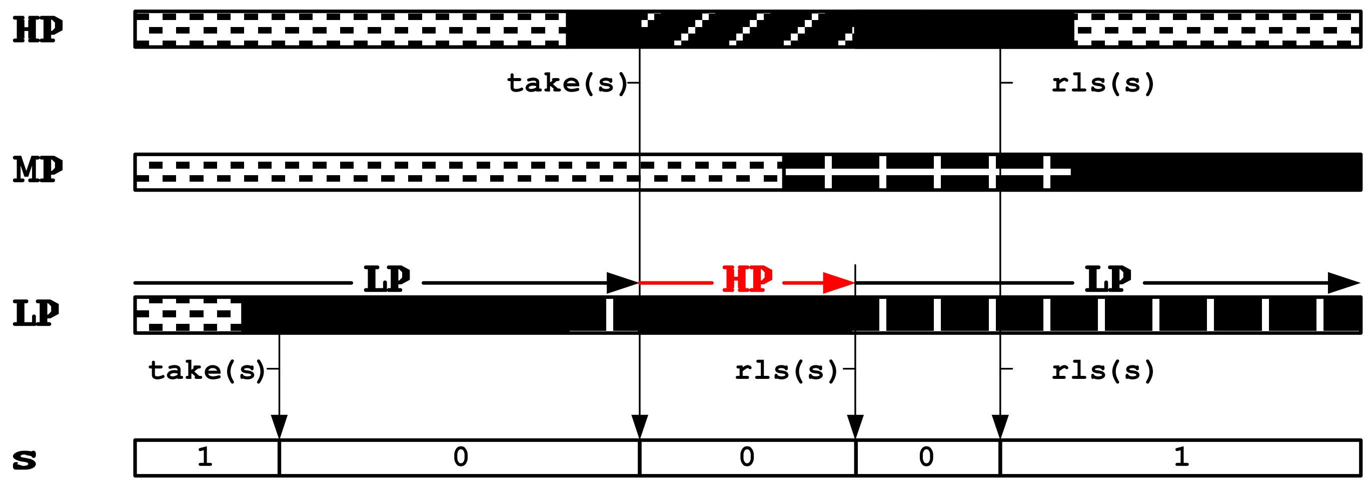
- Priority inheritance:
  - When a thread holds a mutex it is temporarily assigned the priority of the highest-priority thread waiting for the mutex.



- Priority inheritance can be set as a property of some mutexs on creation

Priority inheritance

- Priority inheritance:
  - When a thread holds a mutex it is temporarily assigned the priority of the highest-priority thread waiting for the mutex.

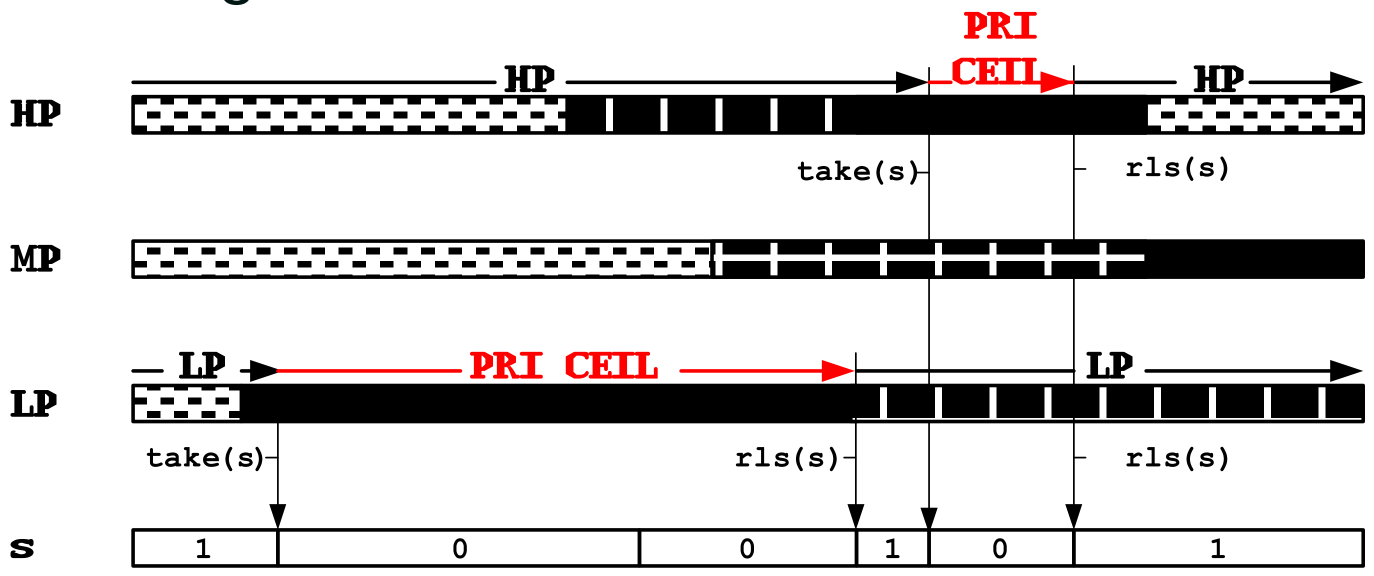


Scenario 1
1. LP runs
2. LP acquires mutex
3. HP is prioritized to run, LP on waiting queue (WQ)
4. HP blocked due to mutex taken
5. LP runs until mutex release, ***but with HP priority (inheritance)***
6. MP wants to run but due to lower priority -> WQ
7. HP acquires mutex and runs until done, MP & LP on WQ
8. MP runs until done, LP on WQ
9. LP runs until done

- Priority inheritance can be set as a property of some mutexs on creation

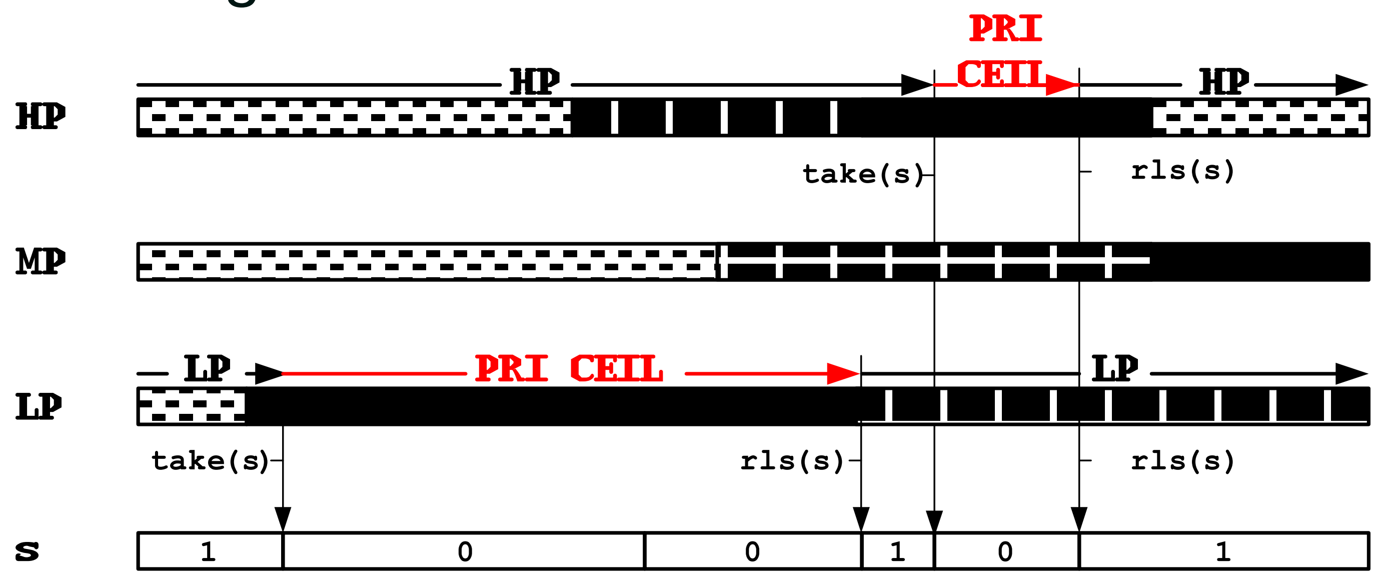
Priority ceiling

- Priority ceiling:
  - All mutexes are assigned a (high) priority (the priority ceiling) which the owner of the mutex is assigned while it holds the mutex



Priority ceiling

- Priority ceiling:
  - All mutexes are assigned a (high) priority (the priority ceiling) which the owner of the mutex is assigned while it holds the mutex



Scenario 1

1. LP runs
2. LP acquires mutex - **its priority is elevated to high priority** - **priority ceiling**
3. HP wants to run but has lower priority -> waiting queue (WQ)
4. MP wants to run but has lower priority -> WQ
5. LP releases mutex and changes priority to low
6. HP acquires mutex and runs until done, MP & LP on WQ
7. MP runs until done, LP on WQ
8. LP run until done

Multiple Mutexes

- …and the fun just started! Introducing multiple mutexes:

Multiple Mutexes

- …and the fun just started! Introducing multiple mutexes:

```
void main()
{
   createThread(printFileFunc1);
   createThread(printFileFunc2);
}
```

Multiple Mutexes

- …and the fun just started! Introducing multiple mutexes:

```
void printFileFunc1()
{
   lock(fileMut);
   lock(printerMut);
   <<print file>>
   unlock(printerMut);
   unlock(fileMut);
}
```

```
void printFileFunc2()
{
   lock(printerMut);
   lock(fileMut);
   <<print file>>
   unlock(fileMut);
   unlock(printerMut);
}
```

```
void main()
{
   createThread(printFileFunc1);
   createThread(printFileFunc2);
}
```

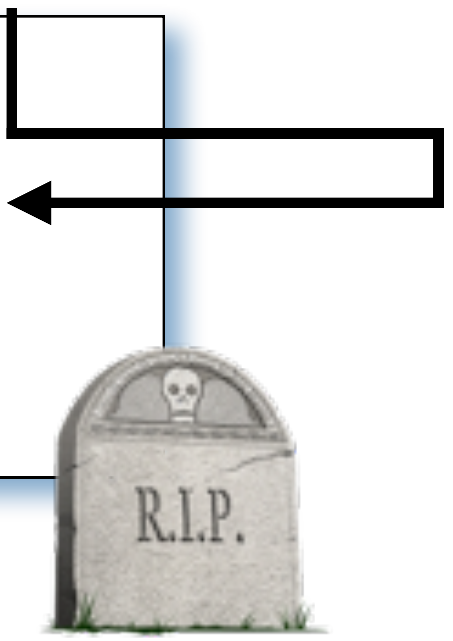
AARHUS UNIVERSITY

Multiple Mutexes

- …and the fun just started! Introducing multiple mutexes:

```
void printFileFunc1()
{
   lock(fileMut);
   lock(printerMut);
   <<print file>>
   unlock(printerMut);
   unlock(fileMut);
}
```

```
void printFileFunc2()
{
   lock(printerMut);
   lock(fileMut);
   <<print file>>
   unlock(fileMut);
   unlock(printerMut);
}
```



```
void main()
{
   createThread(printFileFunc1);
   createThread(printFileFunc2);
}
```

Deadlocks

Deadlocks

- A deadlock is a situation where two (or more) threads are waiting for the other to release a resource, thus neither will ever run.

- The four necessary conditions for deadlocks:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

Deadlocks

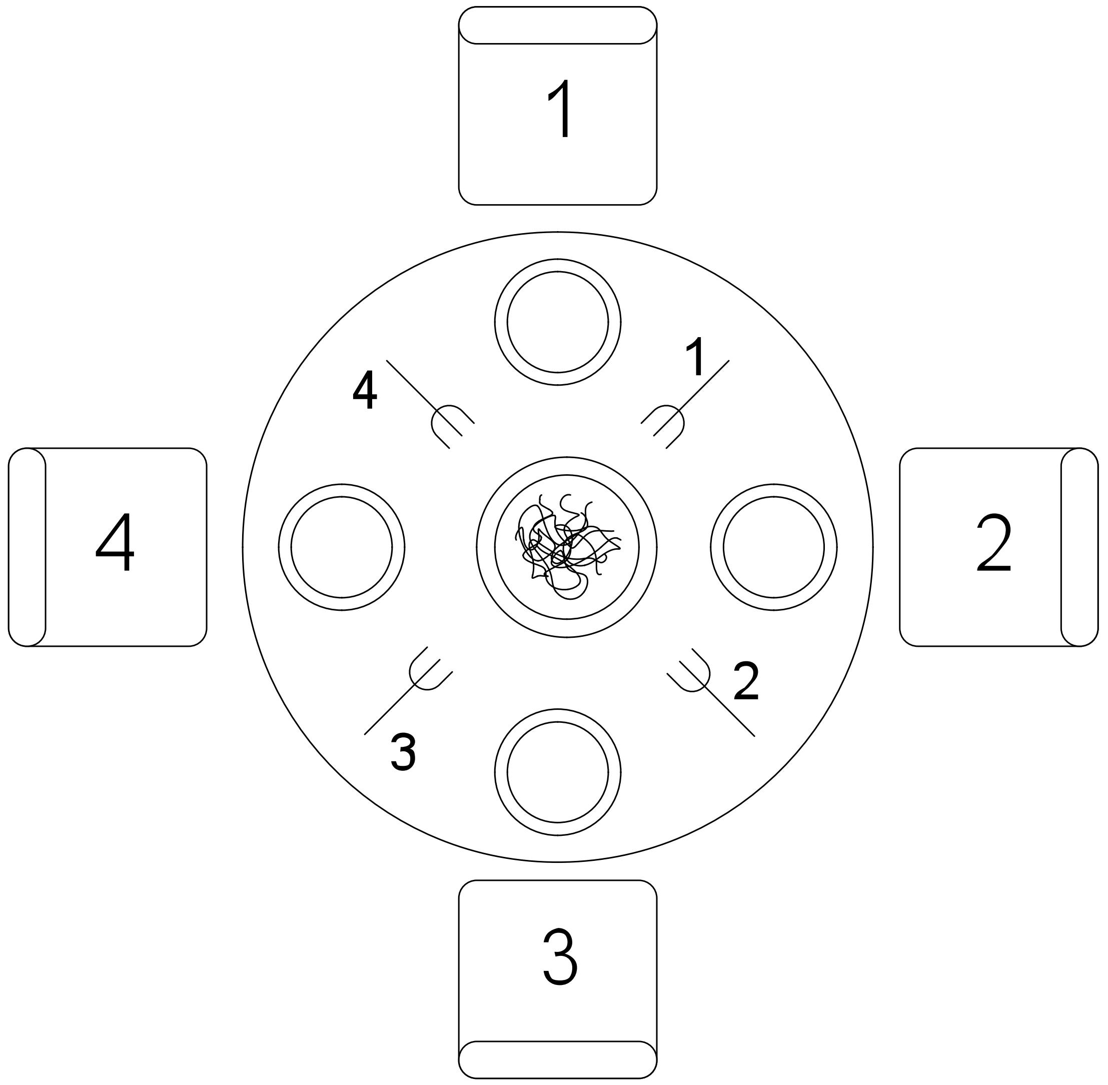
- A deadlock is a situation where two (or more) threads are waiting for the other to release a resource, thus neither will ever run.

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone." (Kansas Legislation)

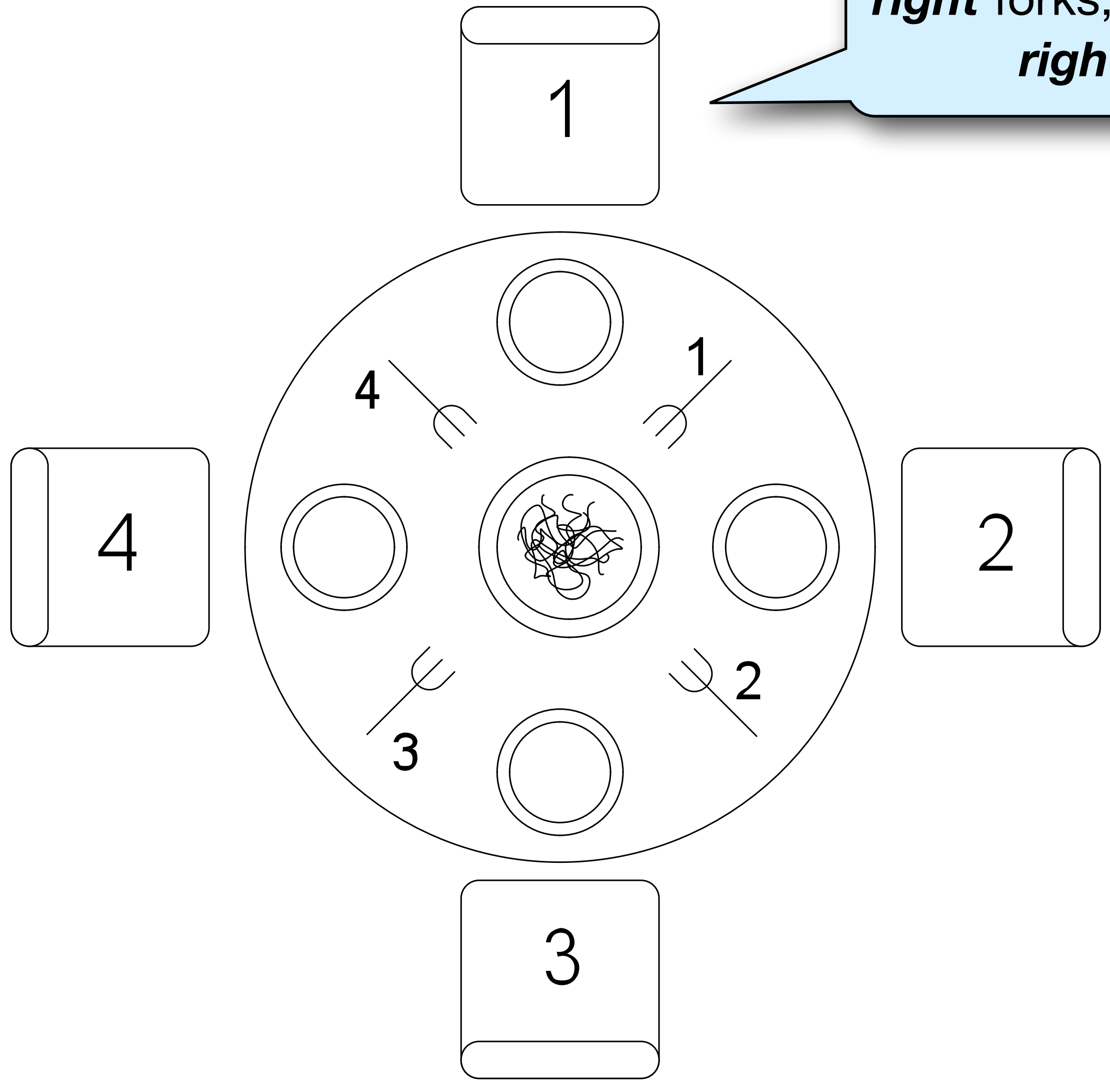
- The four necessary conditions for deadlocks:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

Deadlocks: Dining Philosophers

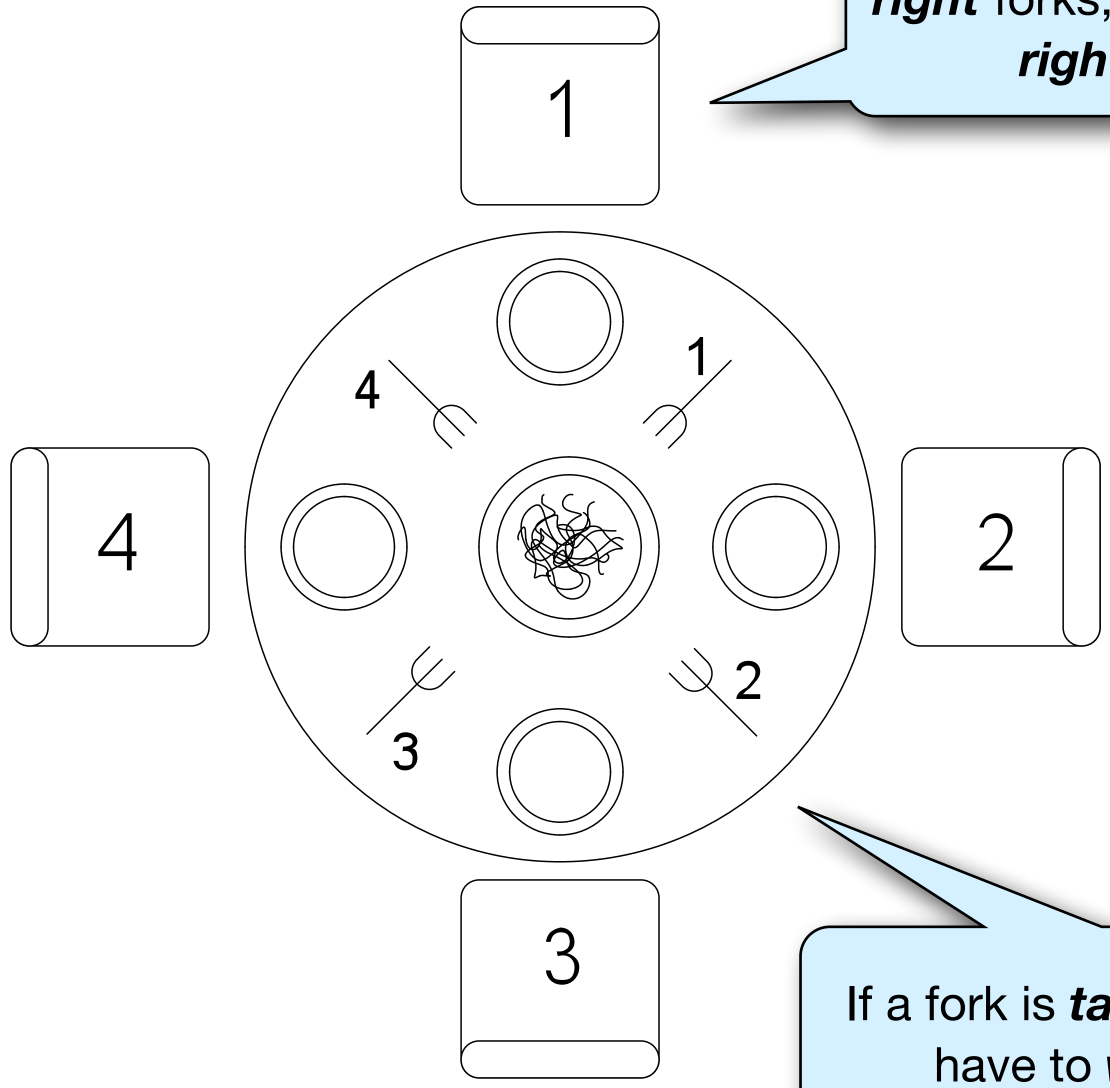


Deadlocks: Dining Philosophers



A philosopher picks up ***left*** and ***right*** forks, eats, put down ***left*** and ***right*** forks and thinks

Deadlocks: Dining Philosophers

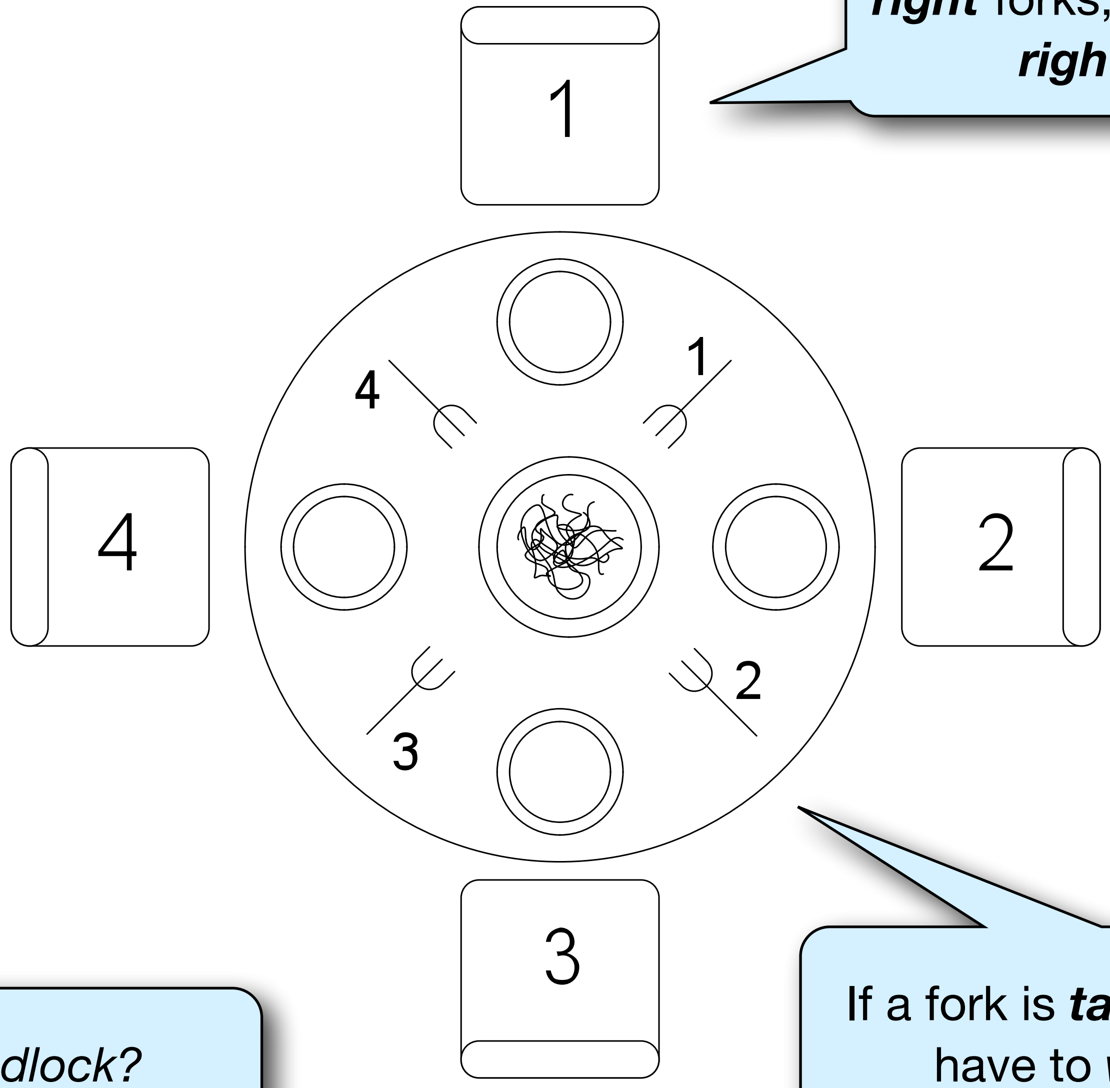


A philosopher picks up ***left*** and ***right*** forks, eats, put down ***left*** and ***right*** forks and thinks

If a fork is ***taken*** the philosopher will have to ***wait*** for it to be ***free***

AARHUS UNIVERSITY

Deadlocks: Dining Philosophers

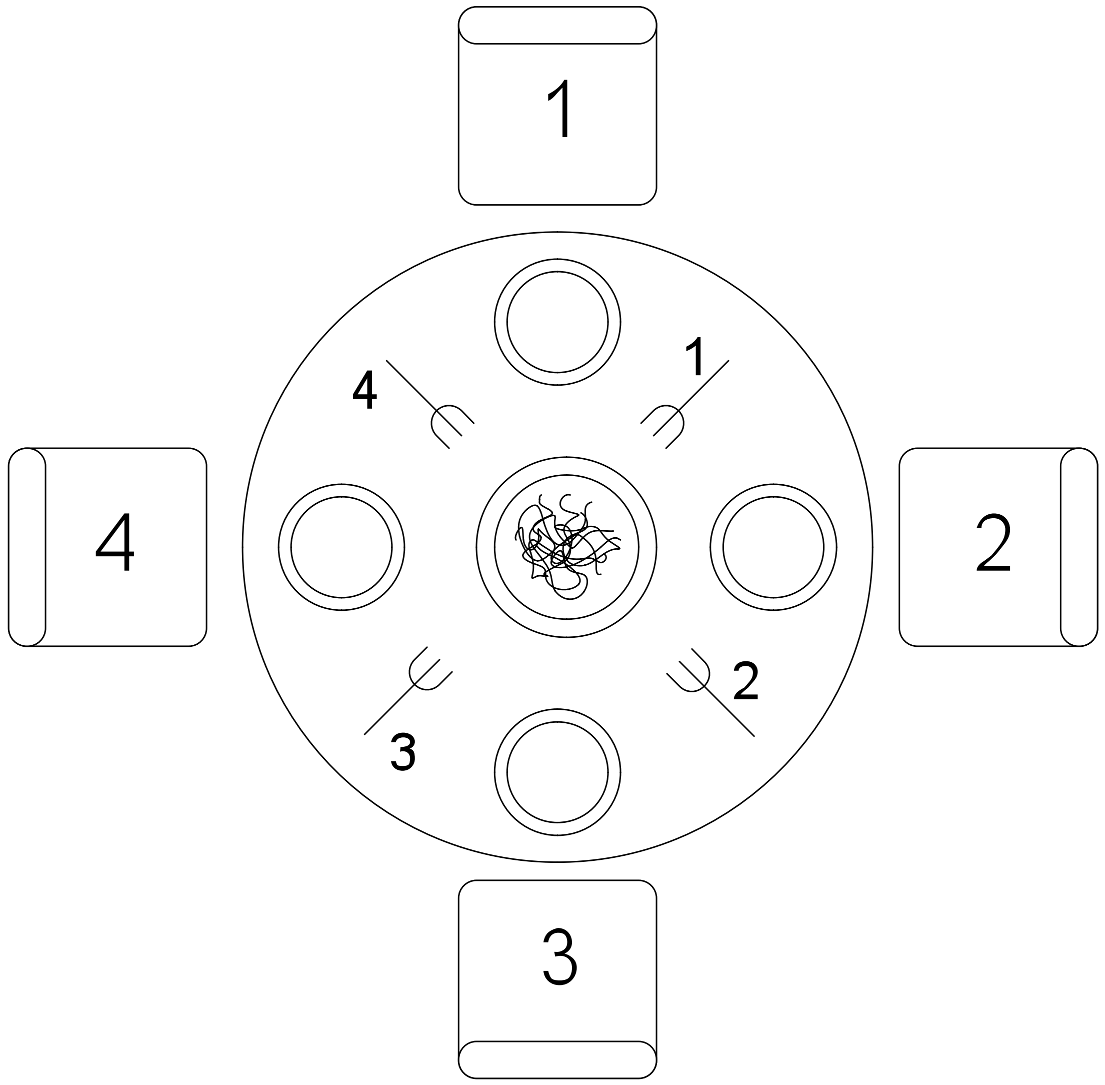


A philosopher picks up ***left*** and ***right*** forks, eats, put down ***left*** and ***right*** forks and thinks

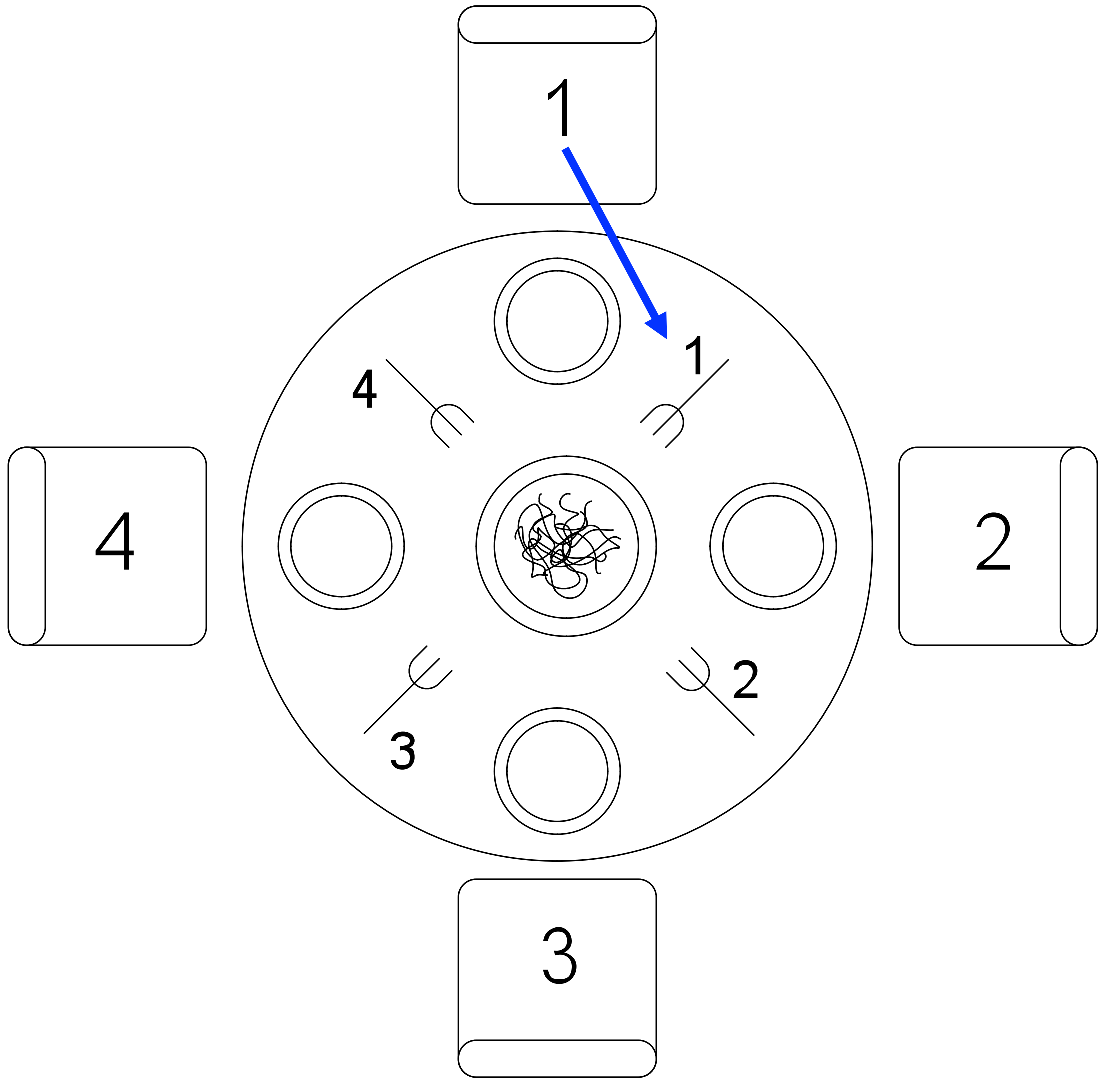
If a fork is ***taken*** the philosopher will have to ***wait*** for it to be ***free***

Can this system deadlock? How?

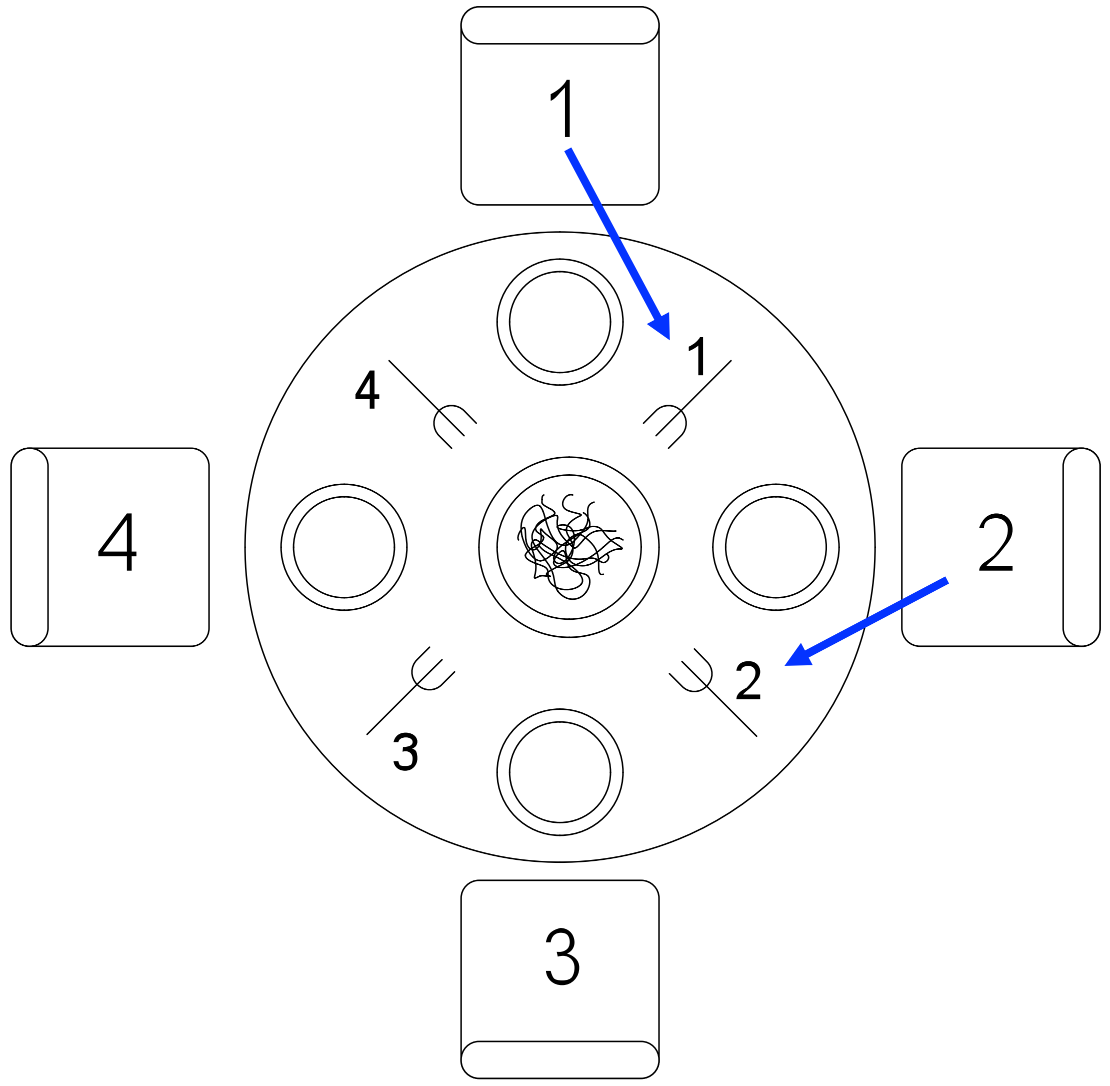
Deadlocks: Dining Philosophers



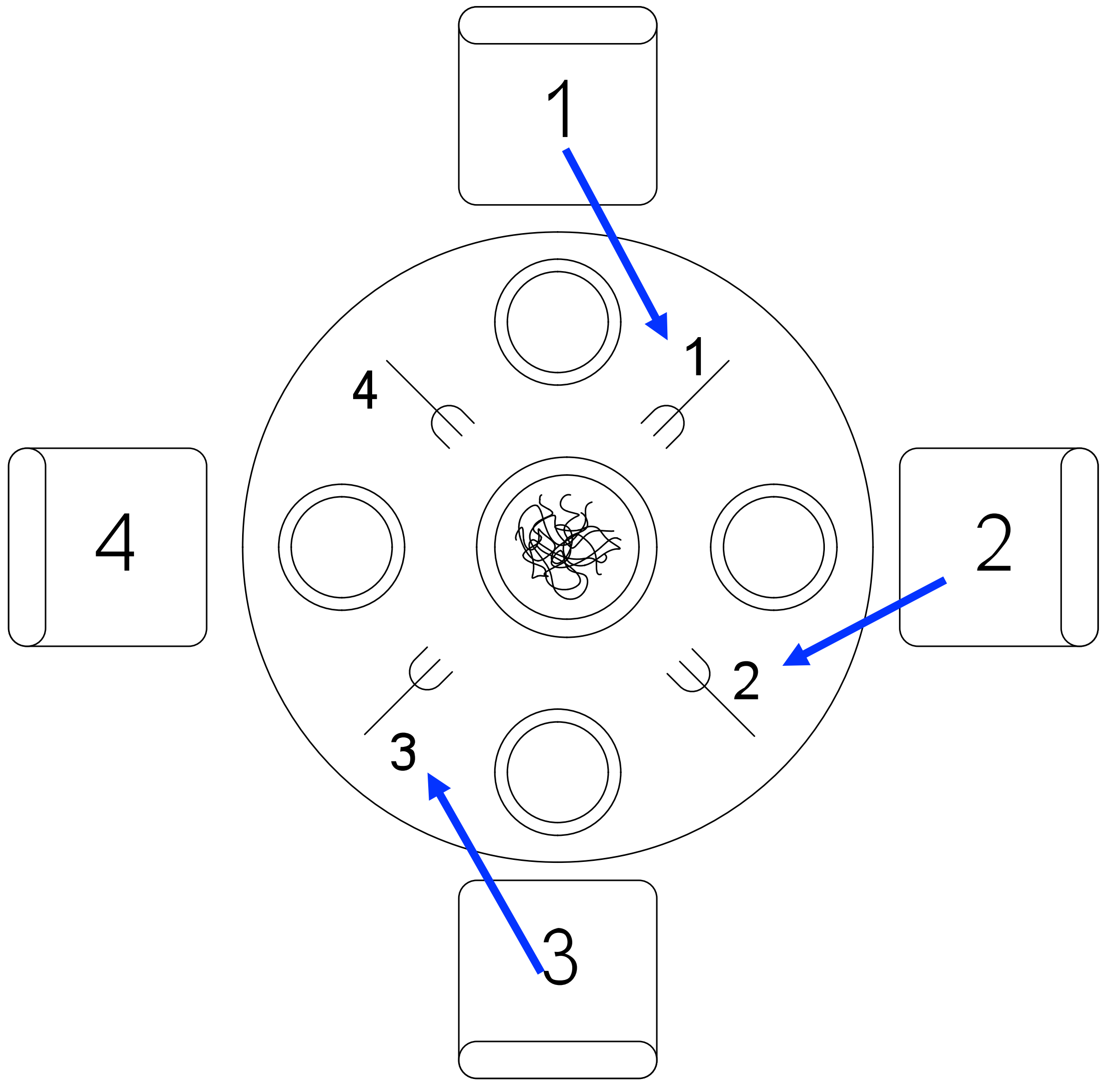
Deadlocks: Dining Philosophers



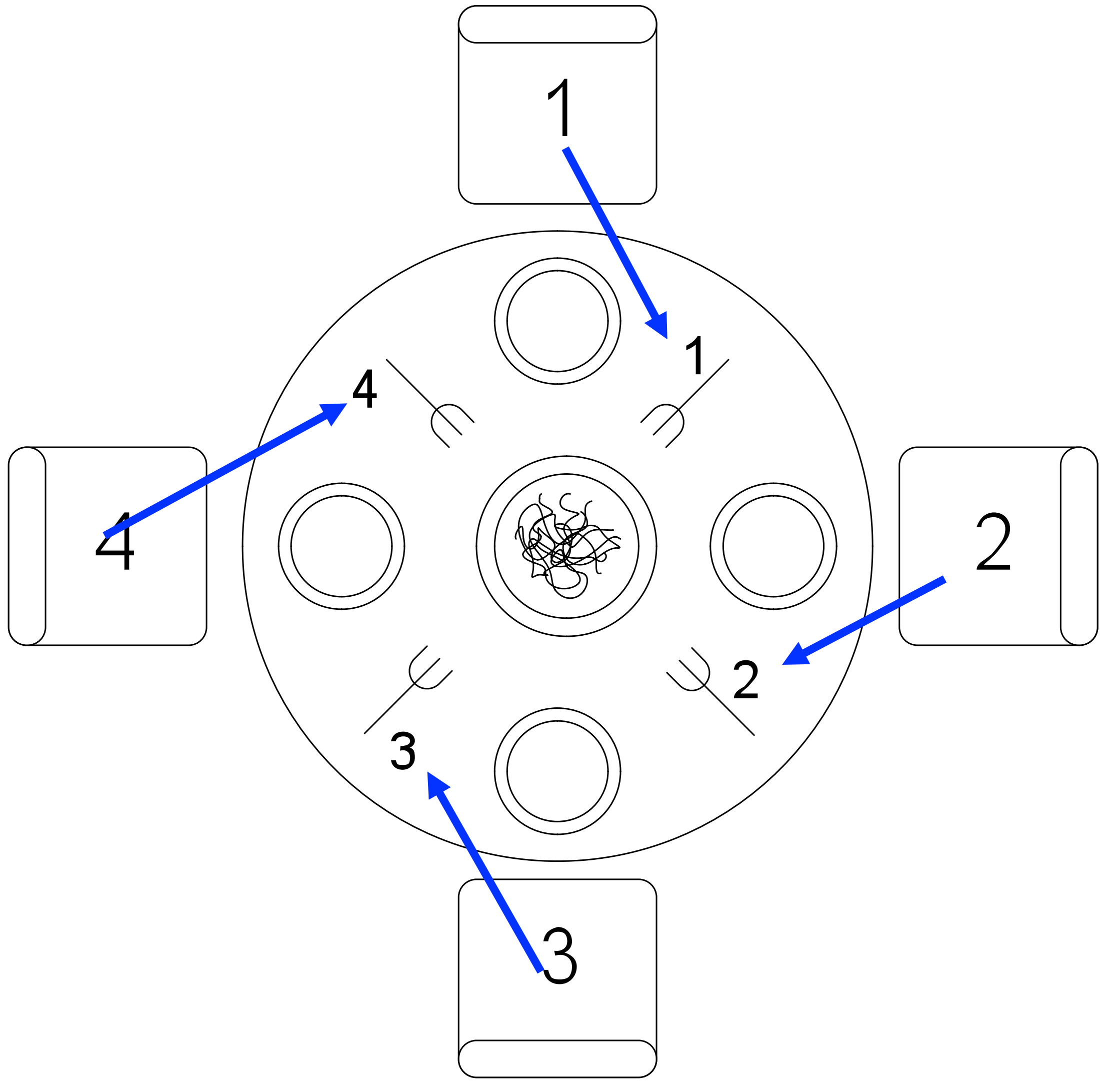
Deadlocks: Dining Philosophers



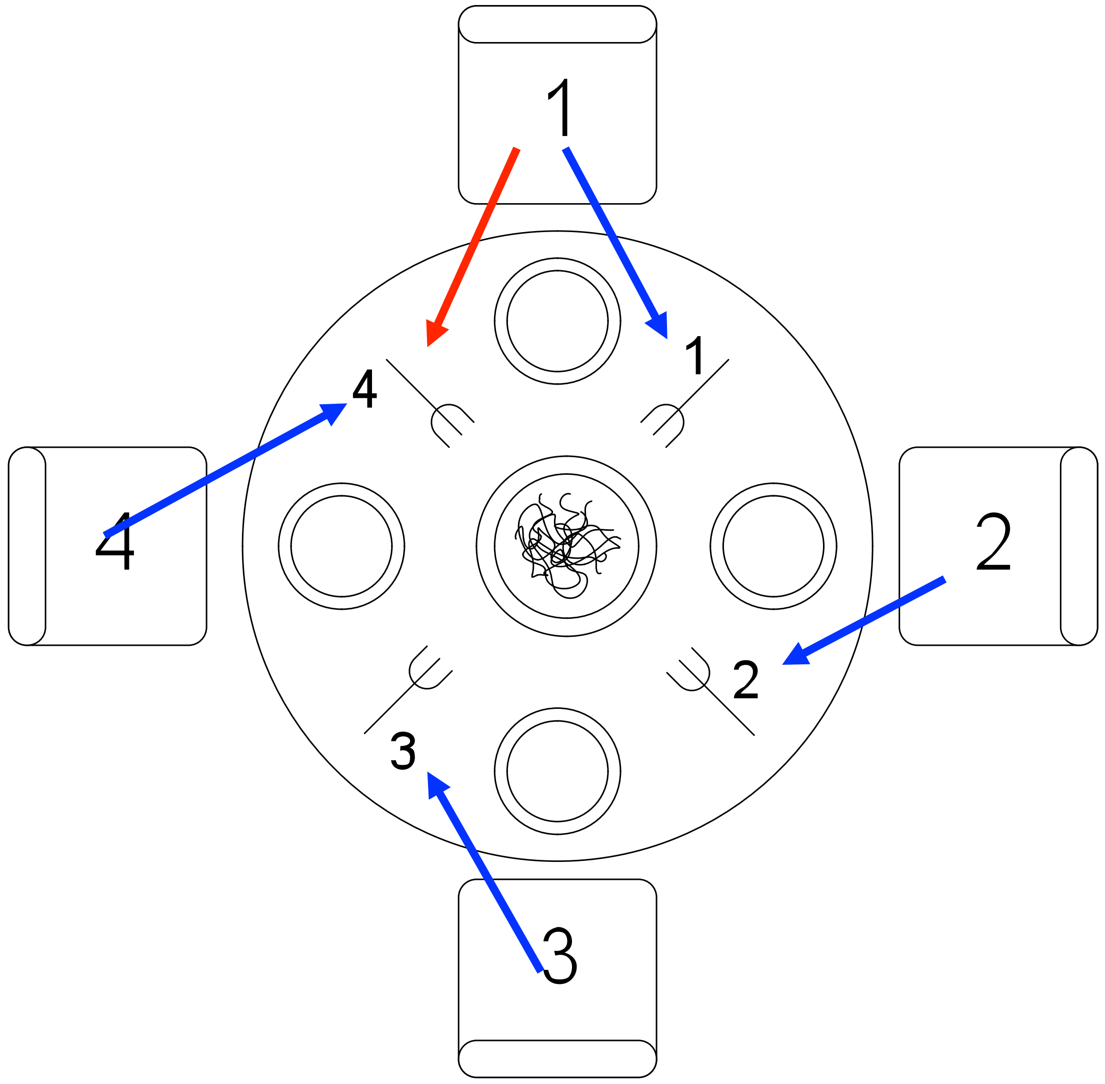
Deadlocks: Dining Philosophers



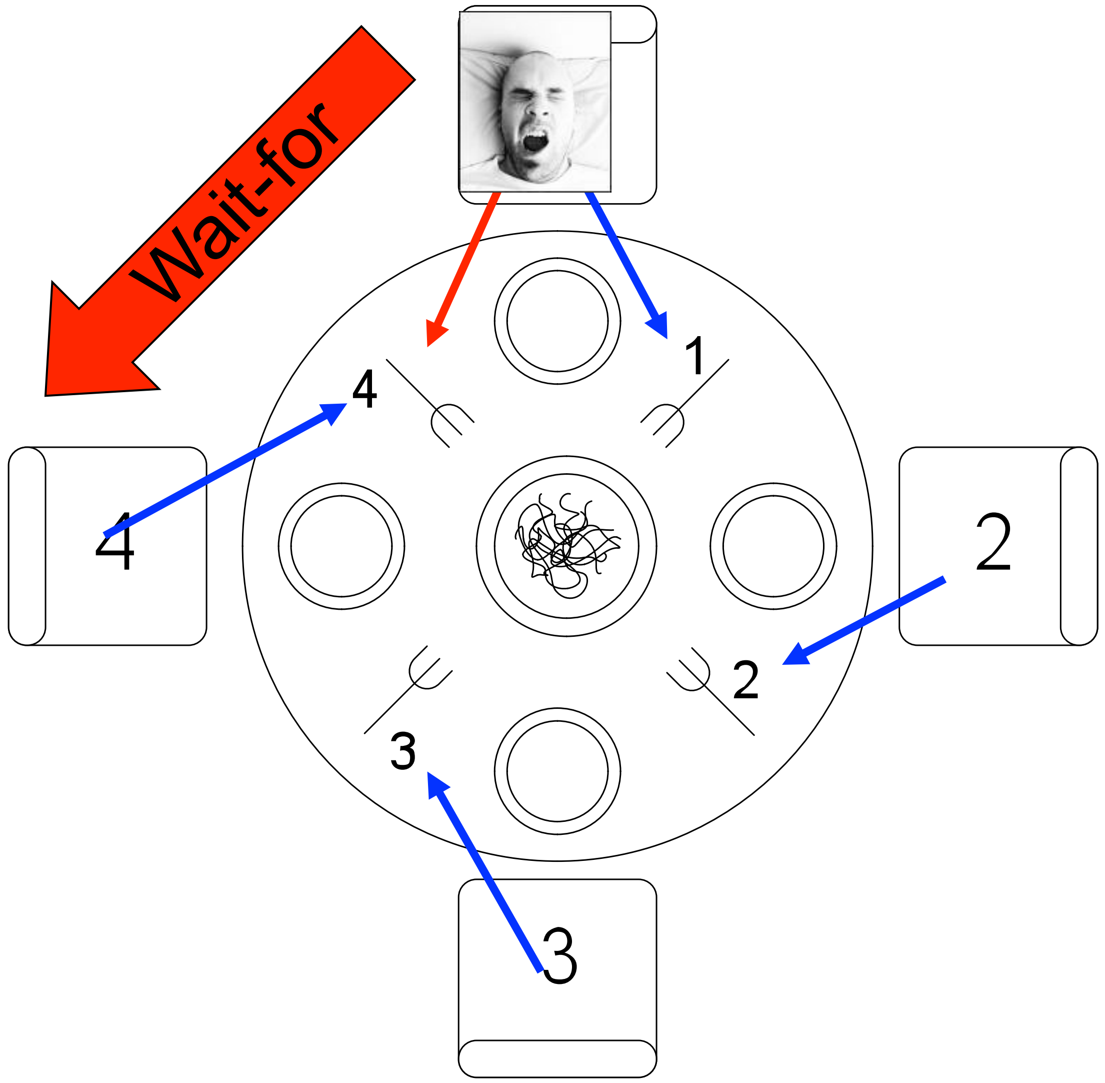
Deadlocks: Dining Philosophers



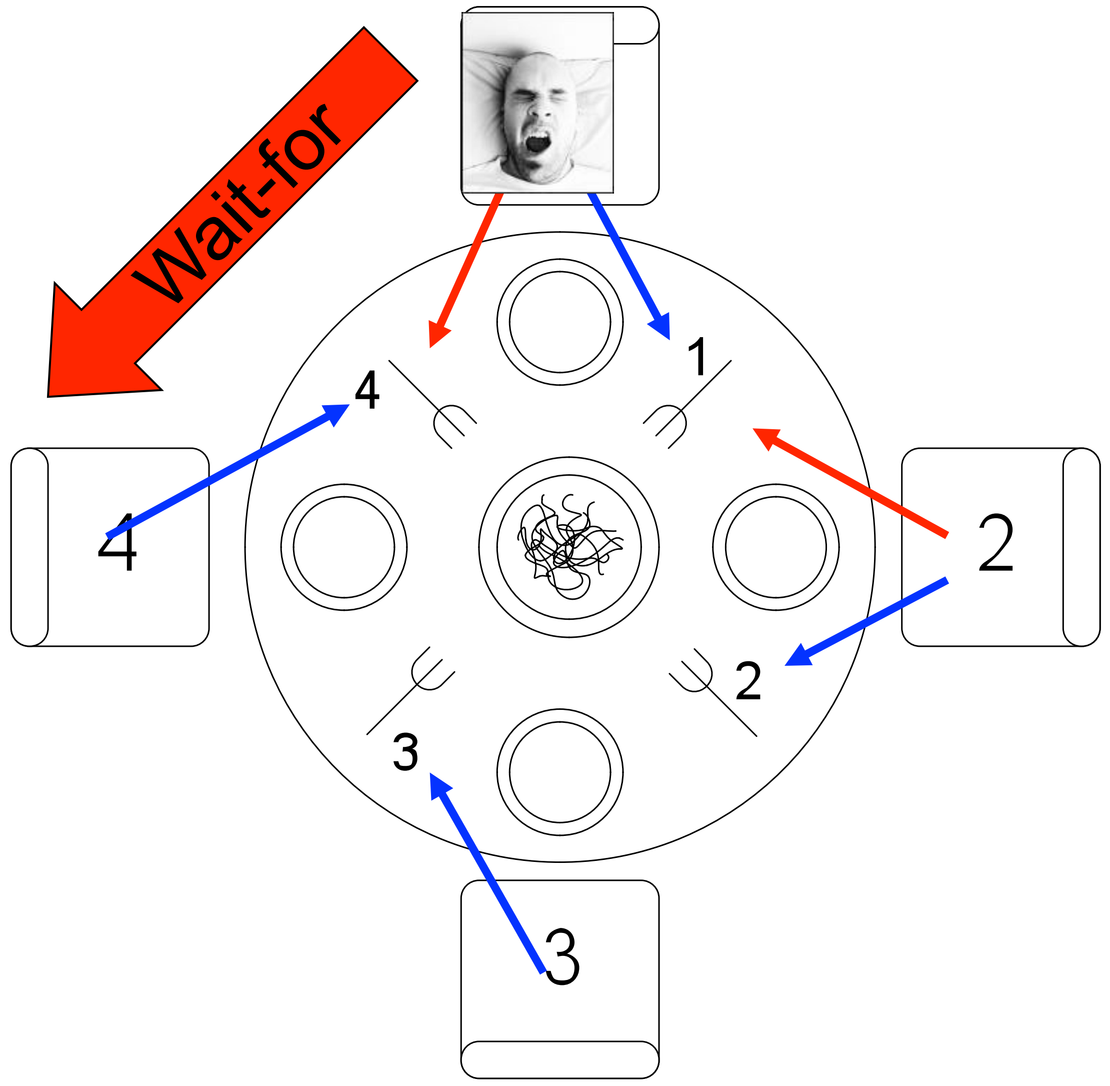
Deadlocks: Dining Philosophers



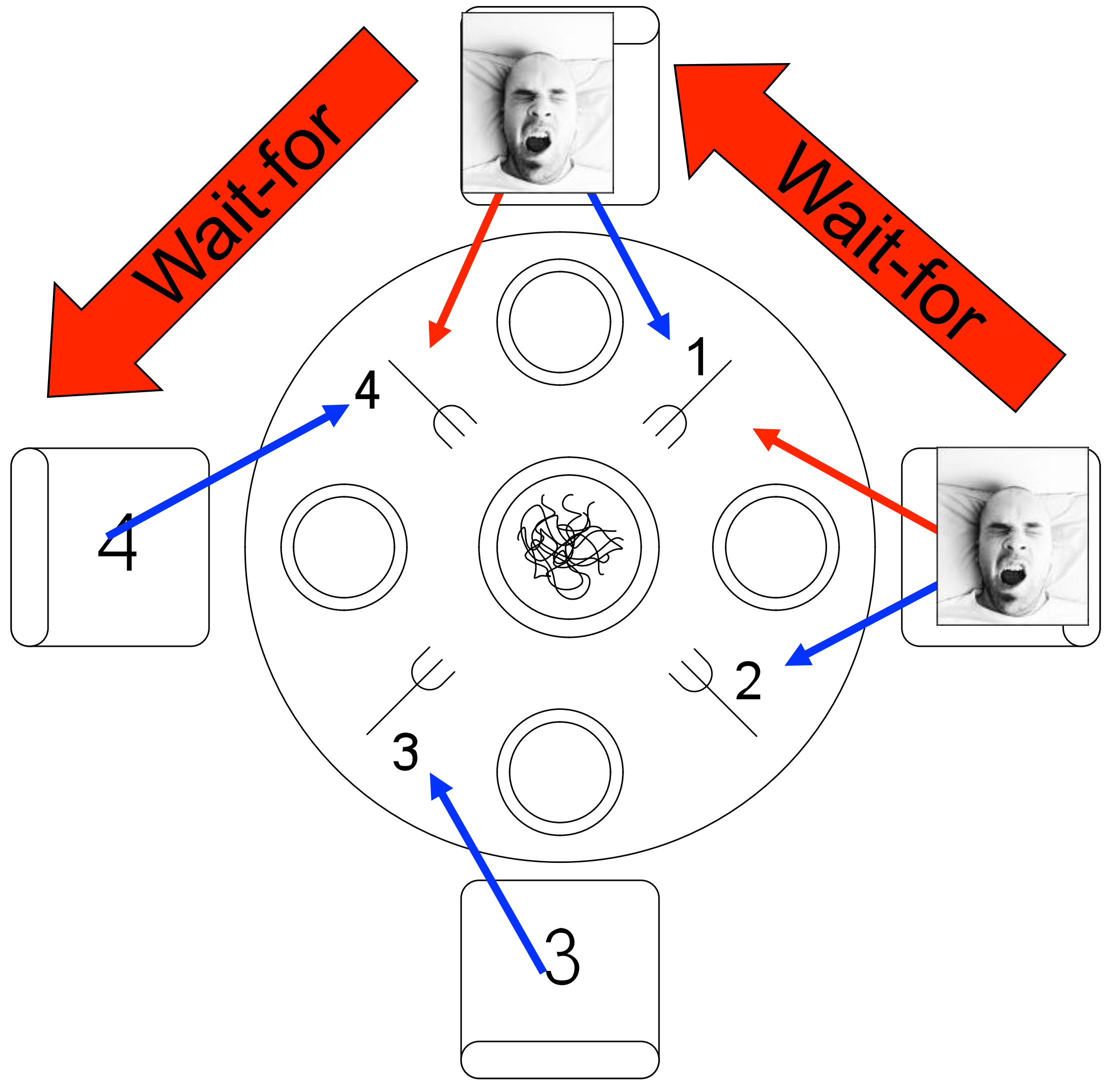
Deadlocks: Dining Philosophers



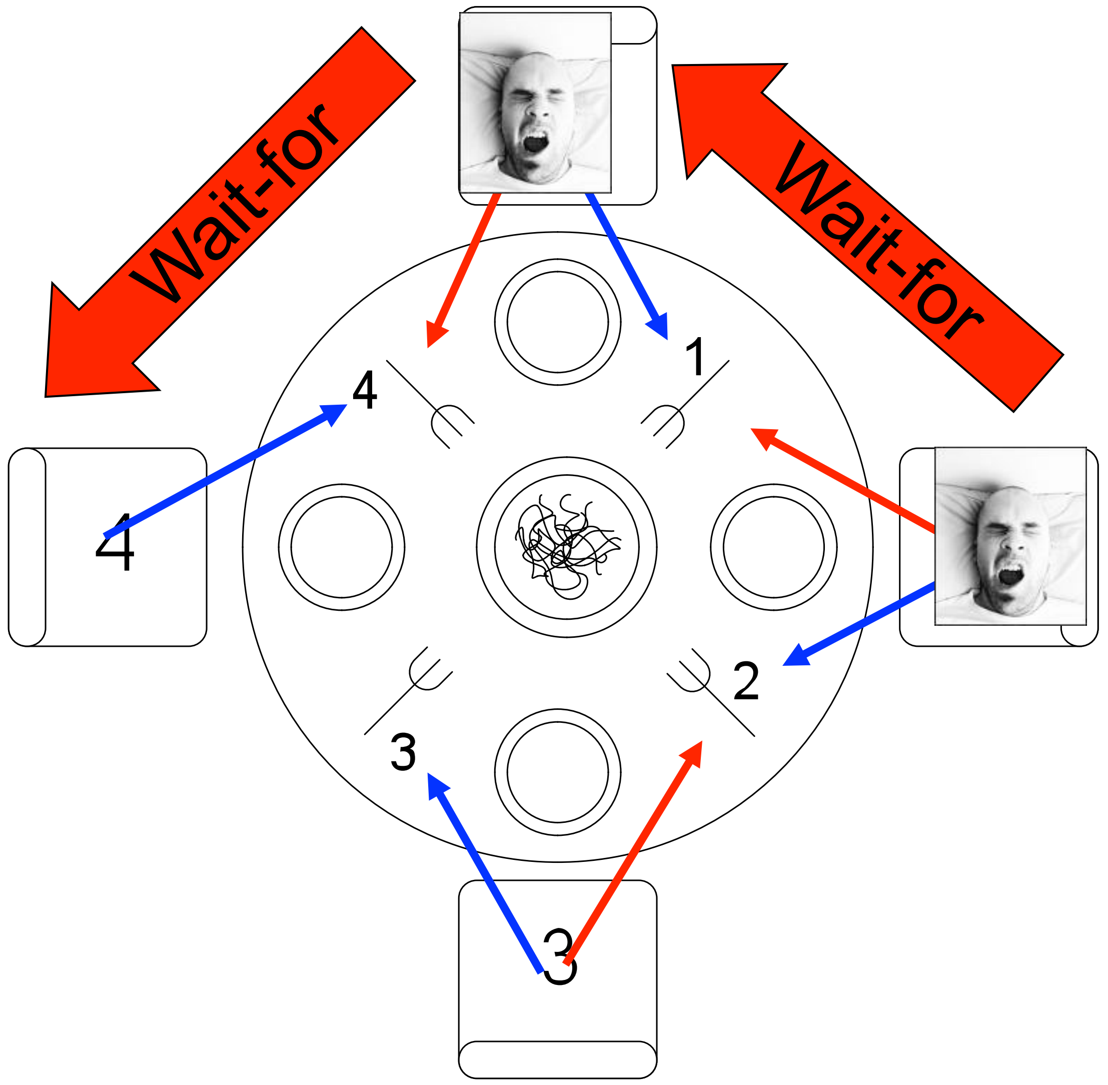
Deadlocks: Dining Philosophers



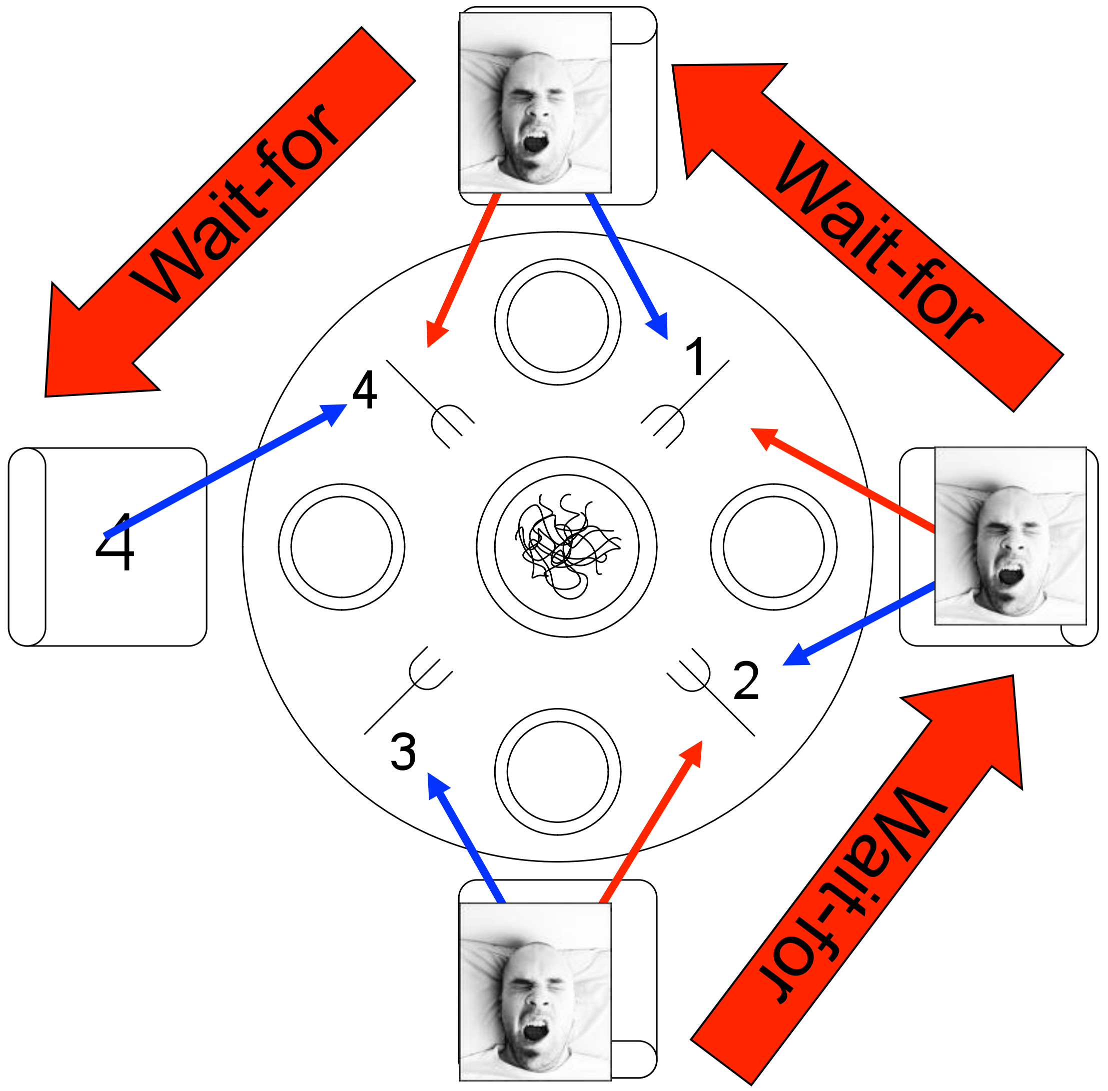
Deadlocks: Dining Philosophers



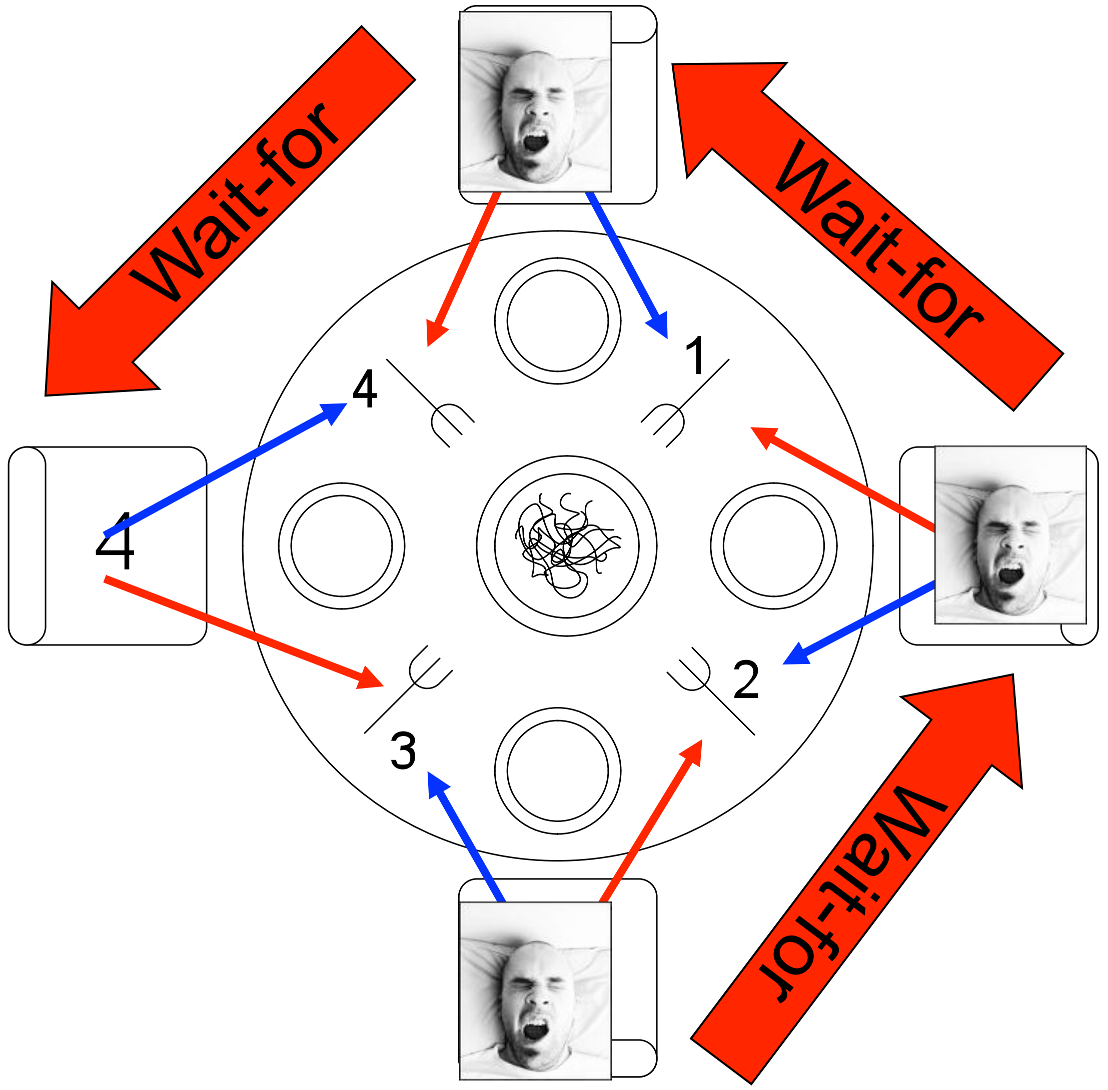
Deadlocks: Dining Philosophers



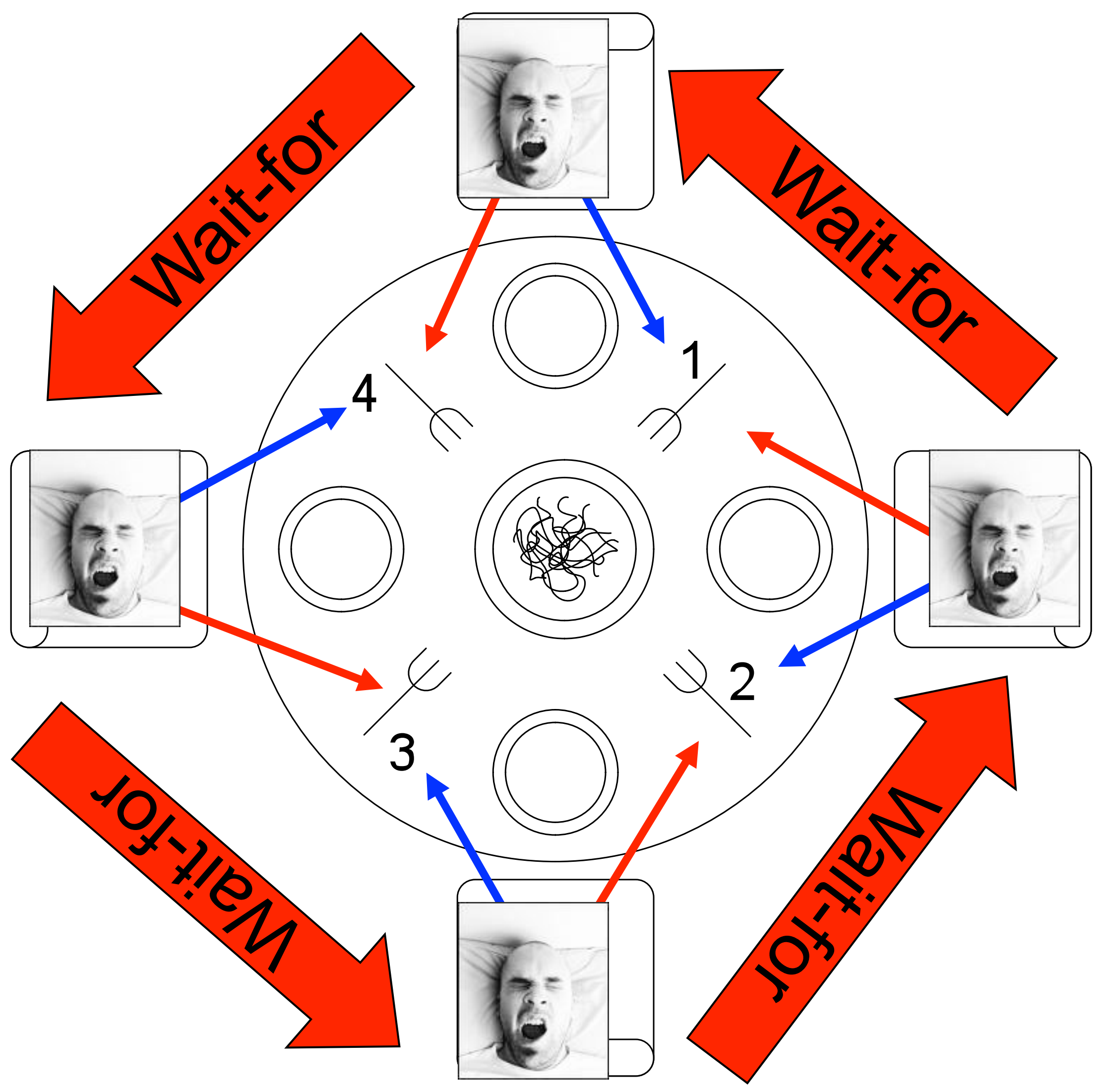
Deadlocks: Dining Philosophers



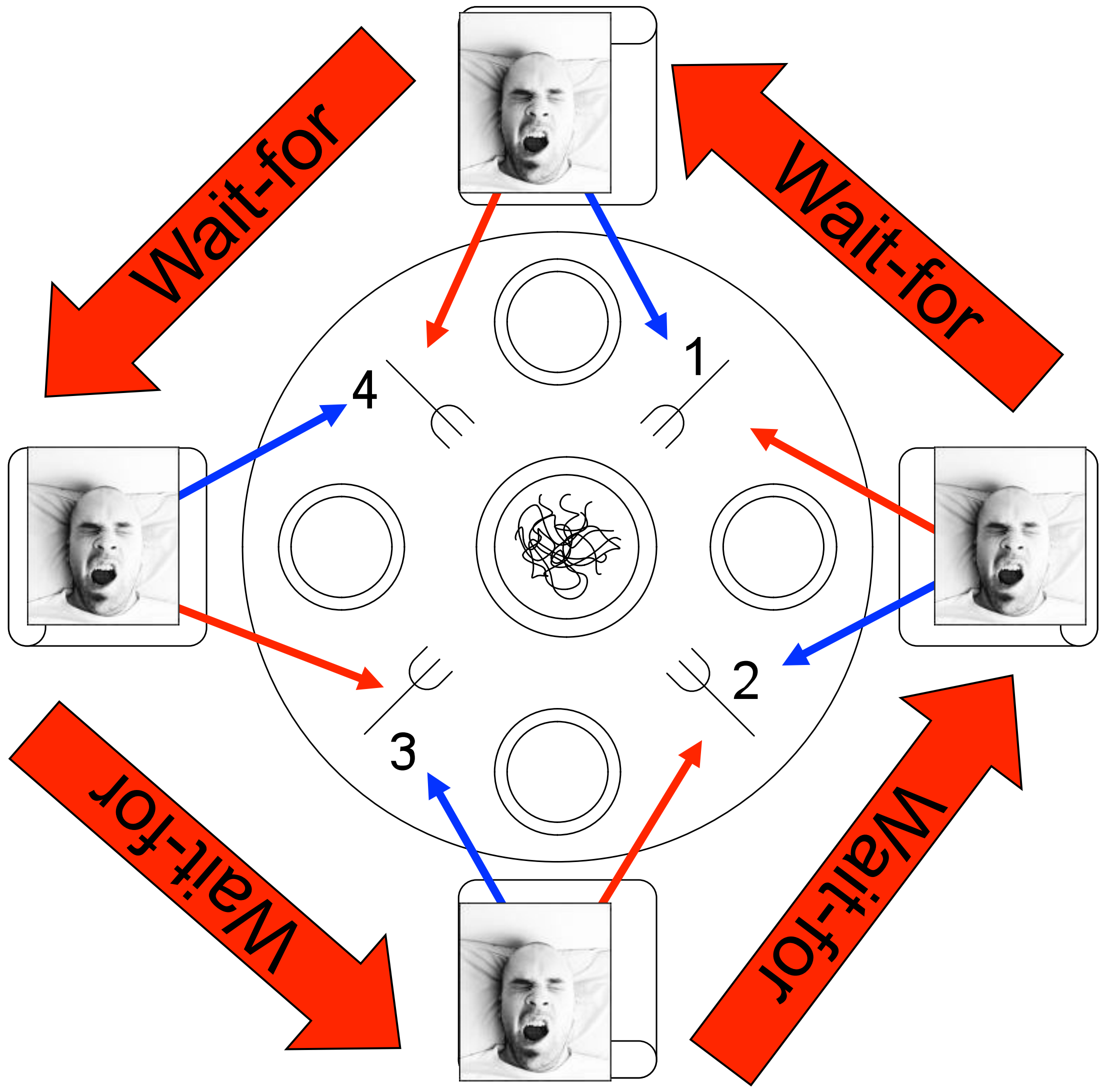
Deadlocks: Dining Philosophers



Deadlocks: Dining Philosophers



Deadlocks: Dining Philosophers



Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

  1?

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

  1? No, two people can't use the same fork at the same time

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

1? No, two people can't use the same fork at the same time

2?

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

1? No, two people can't use the same fork at the same time

2? No, you need two forks to eat spaghetti

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

1? No, two people can't use the same fork at the same time

2? No, you need two forks to eat spaghetti

3?

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

1? No, two people can't use the same fork at the same time

2? No, you need two forks to eat spaghetti

3? No…philosophers don't steal forks from each other

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

1? No, two people can't use the same fork at the same time

2? No, you need two forks to eat spaghetti

3? No…philosophers don't steal forks from each other

4?

Deadlocks: Solutions

- The solution to deadlocks in general is to remove one of the four necessary conditions:

1. *Mutual exclusion*	The resource can only be held by one process at a time
2. *Hold-and-wait*	Process already holding resources may request other resources
3. *No preemption*	No resource can be forcibly removed from its owner process
4. *Circular wait condition*	A cycle p0, p1,…pn, p0 exists where pi waits for a resource that pi+1 holds

- Applied to the Dining Philosopher's problem: Can we remove…

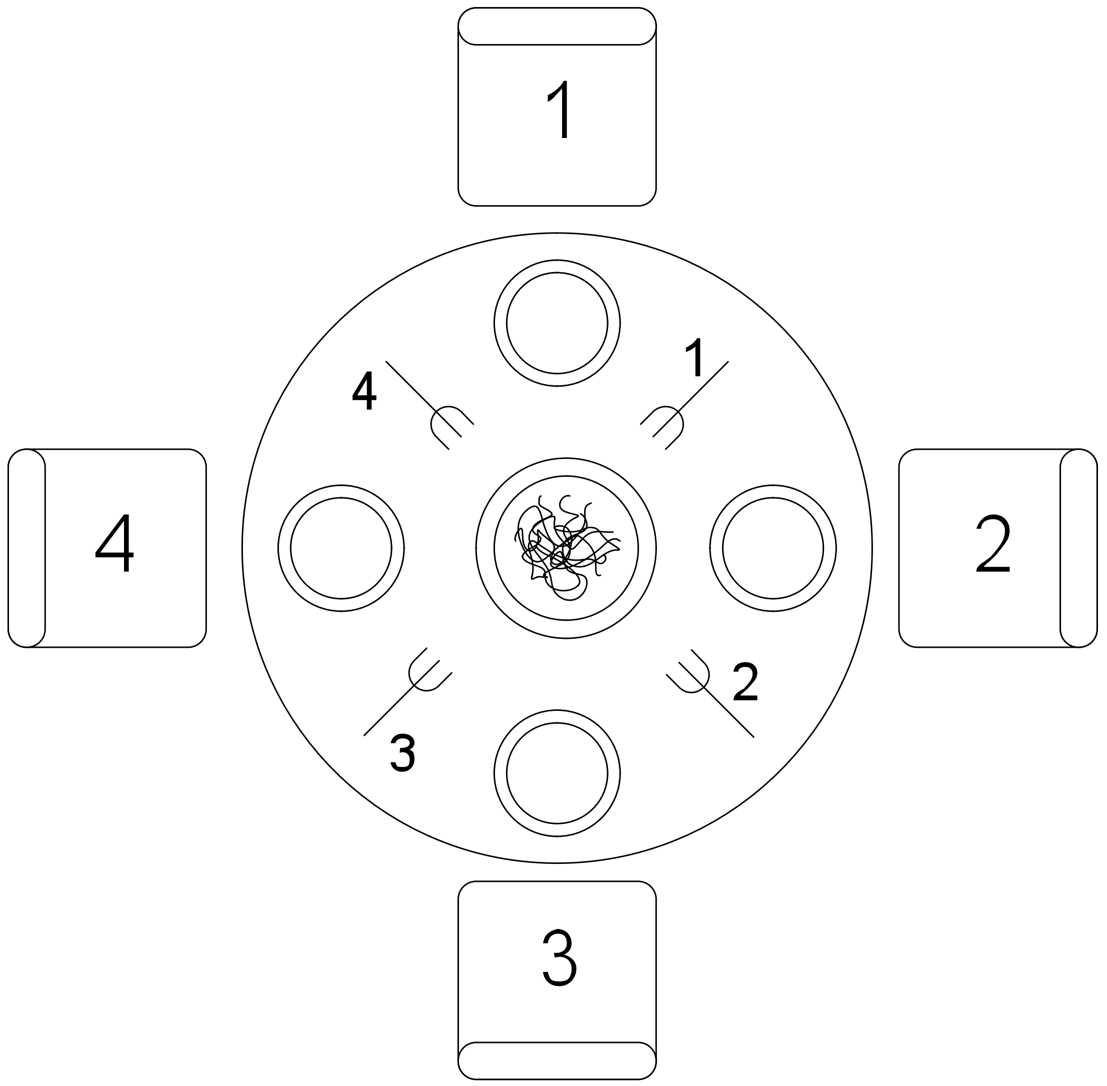
1? No, two people can't use the same fork at the same time

2? No, you need two forks to eat spaghetti

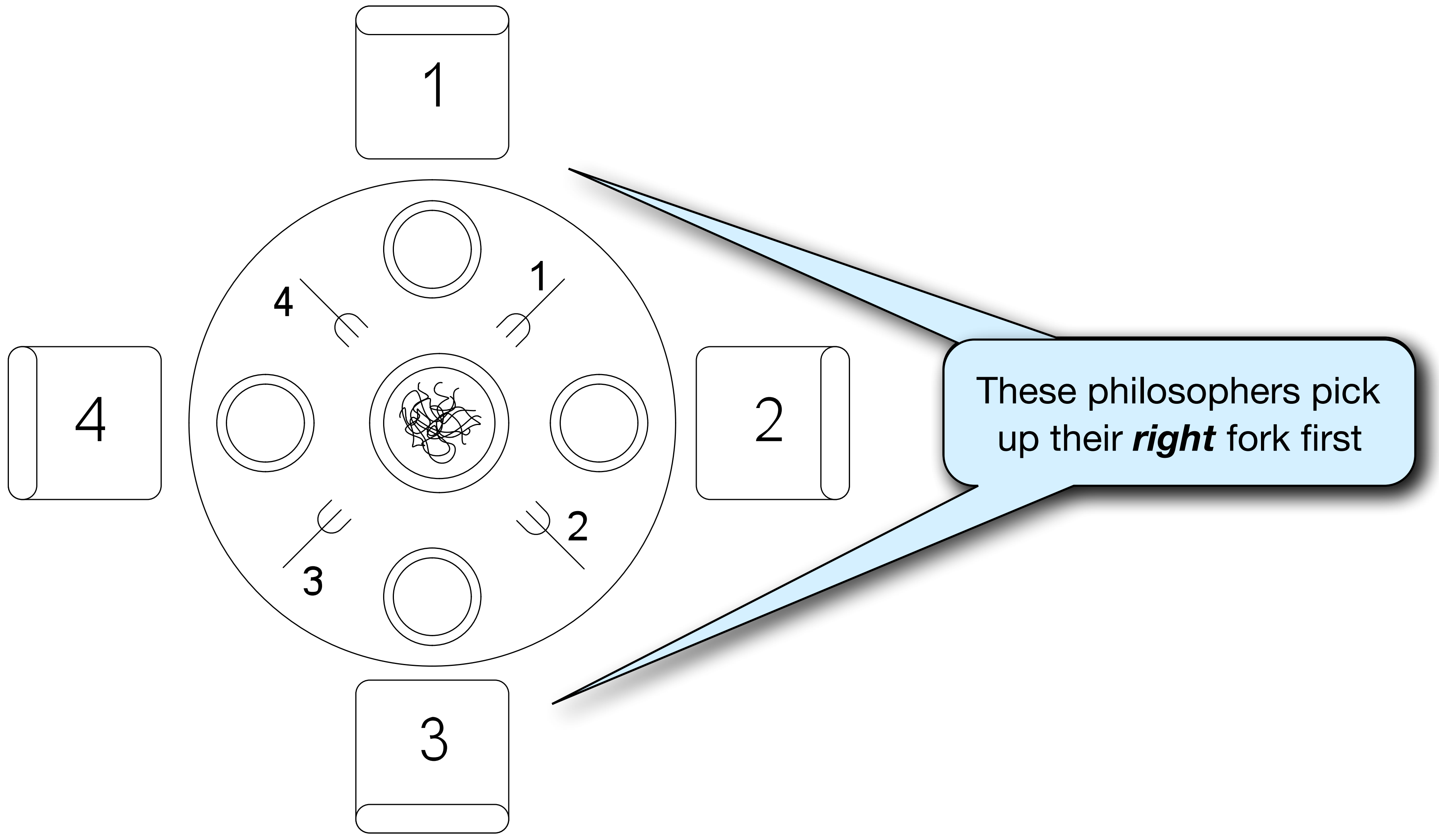
3? No…philosophers don't steal forks from each other

4? Yes…we can break the cycle!

Dining Philosophers - solution

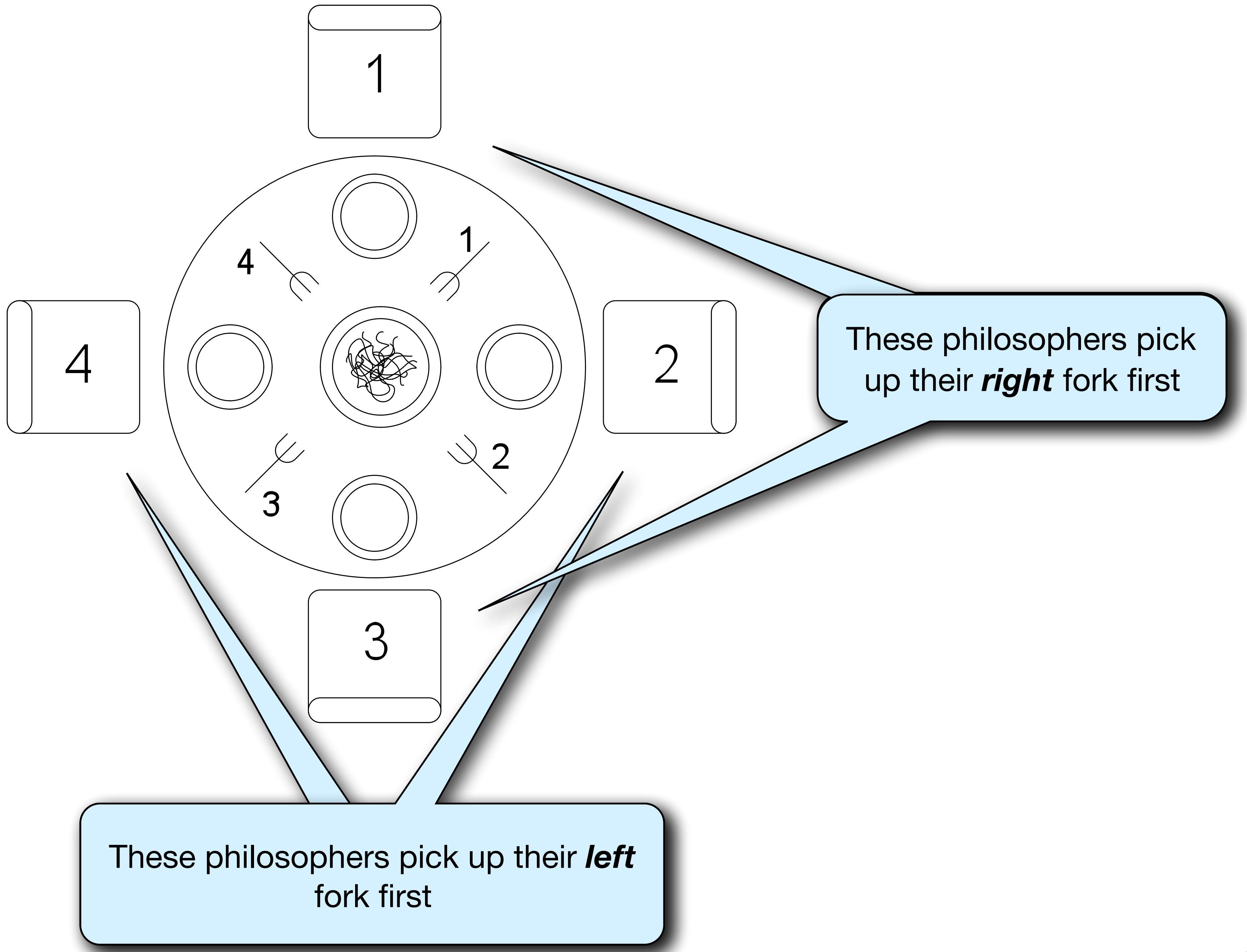


Dining Philosophers - solution

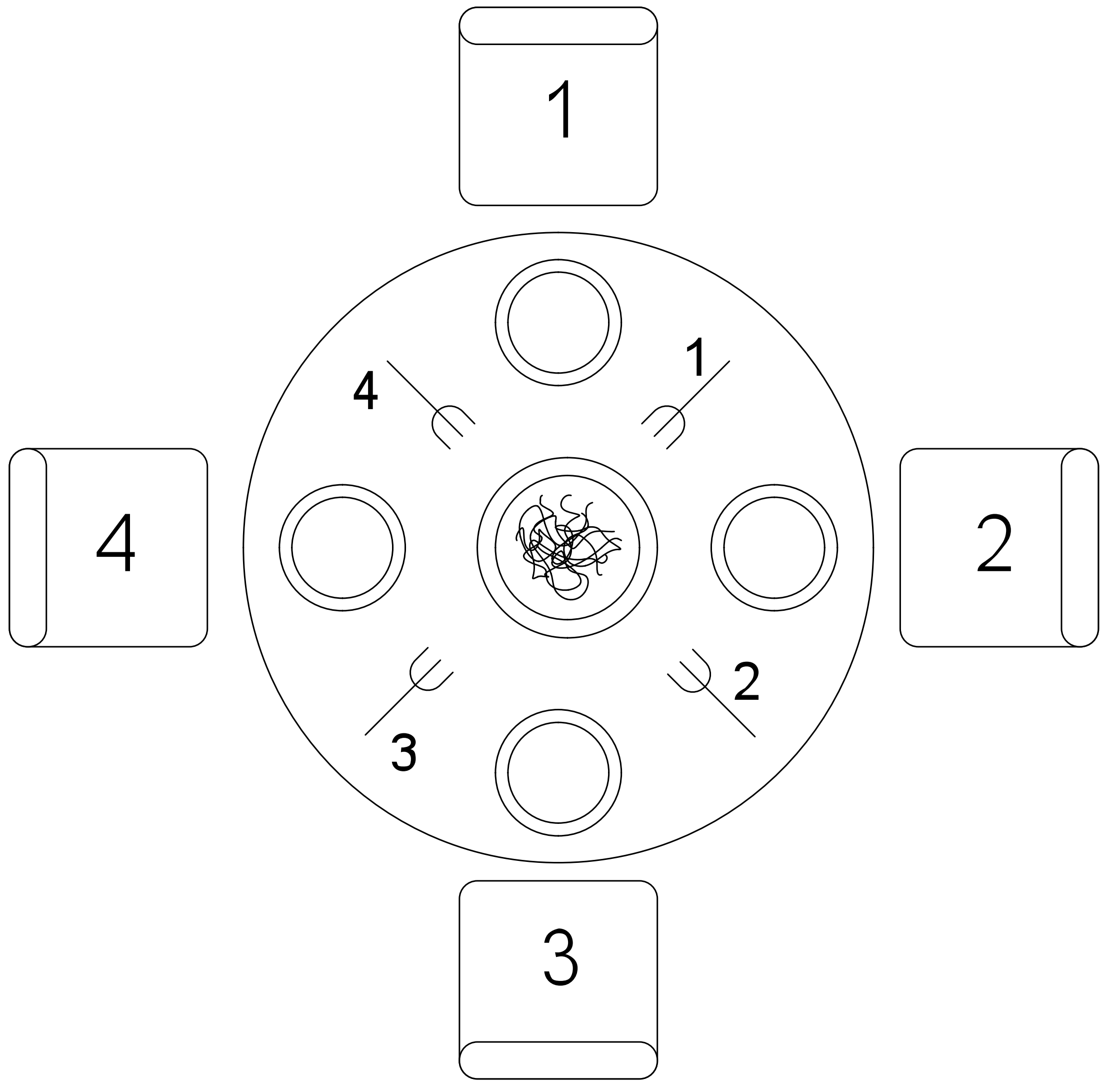


These philosophers pick up their ***right*** fork first

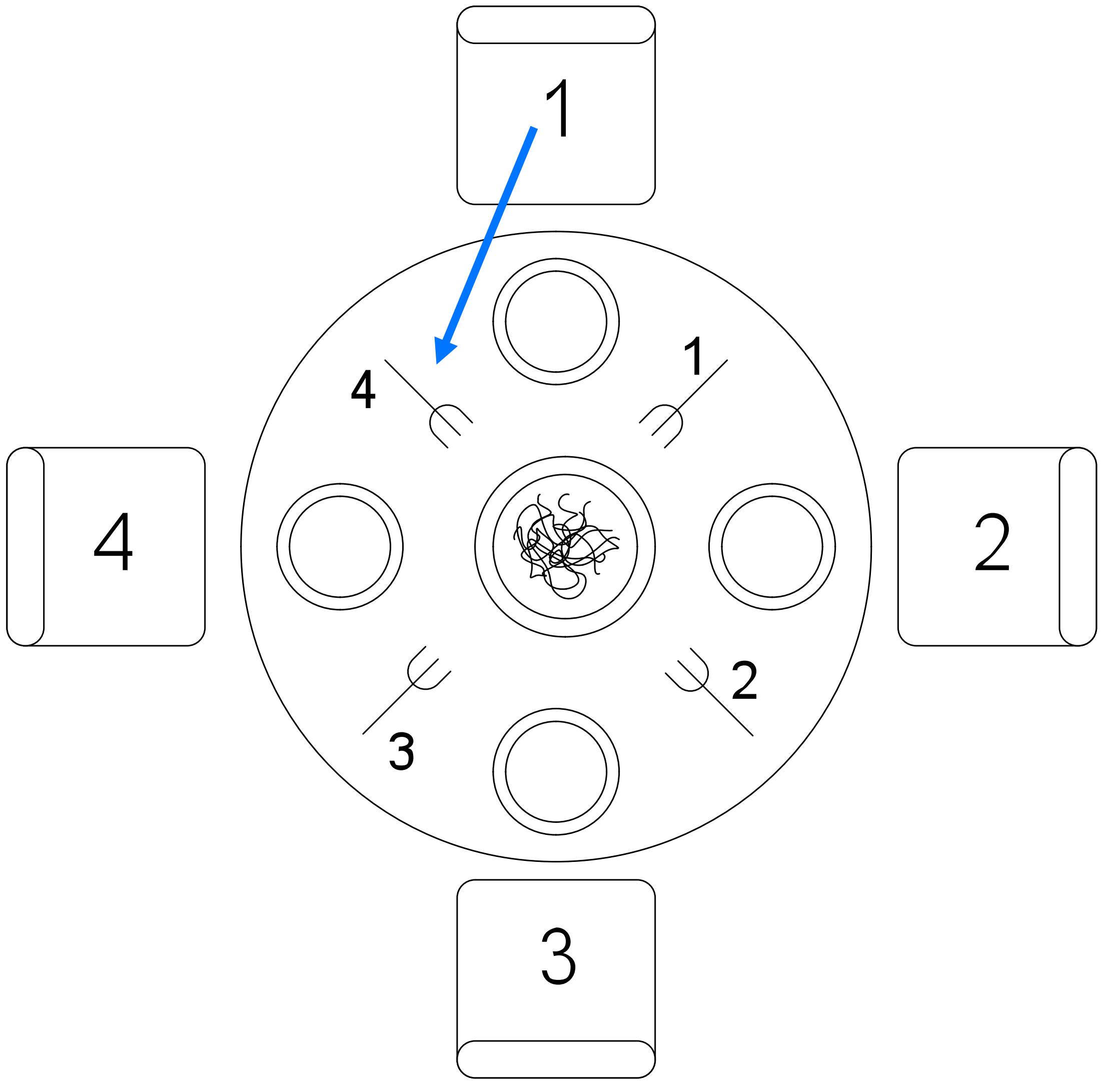
Dining Philosophers - solution


1
4
1
4
2
2
3
3
These philosophers pick up their ***right*** fork first
These philosophers pick up their ***left*** fork first

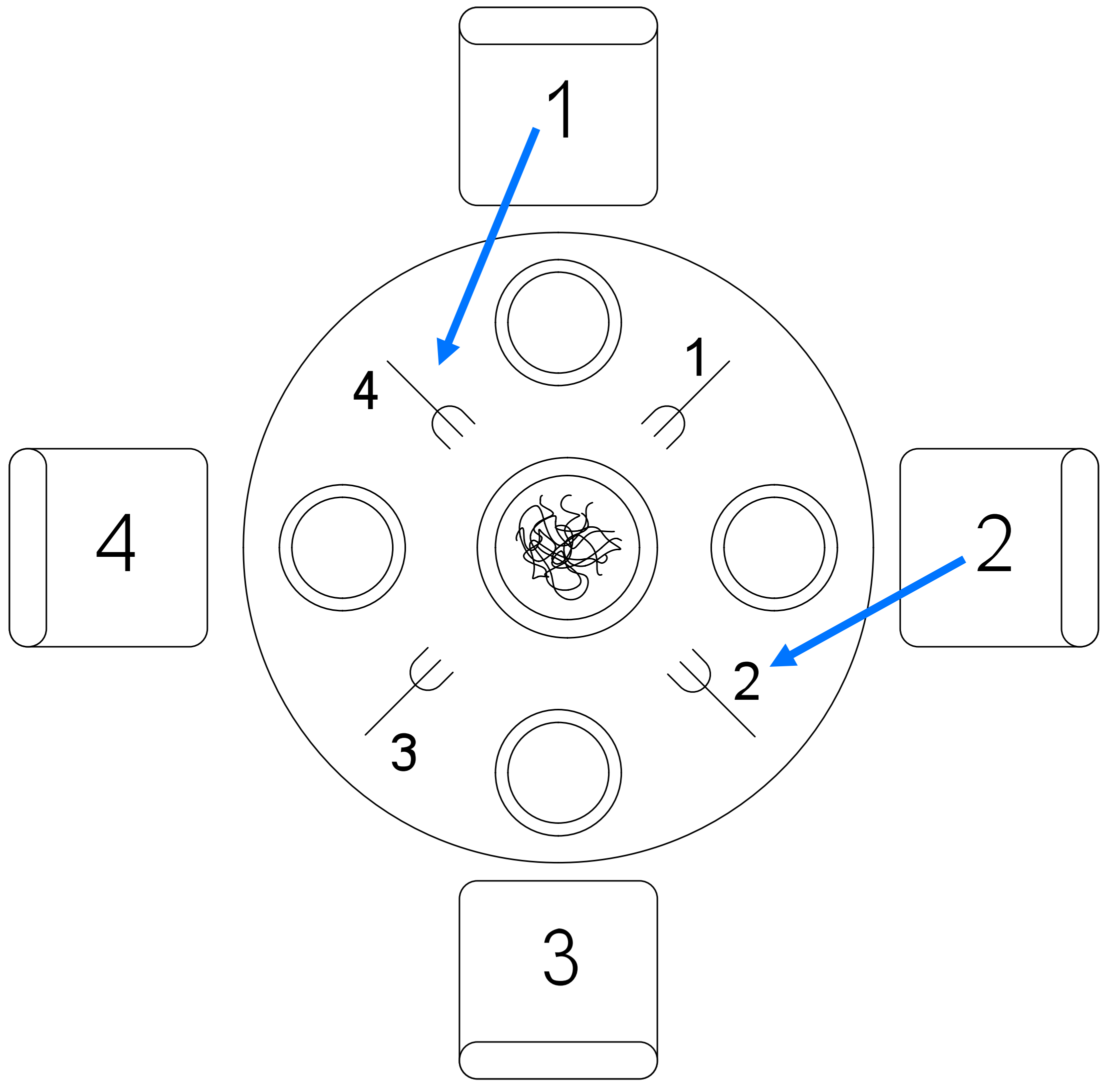
Deadlocks: Dining Philosophers - Solution



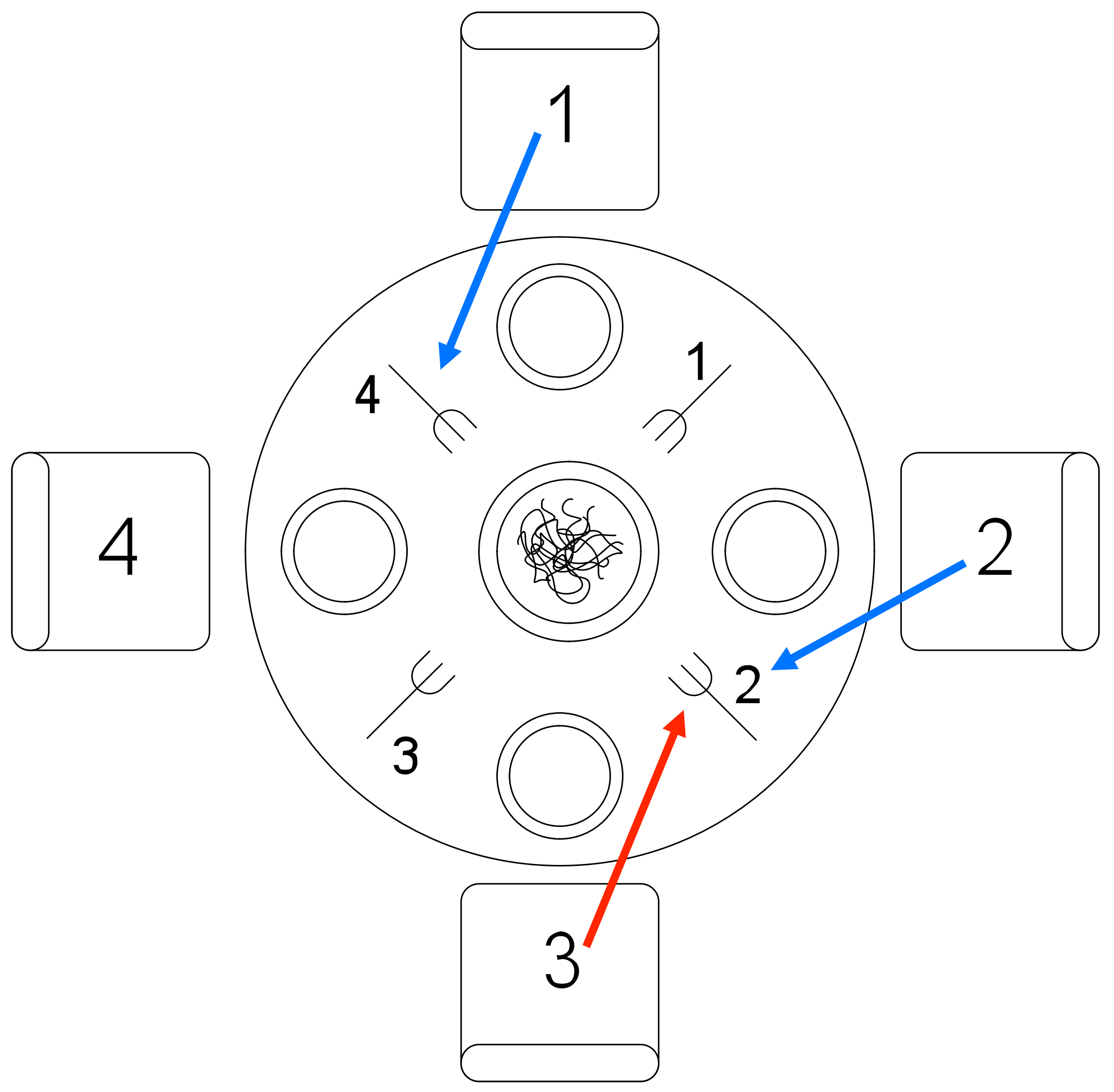
Deadlocks: Dining Philosophers - Solution



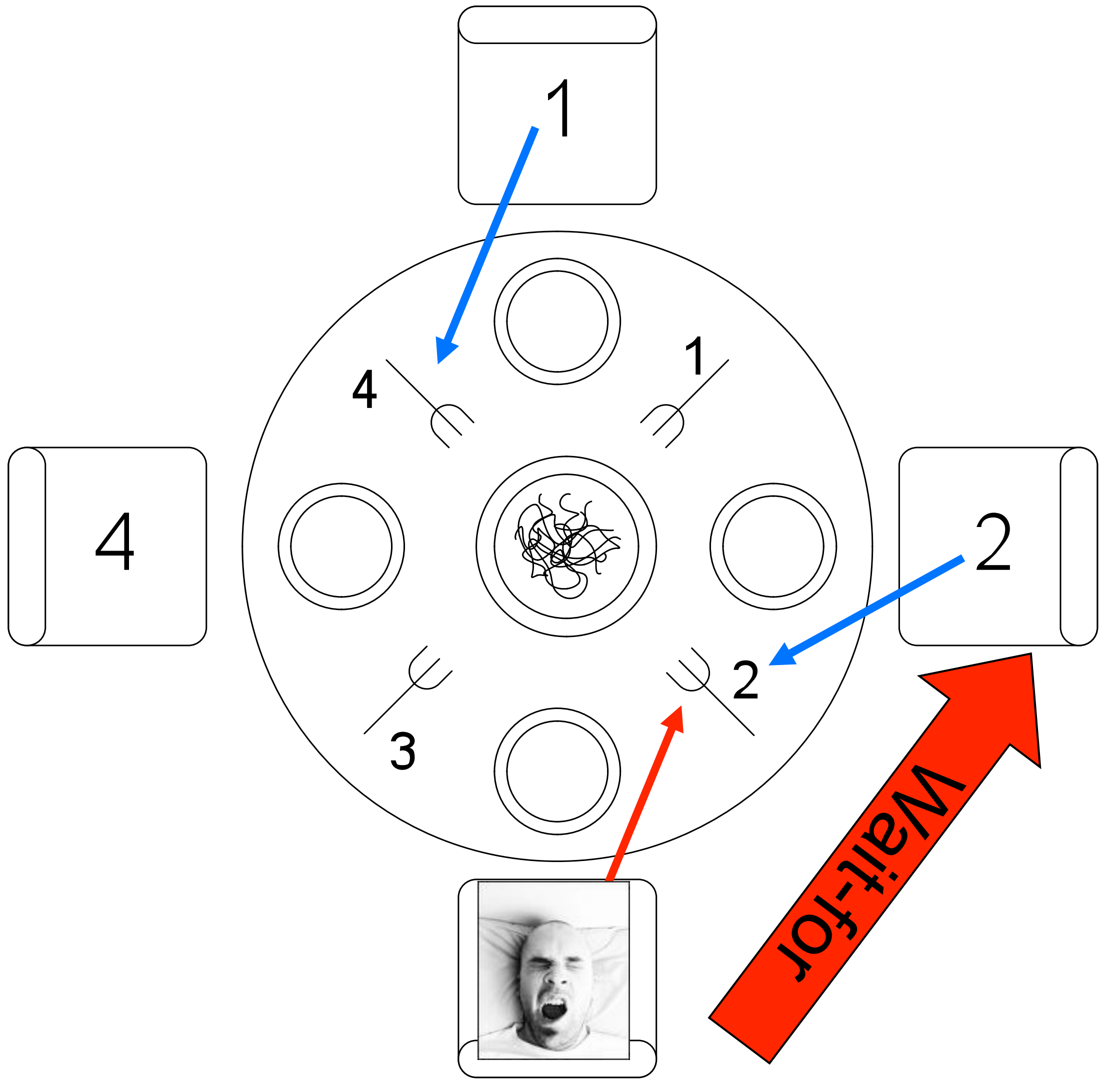
Deadlocks: Dining Philosophers - Solution



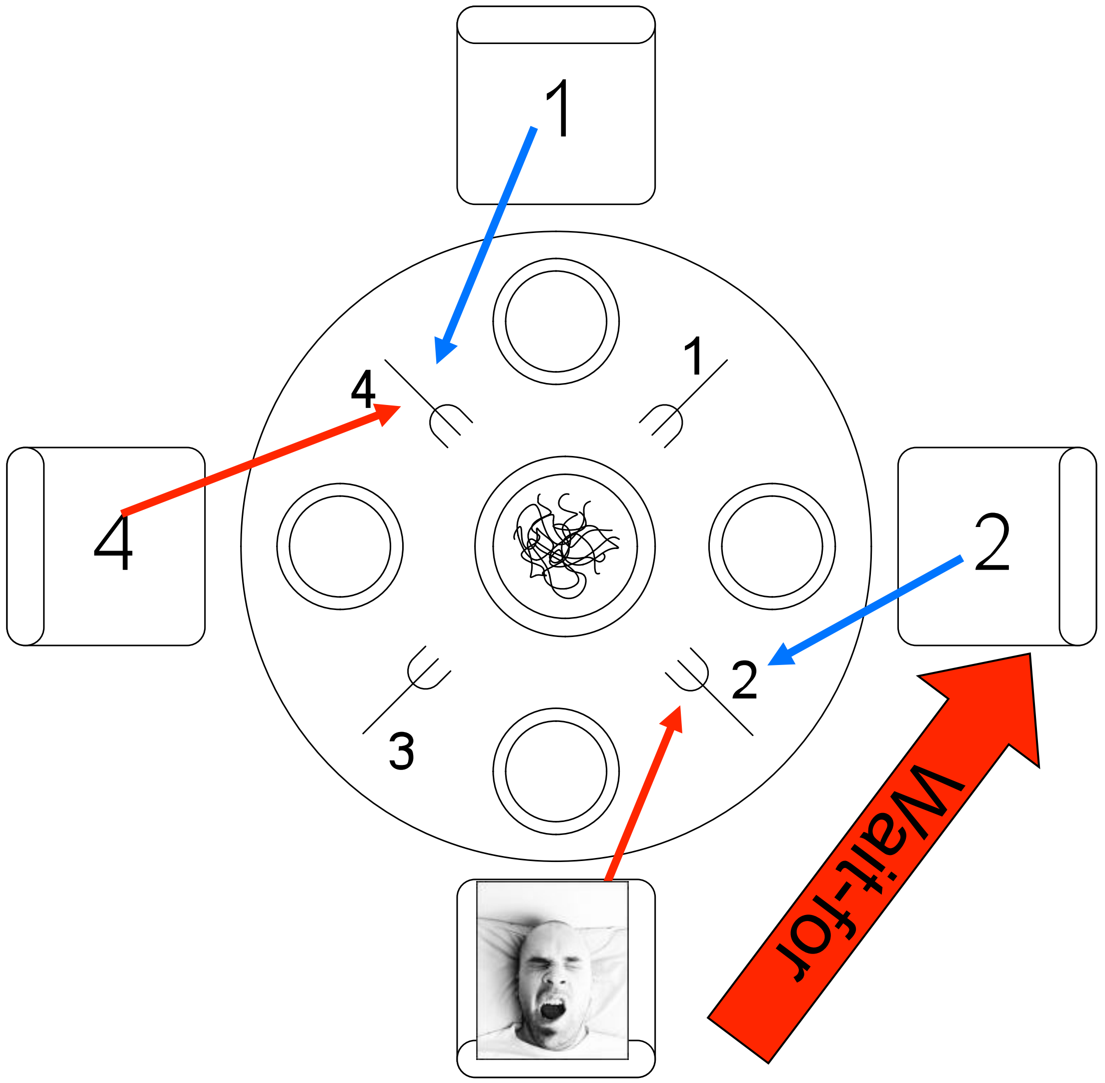
Deadlocks: Dining Philosophers - Solution



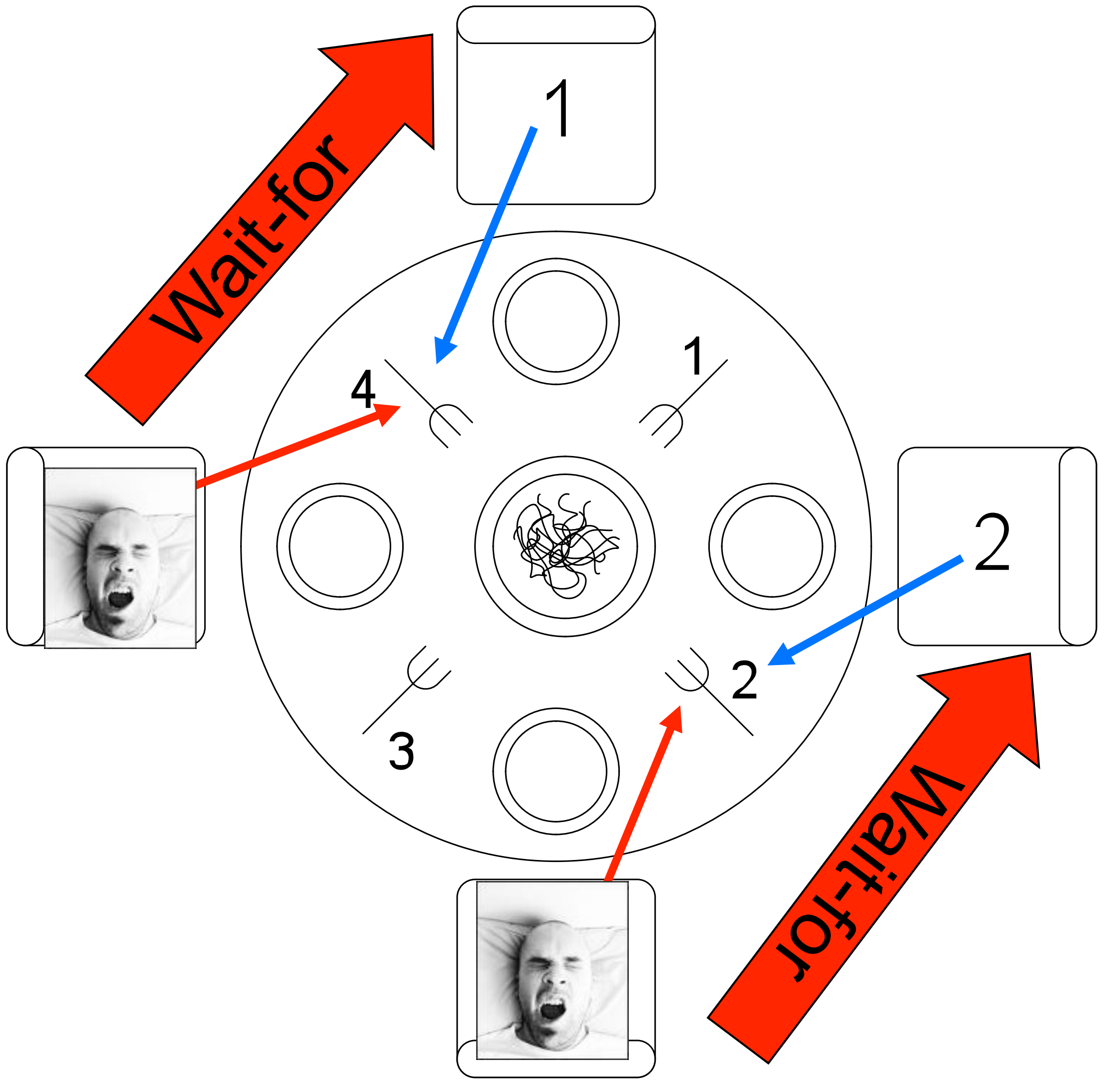
Deadlocks: Dining Philosophers - Solution



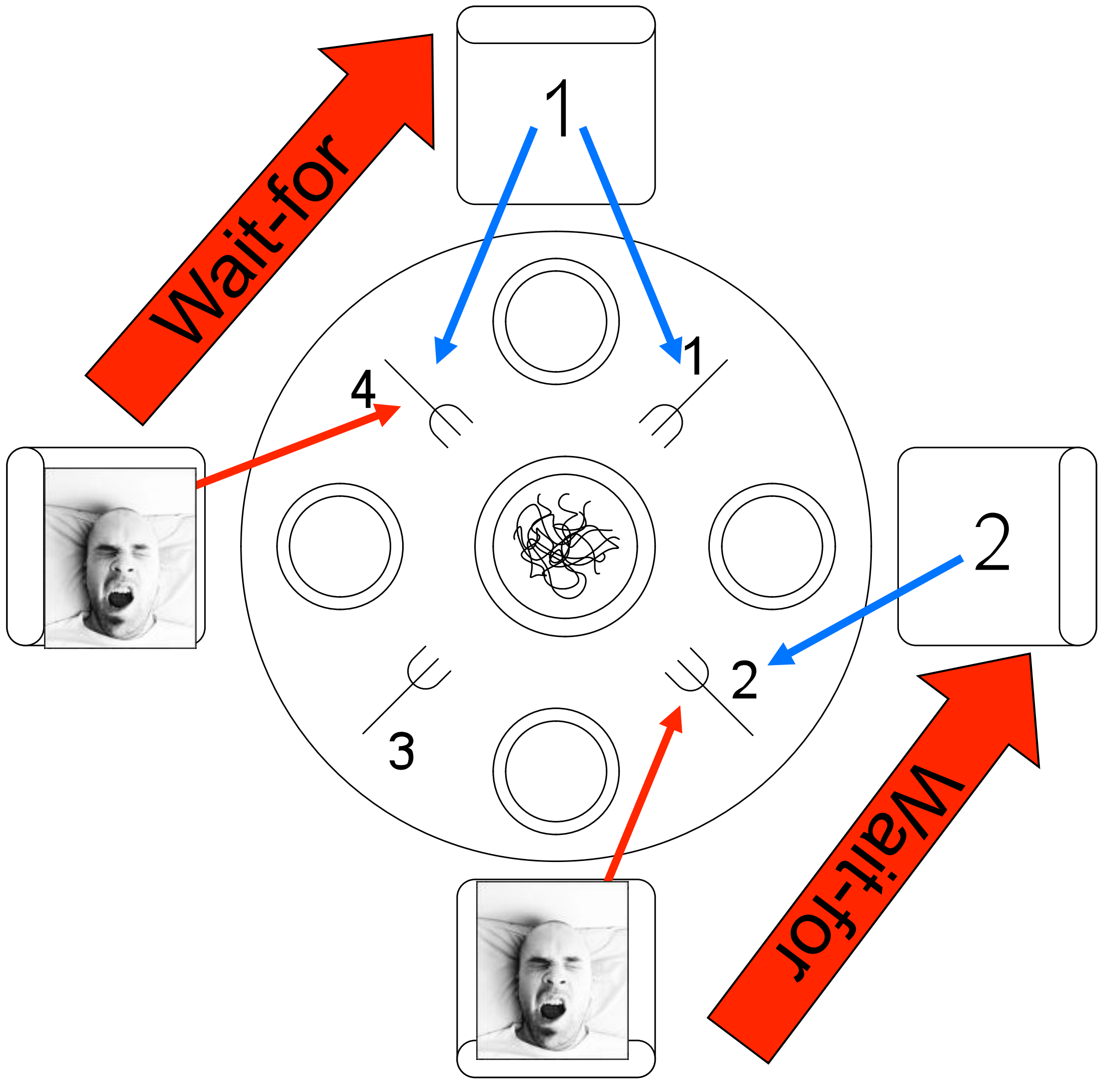
Deadlocks: Dining Philosophers - Solution



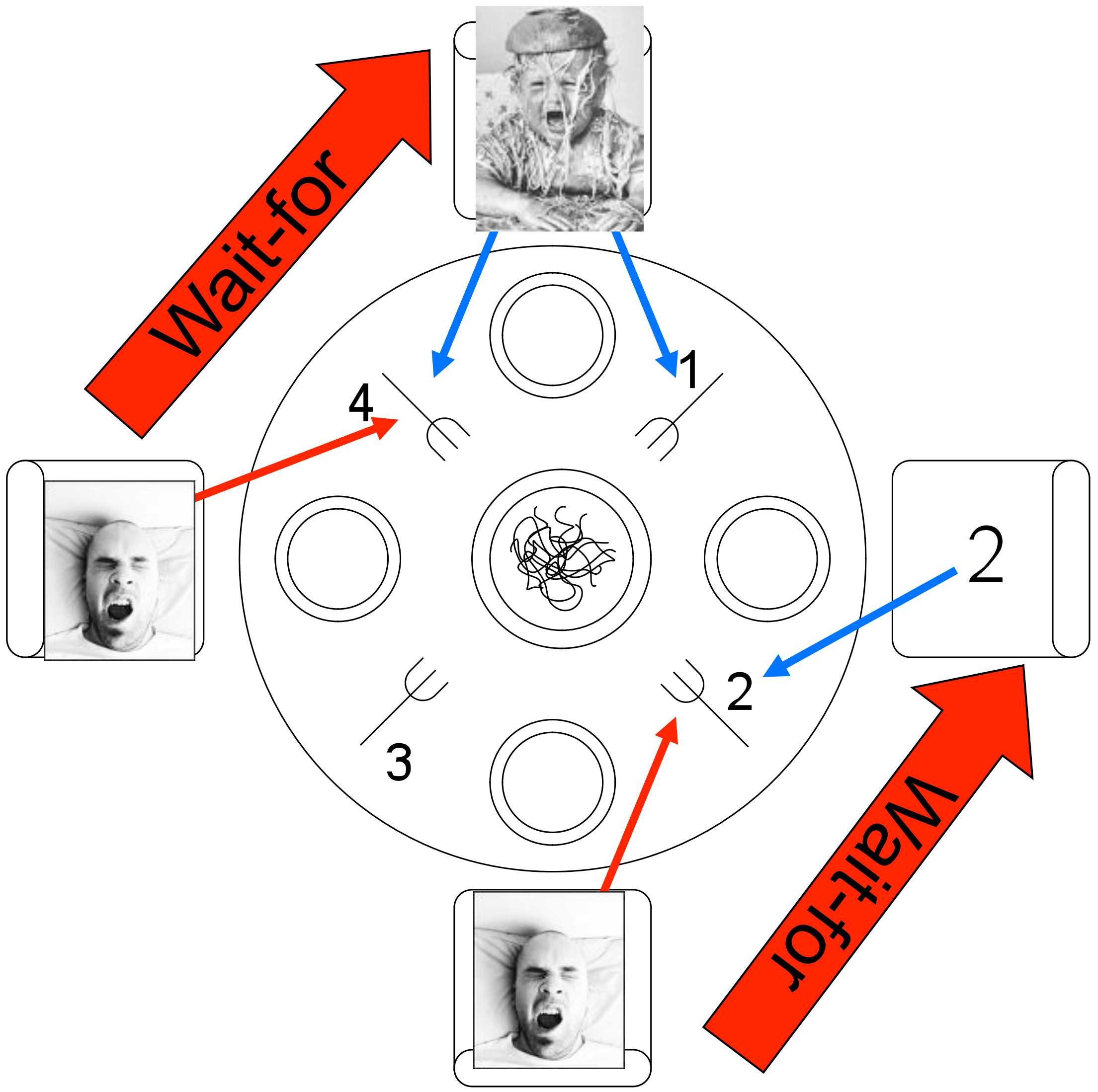
Deadlocks: Dining Philosophers - Solution



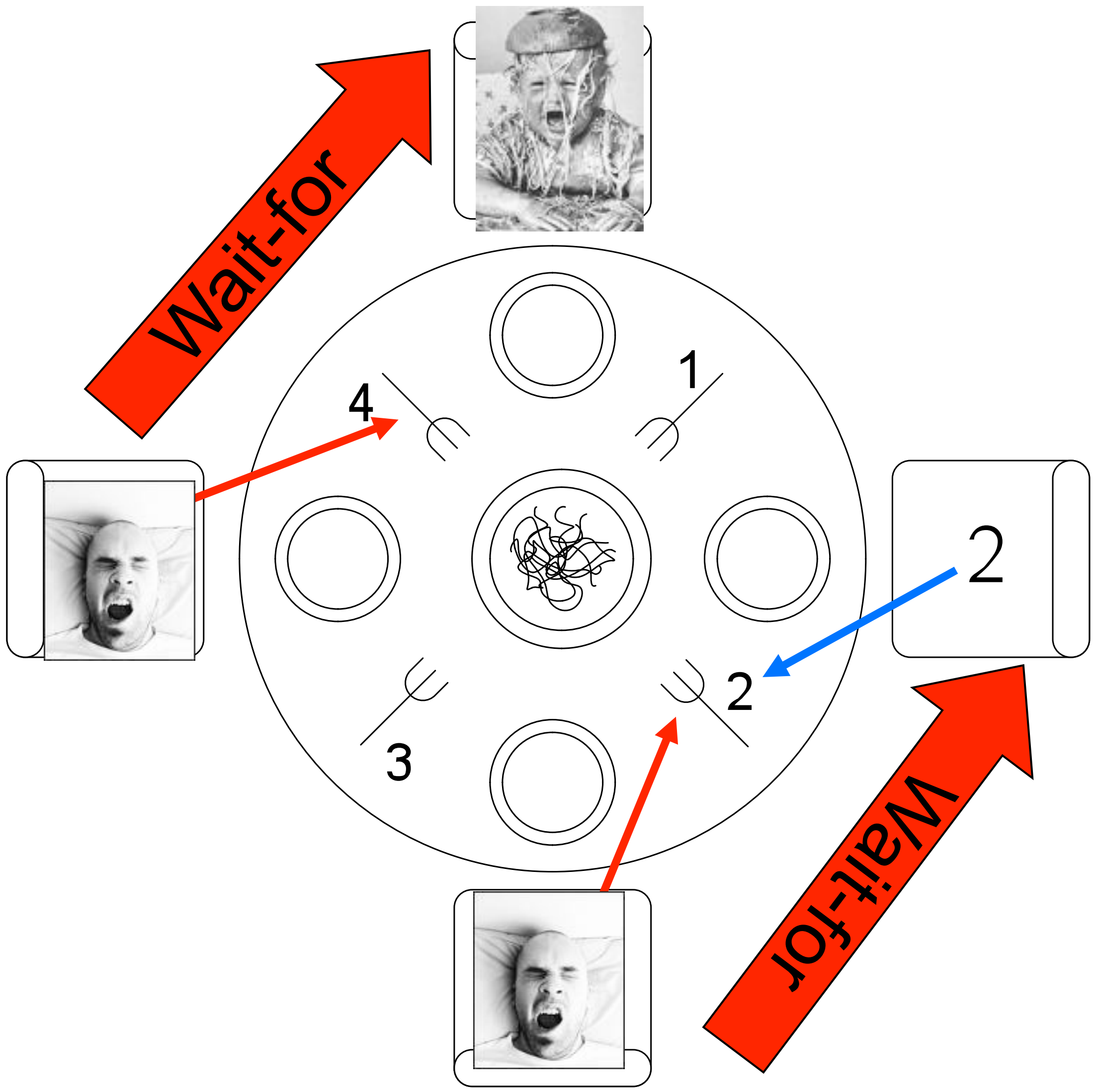
Deadlocks: Dining Philosophers - Solution



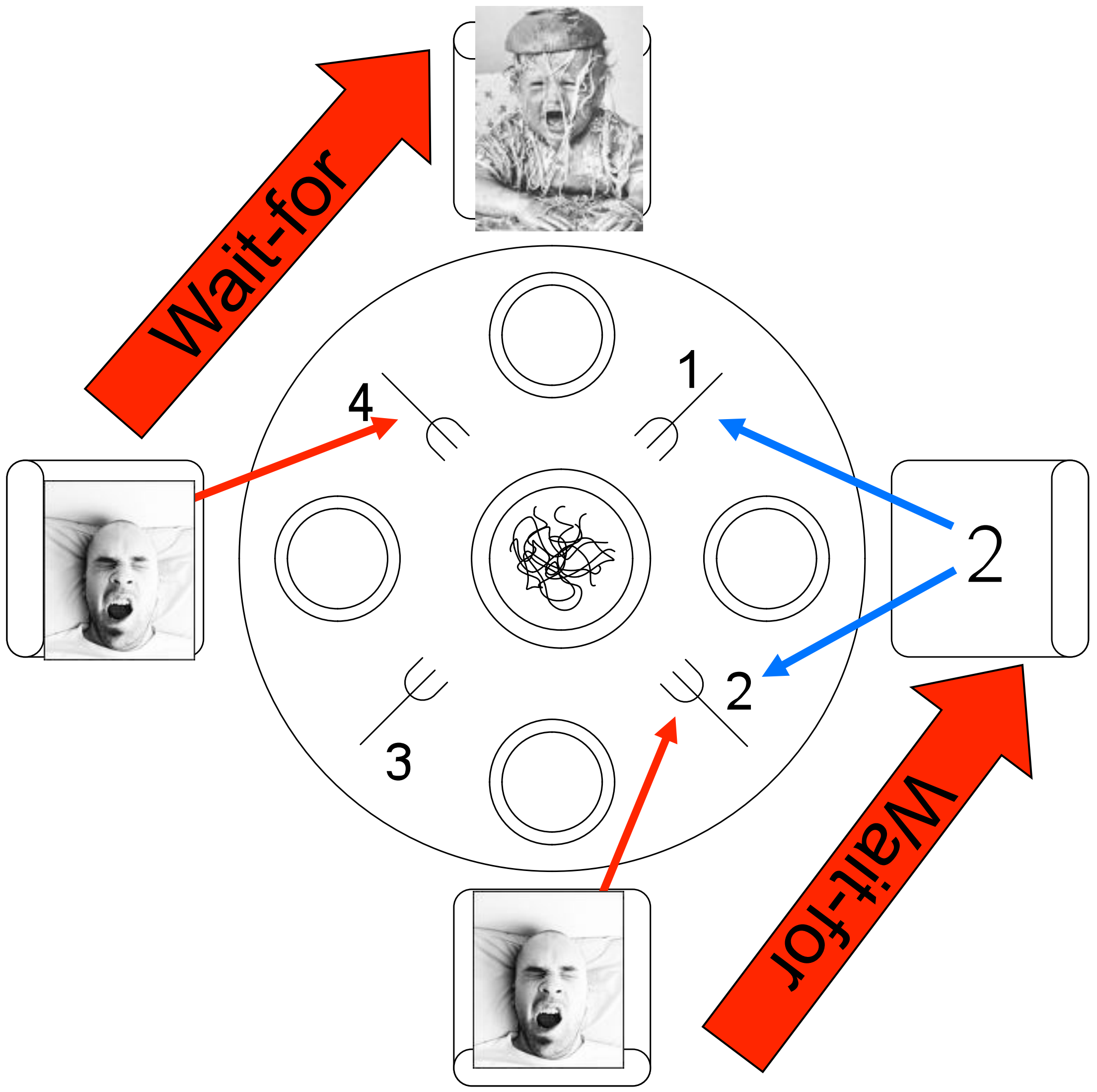
Deadlocks: Dining Philosophers - Solution



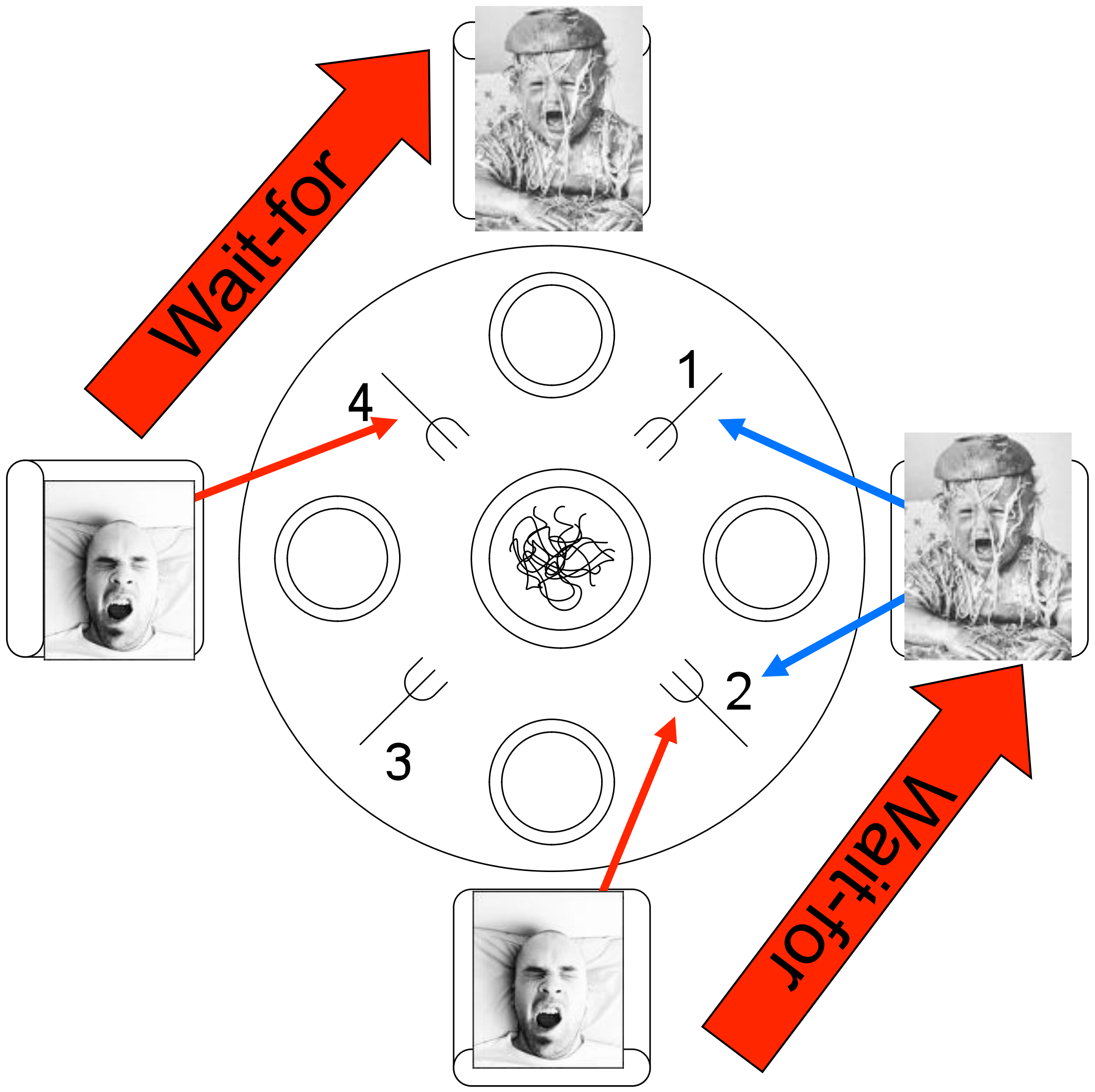
Deadlocks: Dining Philosophers - Solution



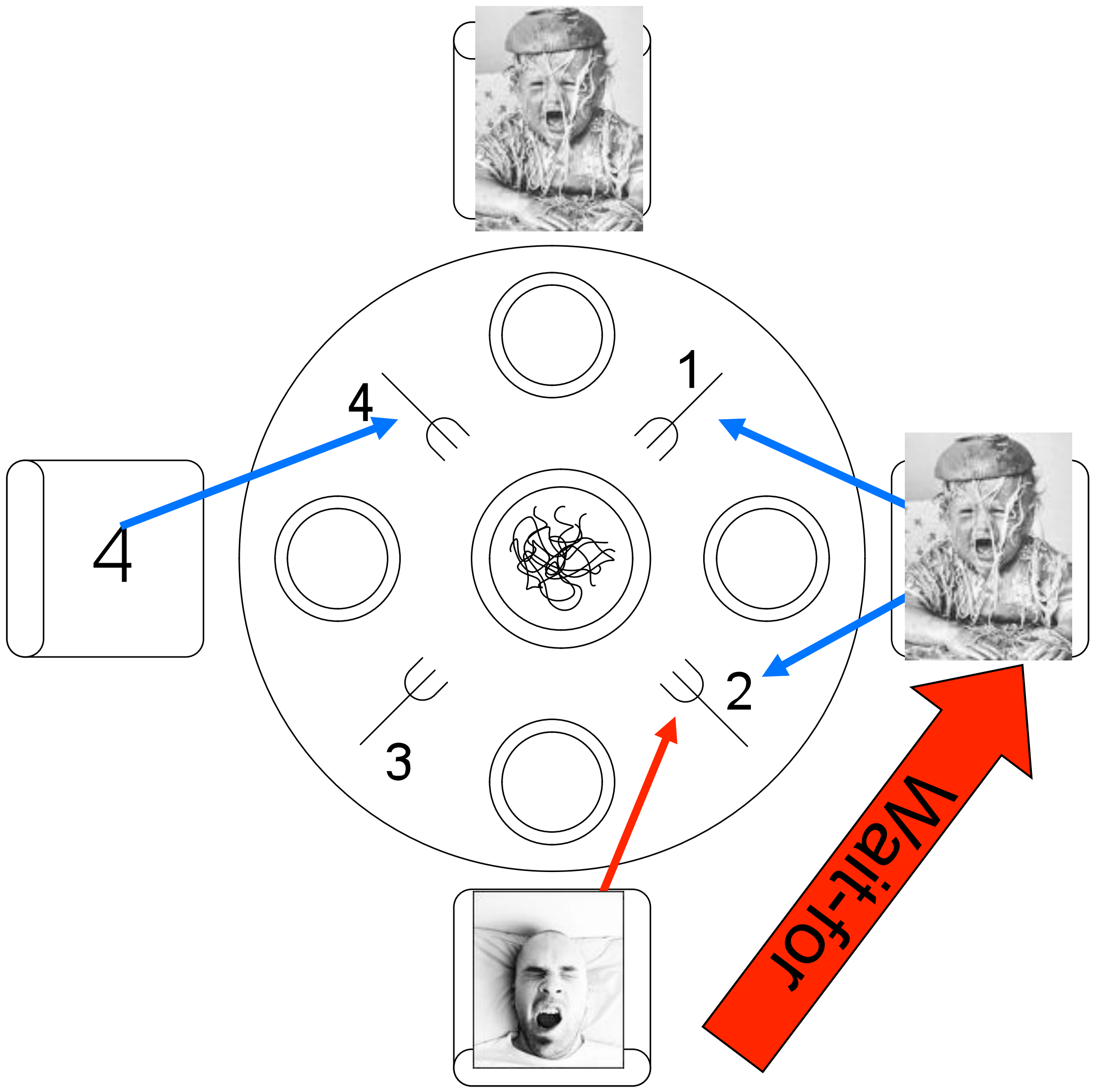
Deadlocks: Dining Philosophers - Solution



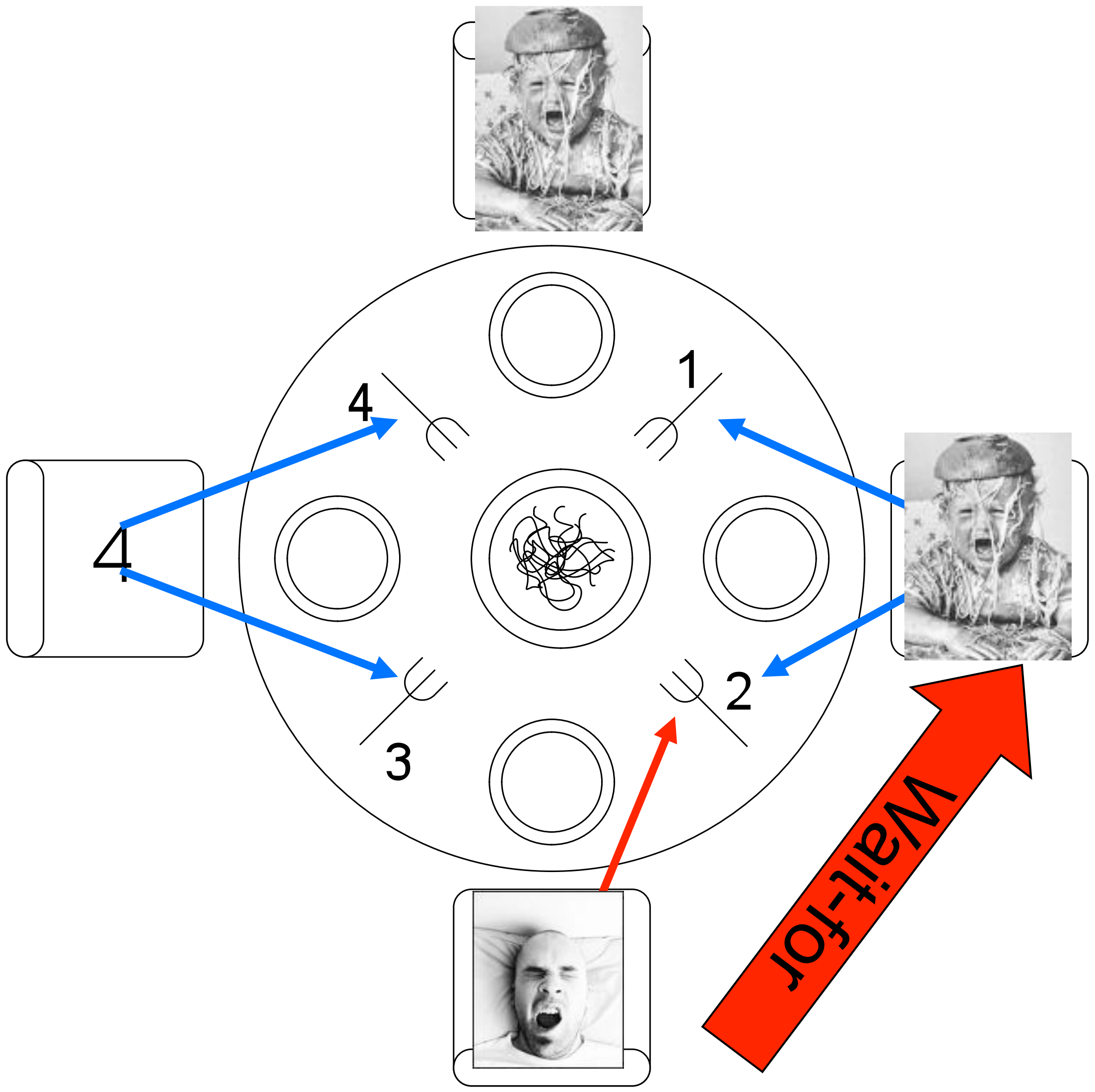
Deadlocks: Dining Philosophers - Solution



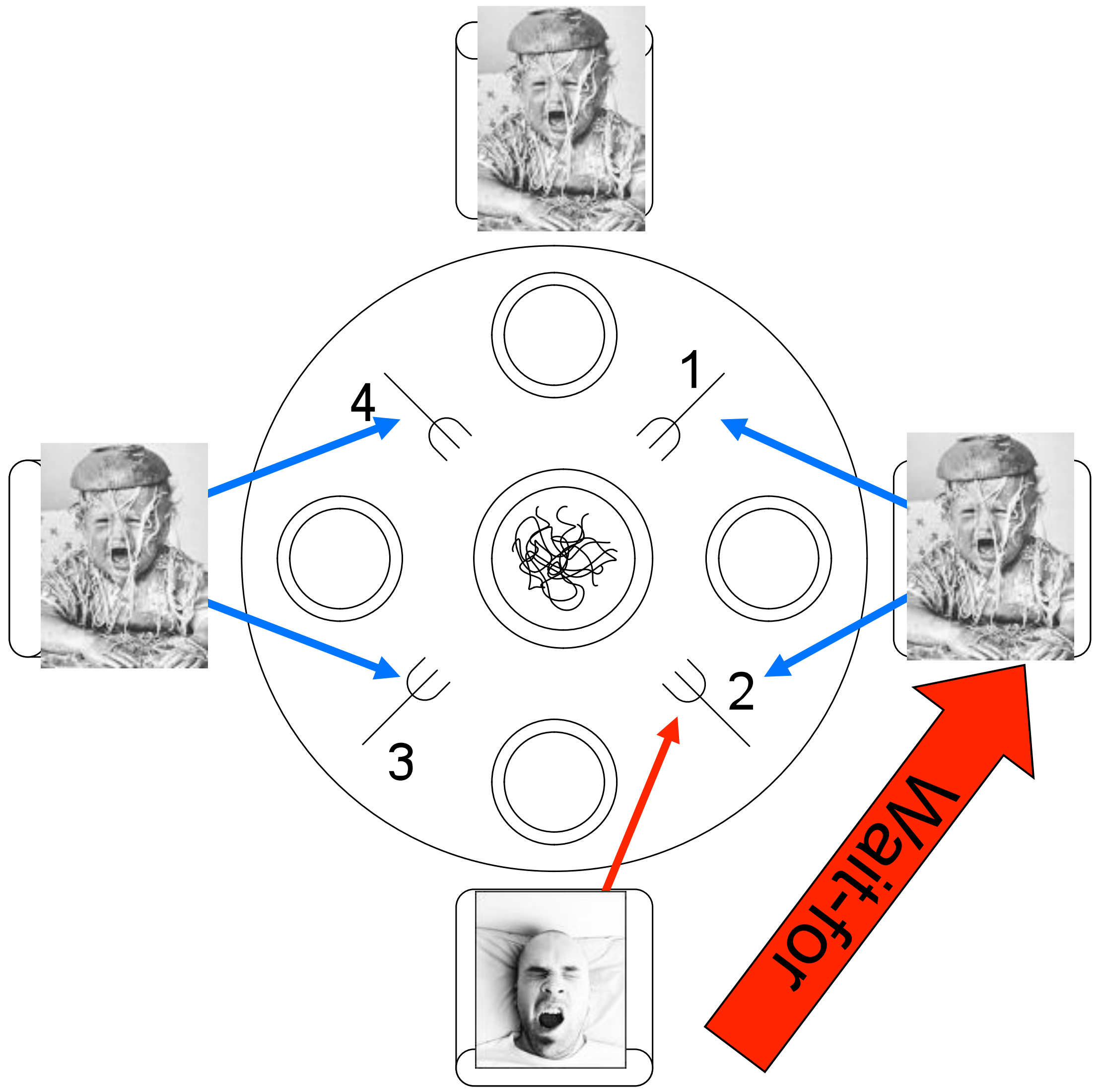
Deadlocks: Dining Philosophers - Solution



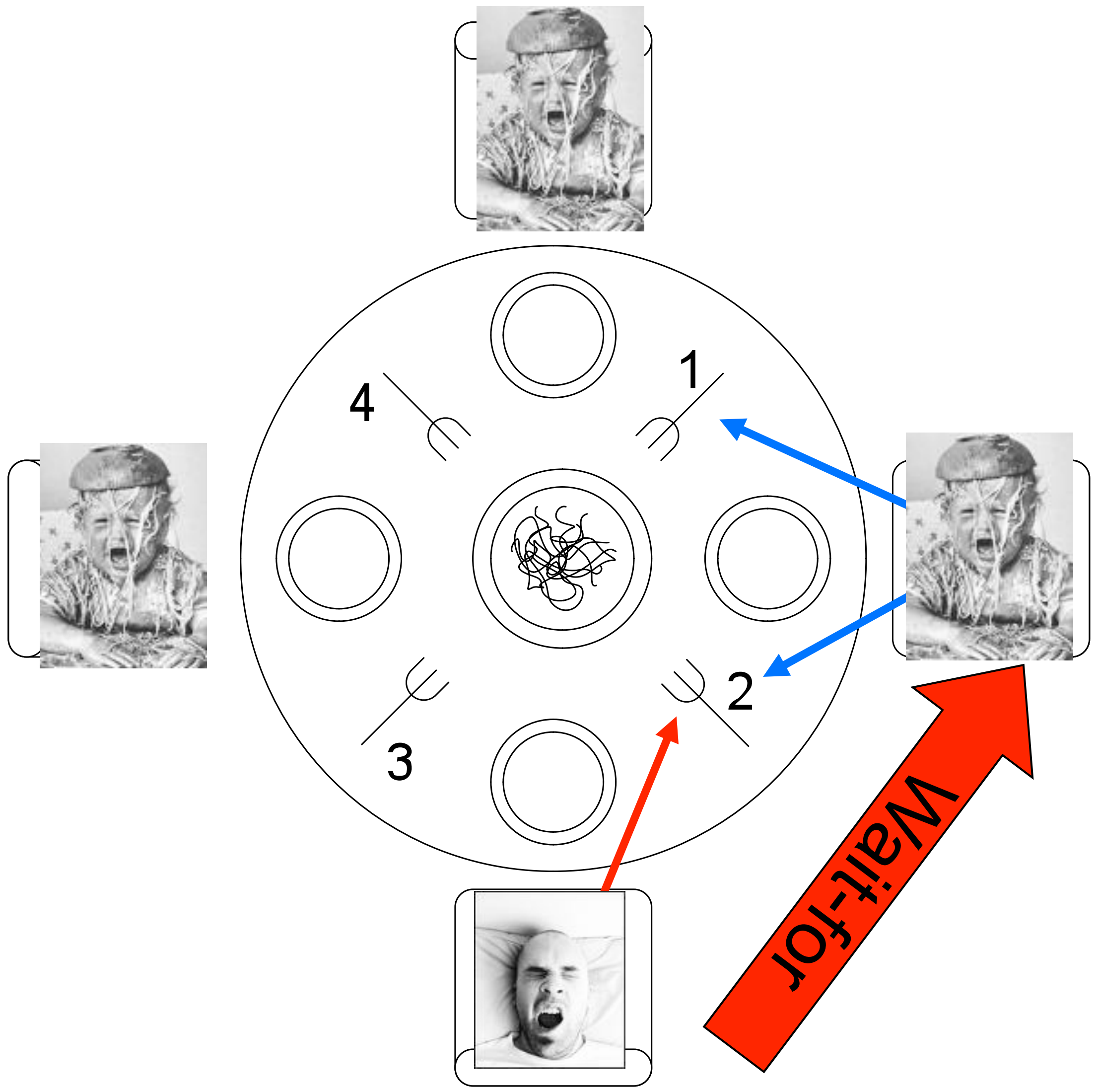
Deadlocks: Dining Philosophers - Solution



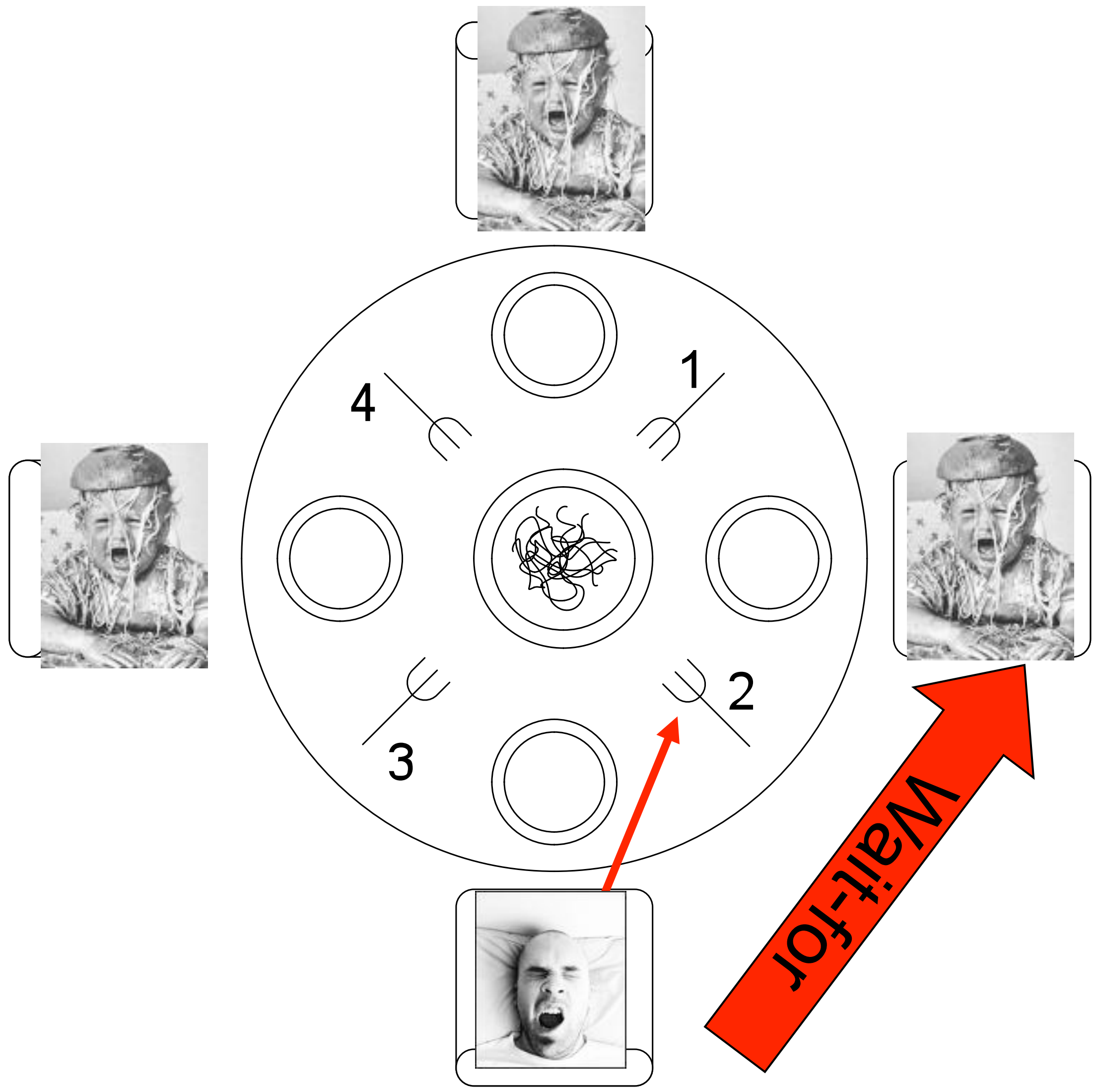
Deadlocks: Dining Philosophers - Solution



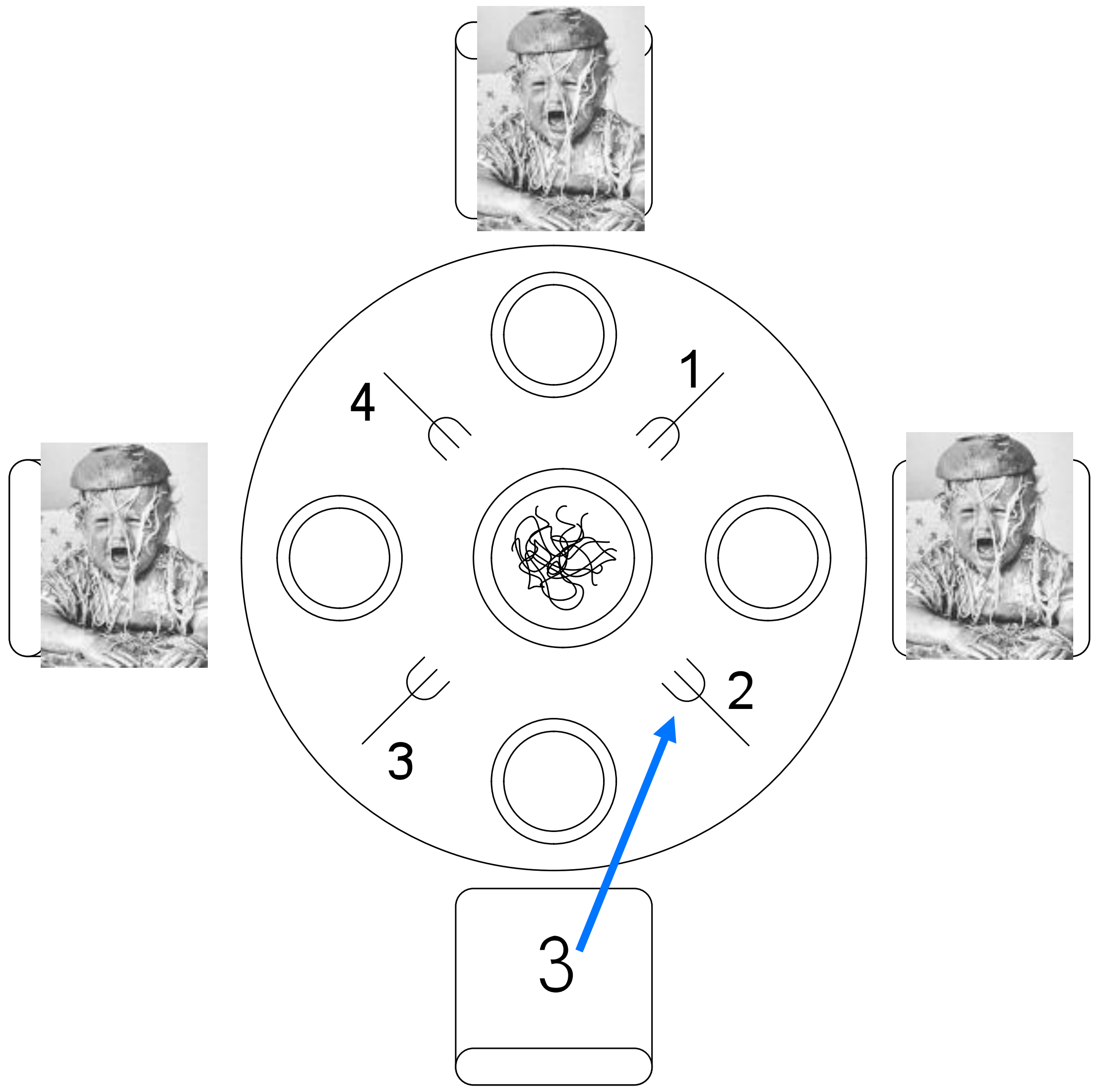
Deadlocks: Dining Philosophers - Solution



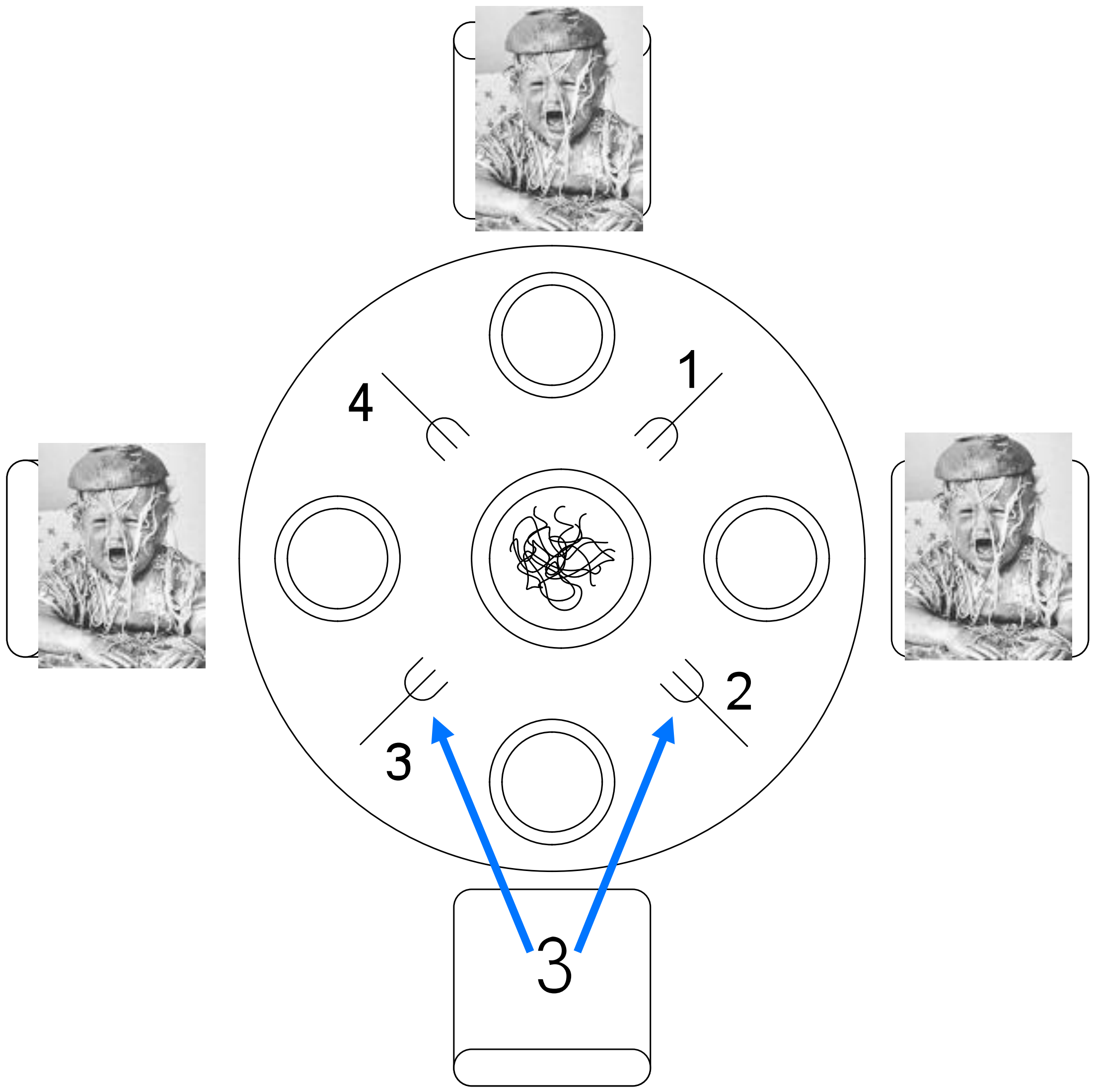
Deadlocks: Dining Philosophers - Solution



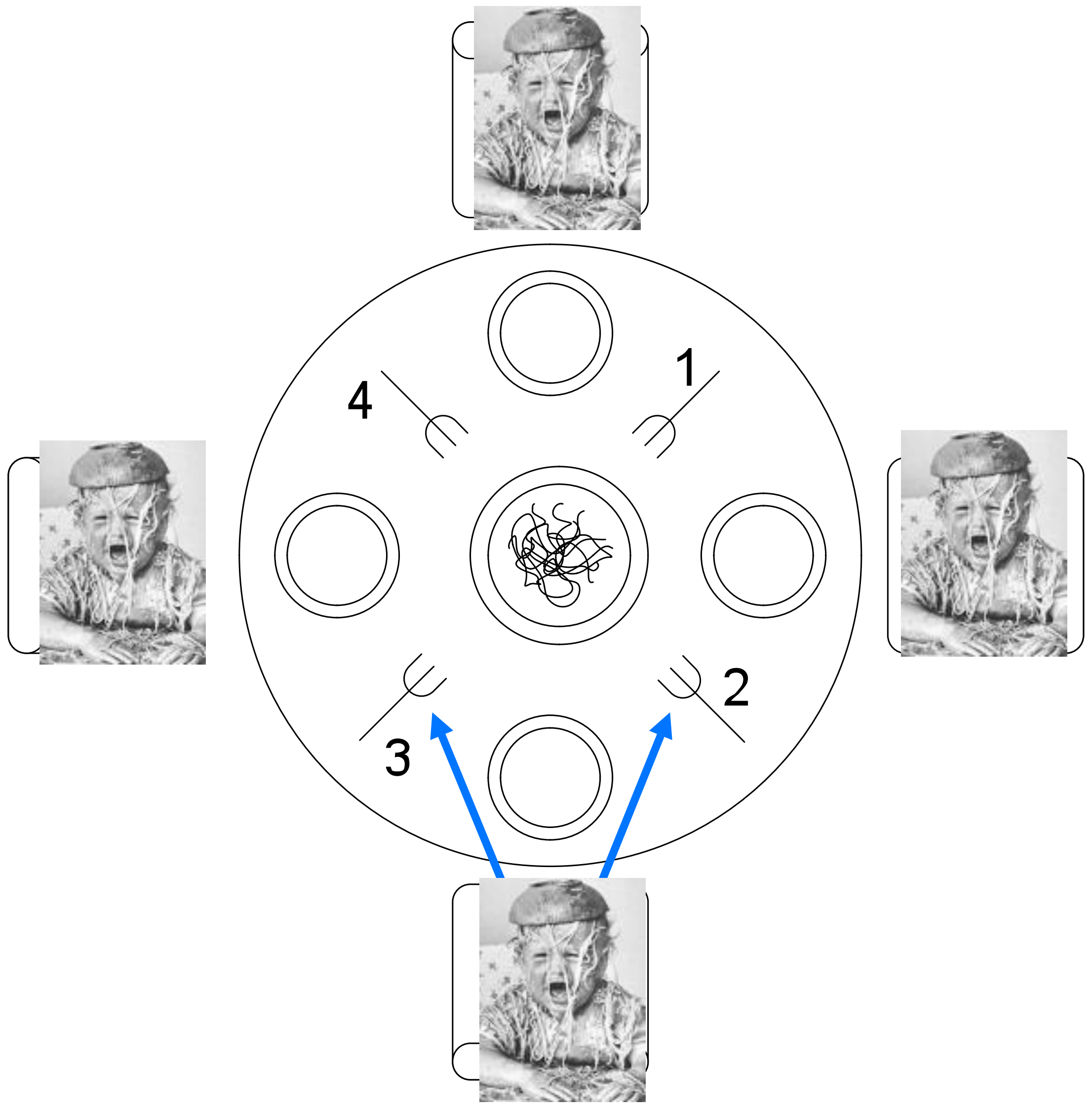
Deadlocks: Dining Philosophers - Solution



Deadlocks: Dining Philosophers - Solution



Deadlocks: Dining Philosophers - Solution



Deadlocks: Dining Philosophers - Solution