THREAD SYNCHRONIZATION II

AARHUS
UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

AGENDA

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

MUTEXES/SEMAPHORES - 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**

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

```
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 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 semaphores!
- **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 semaphores!
- **Example 3: Find and explain the problem**

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

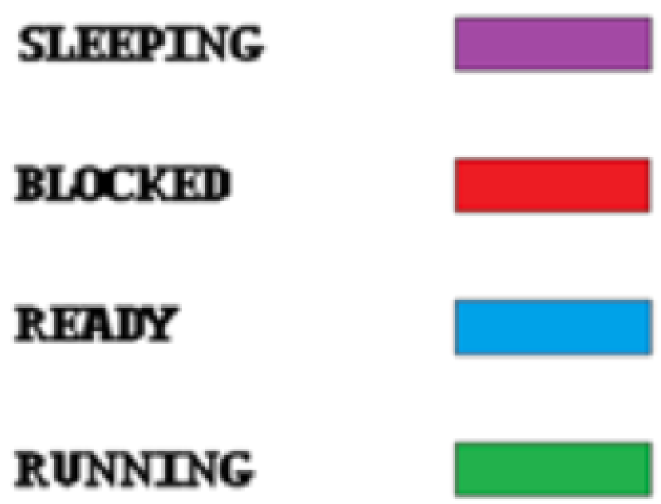
```
unsigned int shared;
SEM_ID s;

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

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

PRIORITY INVERSION

PRIORITY INVERSION

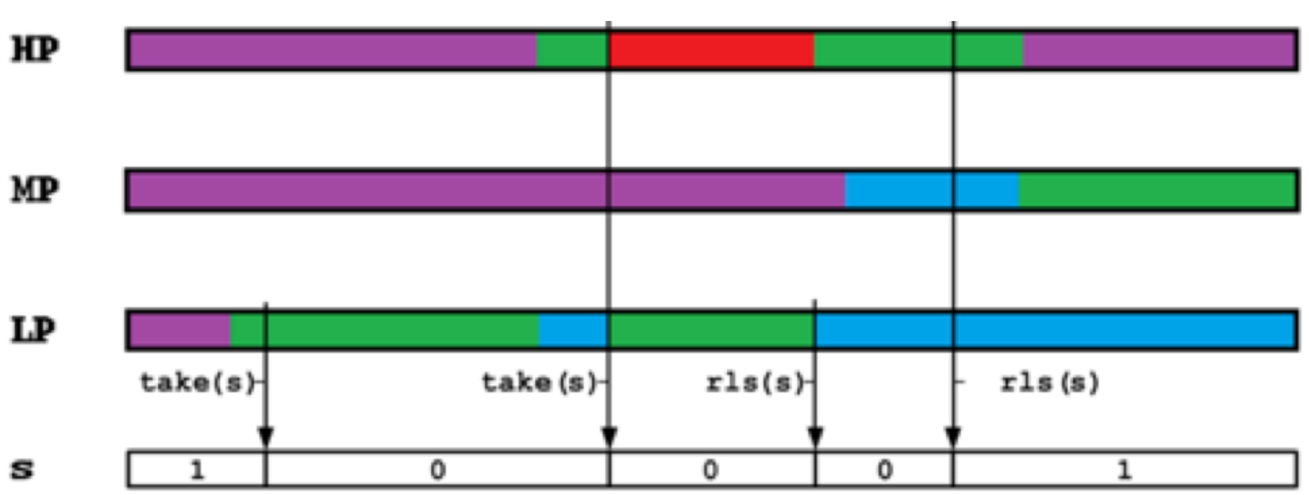


HP: High priority thread

MP: Medium priority thread

LP: Low priority thread

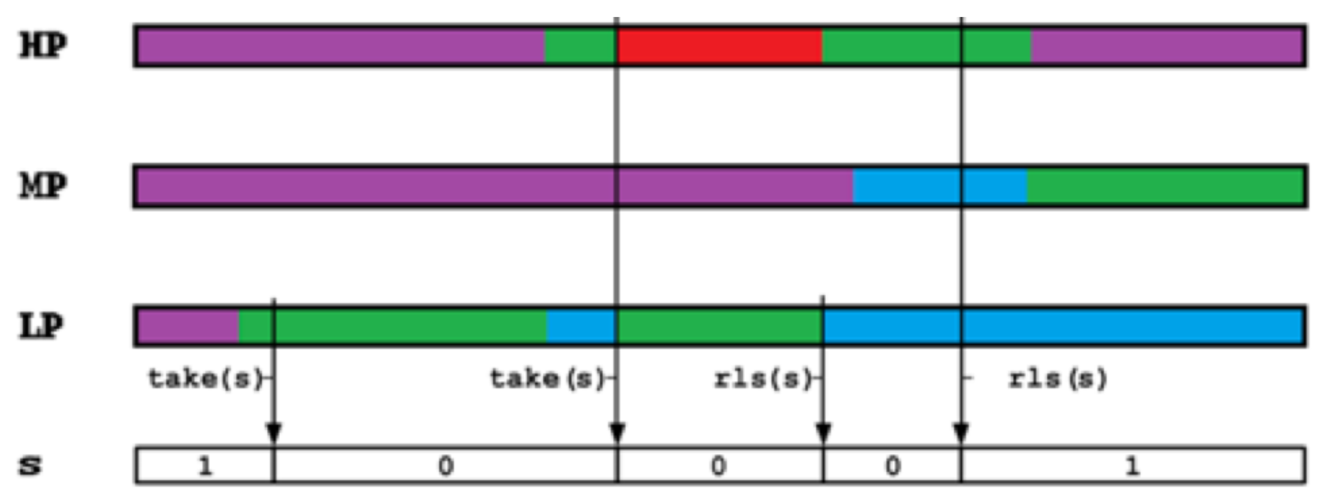
SCENARIO 1 - BOUNDED PRIORITY INVERSION



- LP runs
- LP acquires mutex
- HP is prioritized to run, LP on waiting queue (WQ)
- HP blocked due to mutex taken
- LP runs until mutex release
- HP runs until done, LP on WQ
- MP is ready but due to lower priority -> WQ
- LP waits until both HP and MP done and then run until done

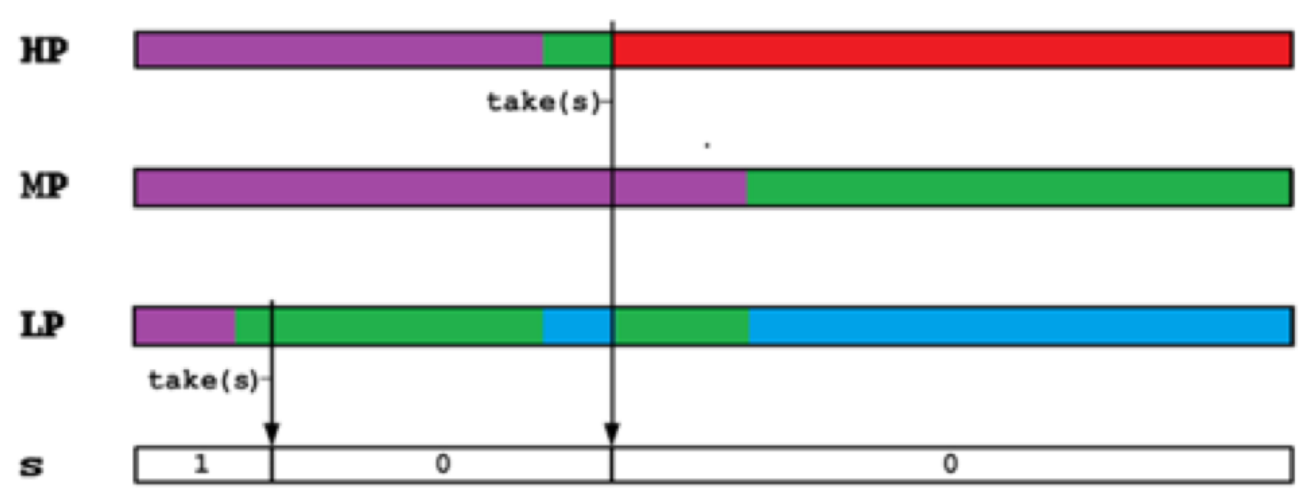
PRIORITY INVERSION

SCENARIO 1 - BOUNDED PRIORITY INVERSION (FROM LAST SLIDE)



- LP runs
- LP acquires mutex
- HP is prioritized to run, LP on waiting queue (WQ)
- HP blocked due to mutex taken
- LP runs until mutex release
- HP runs until done, LP on WQ
- MP is ready but due to lower priority -> WQ
- LP waits until both HP and MP done and then run until done

SCENARIO 2 - UNBOUNDED PRIORITY INVERSION



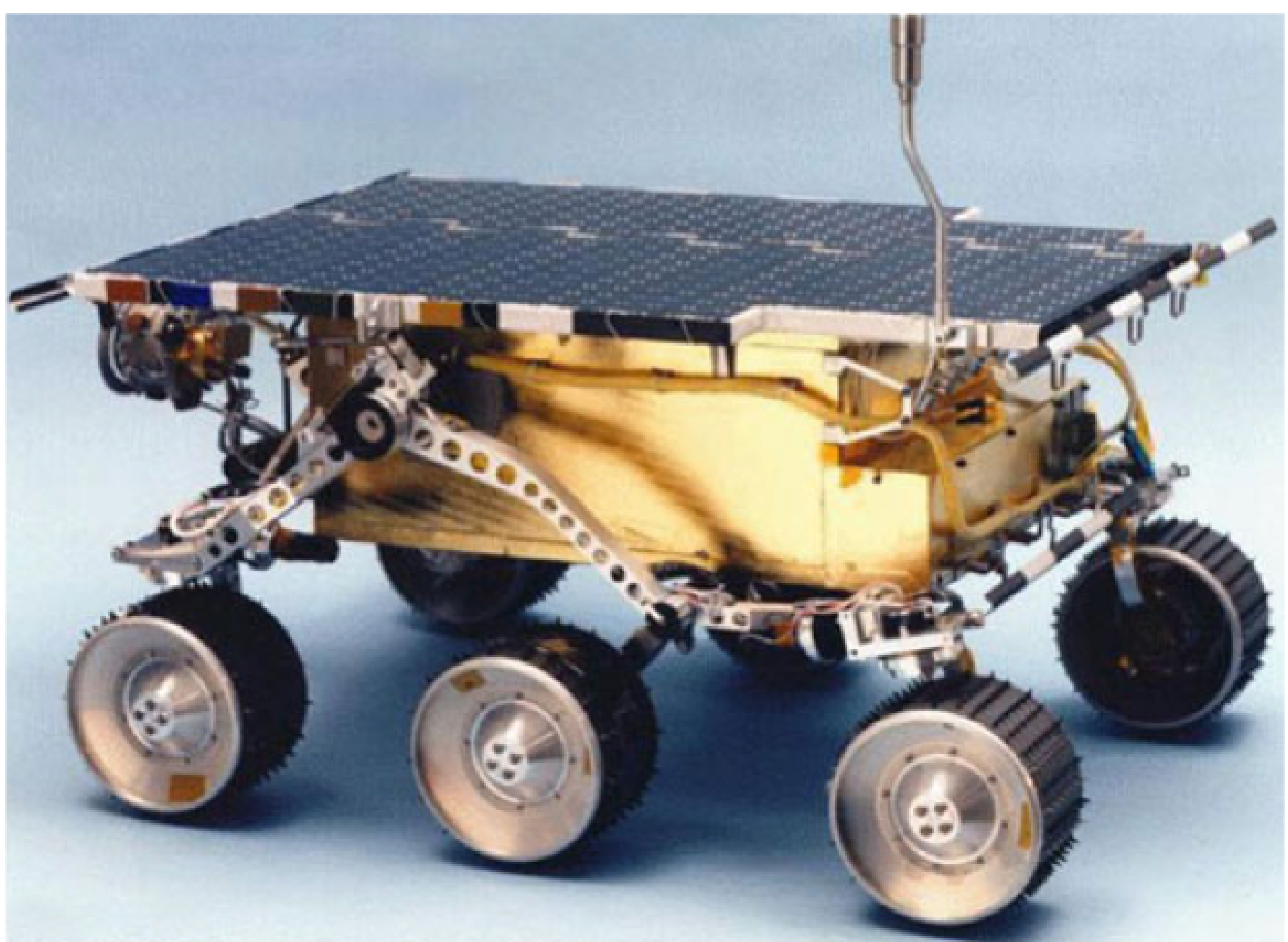
- LP runs
- LP acquires mutex
- HP is prioritized to run, LP on WQ
- HP blocked due to mutex taken - LP continues
- MP is prioritized to run (over LP) until done
- MP is thus scheduled ahead of HP - priority inversion
- LP runs until mutex release, HP is blocked
- HP runs until done
- LP waits until HP done

UNBOUNDED PRIORITY INVERSION

- Unbounded 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

UNBOUNDED PRIORITY INVERSION

- Unbounded 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

UNBOUNDED PRIORITY INVERSION

- Unbounded 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

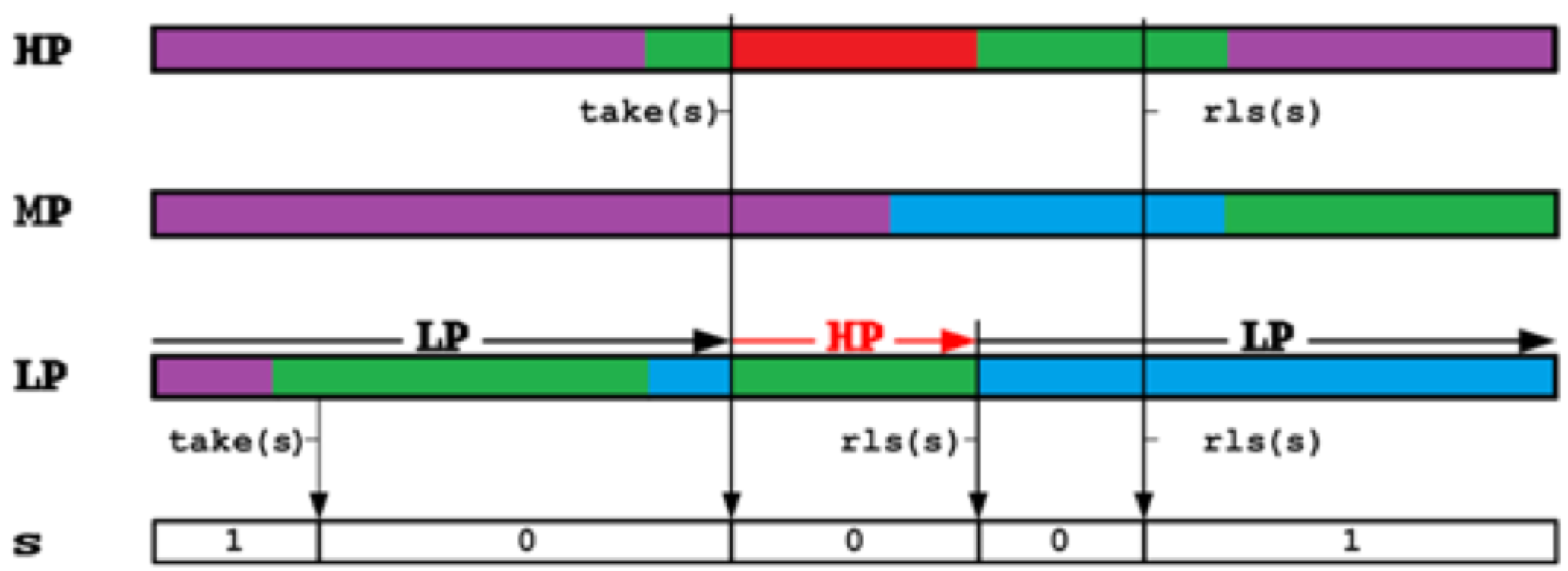
UNBOUNDED PRIORITY INVERSION

- Unbounded 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.

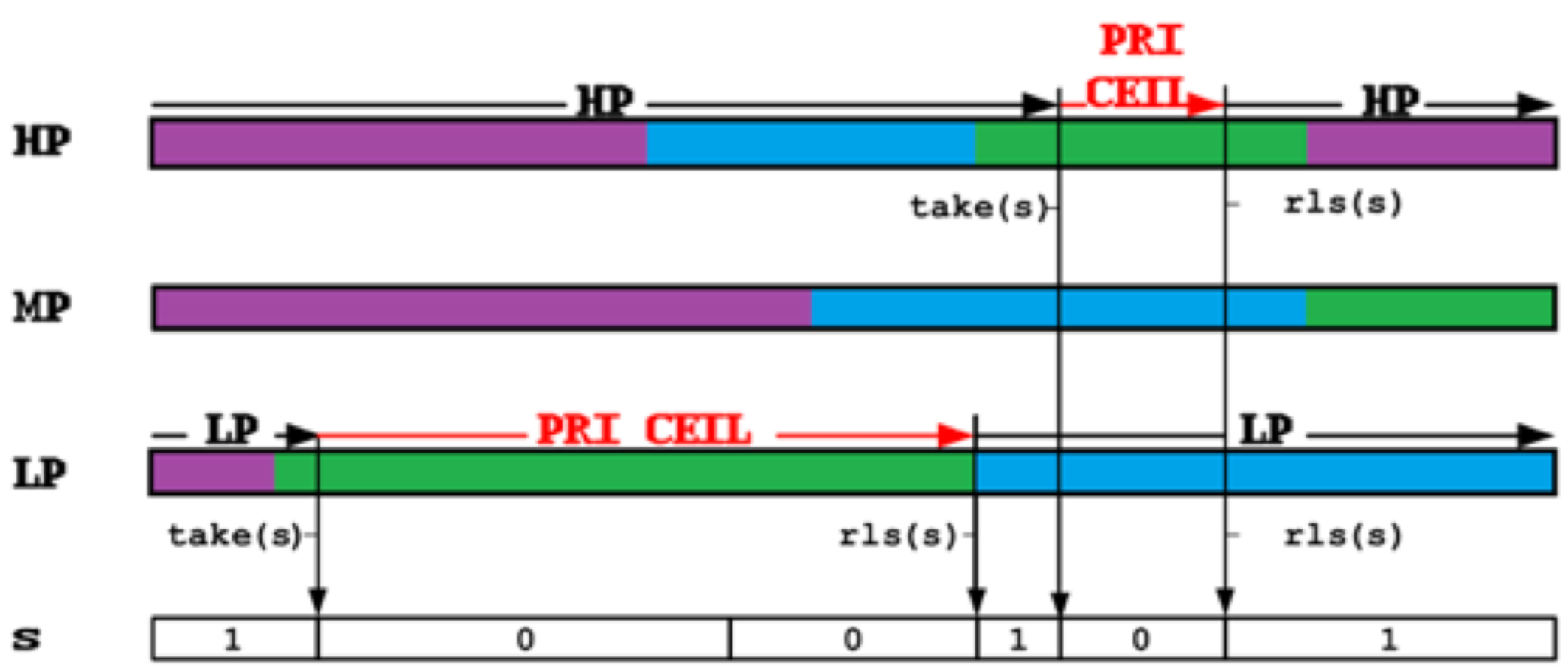


1. LP runs
2. LP acquires mutex
3. HP is prioritized to run, LP on waiting queue (WQ)
4. HP blocked due to mutex taken
5. LP runs until mutex release, ***but with HP priority (inheritance)***
6. MP wants to run but due to lower priority -> WQ
7. HP acquires mutex and runs until done, MP & LP on WQ
8.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



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

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);
}
```

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.

"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)

DEADLOCKS

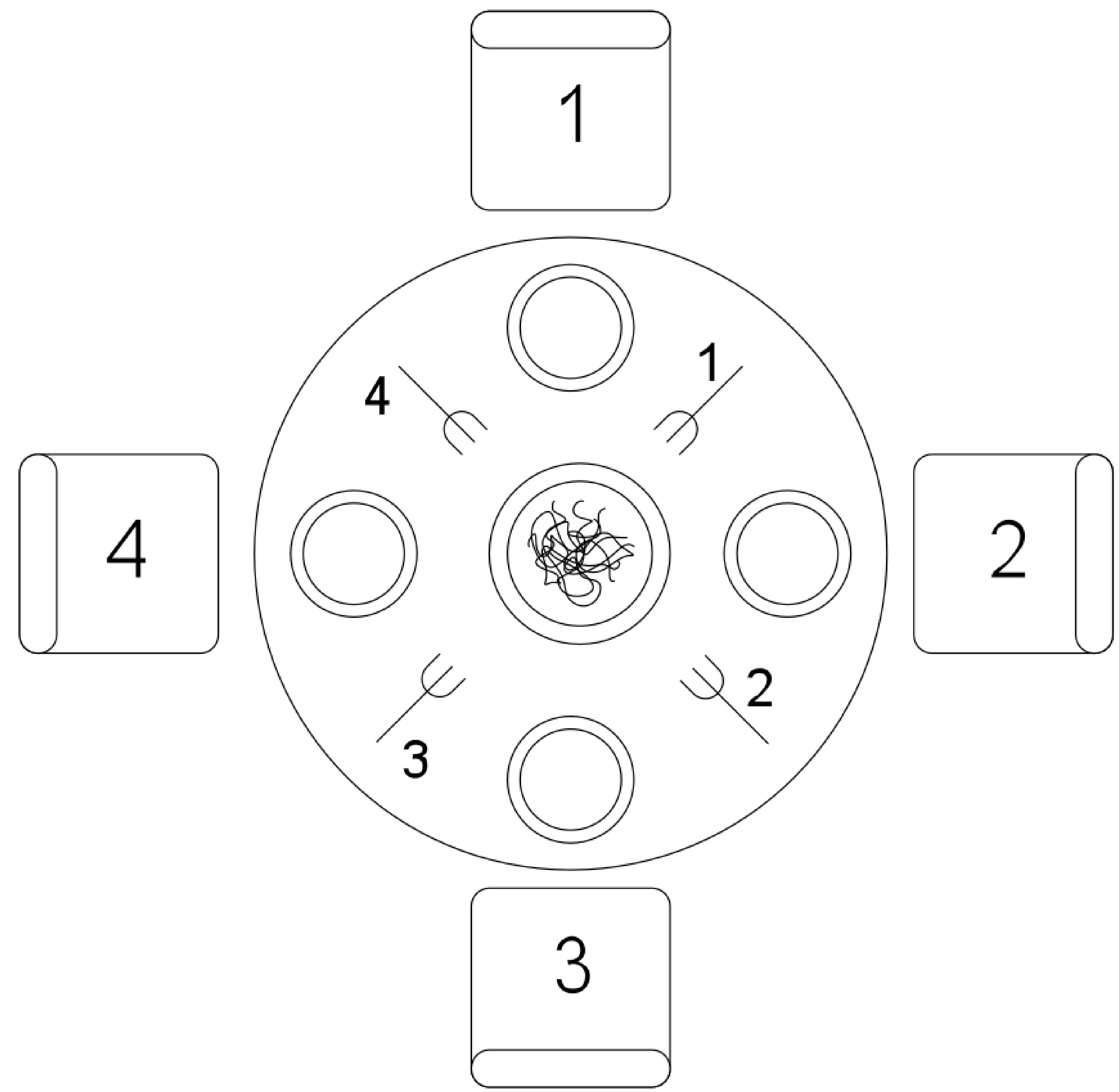
THE FOUR NECESSARY CONDITIONS FOR DEADLOCKS

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

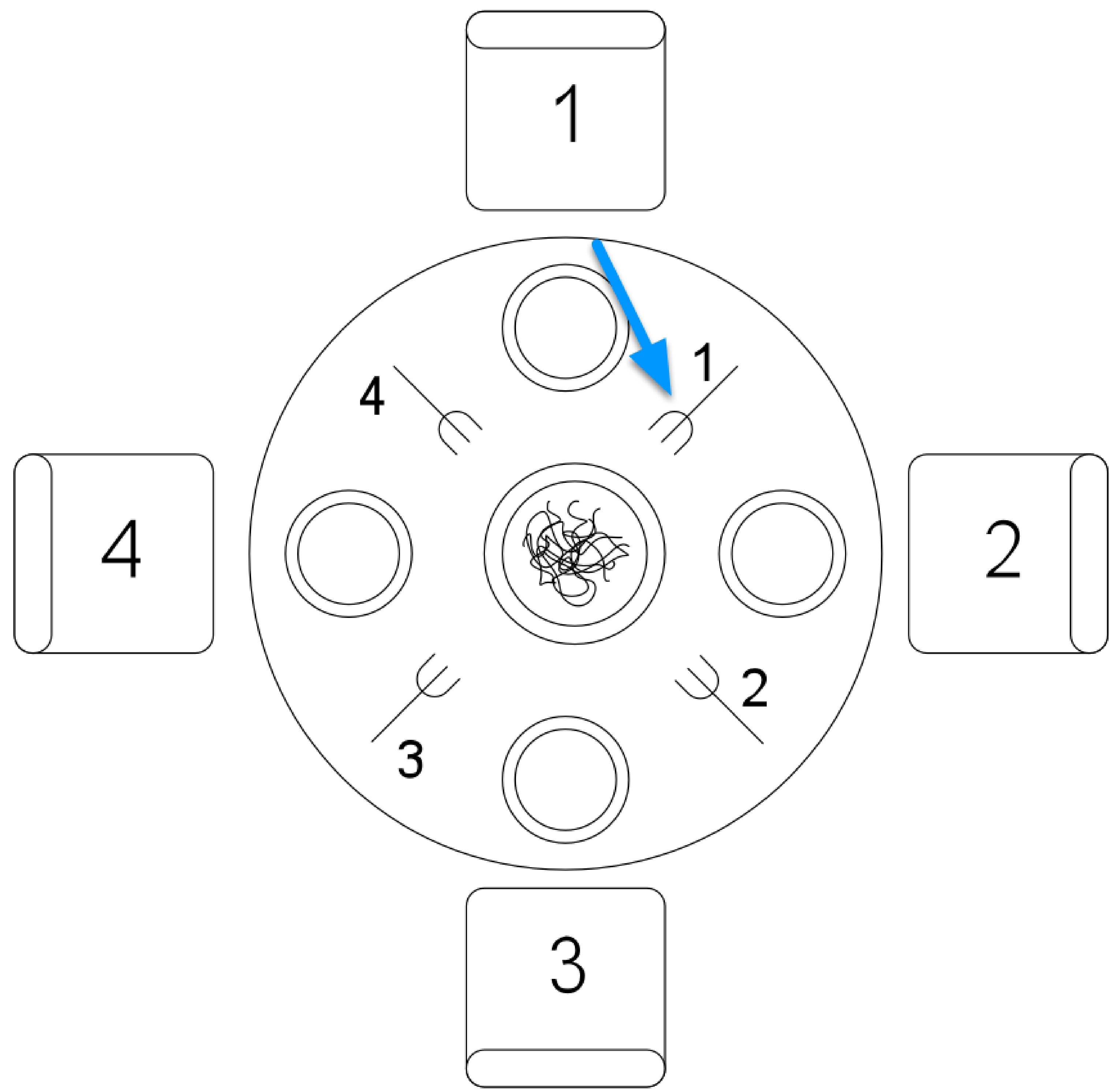
1. A philosopher picks up ***left*** and ***right*** forks, eats, put down ***left*** and ***right*** forks and thinks
2. If a fork is ***taken*** the philosopher will have to ***wait*** for it to be ***free***

Question: Can this system deadlock? If so how?

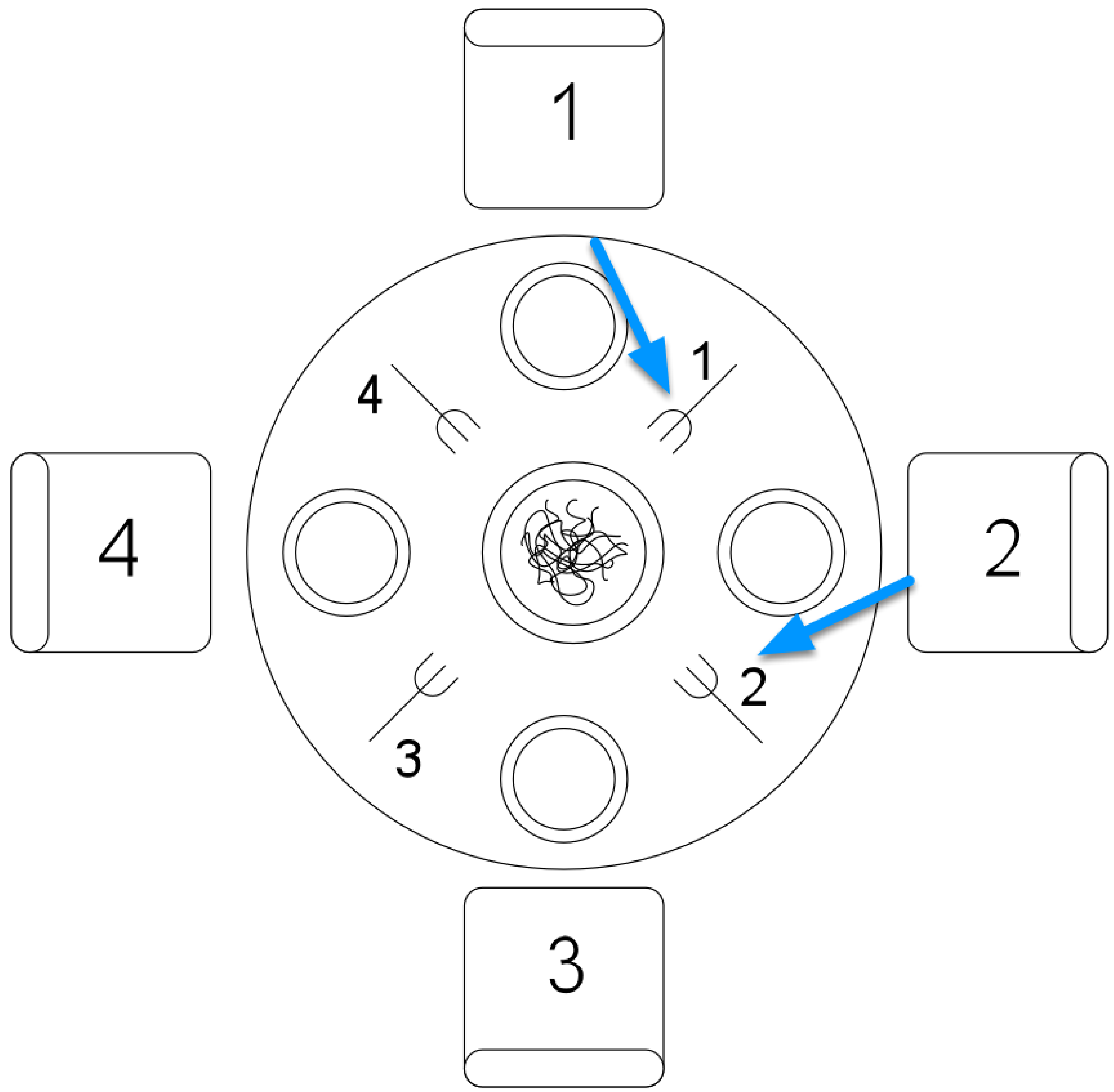


DEADLOCKS: DINING PHILOSOPHERS

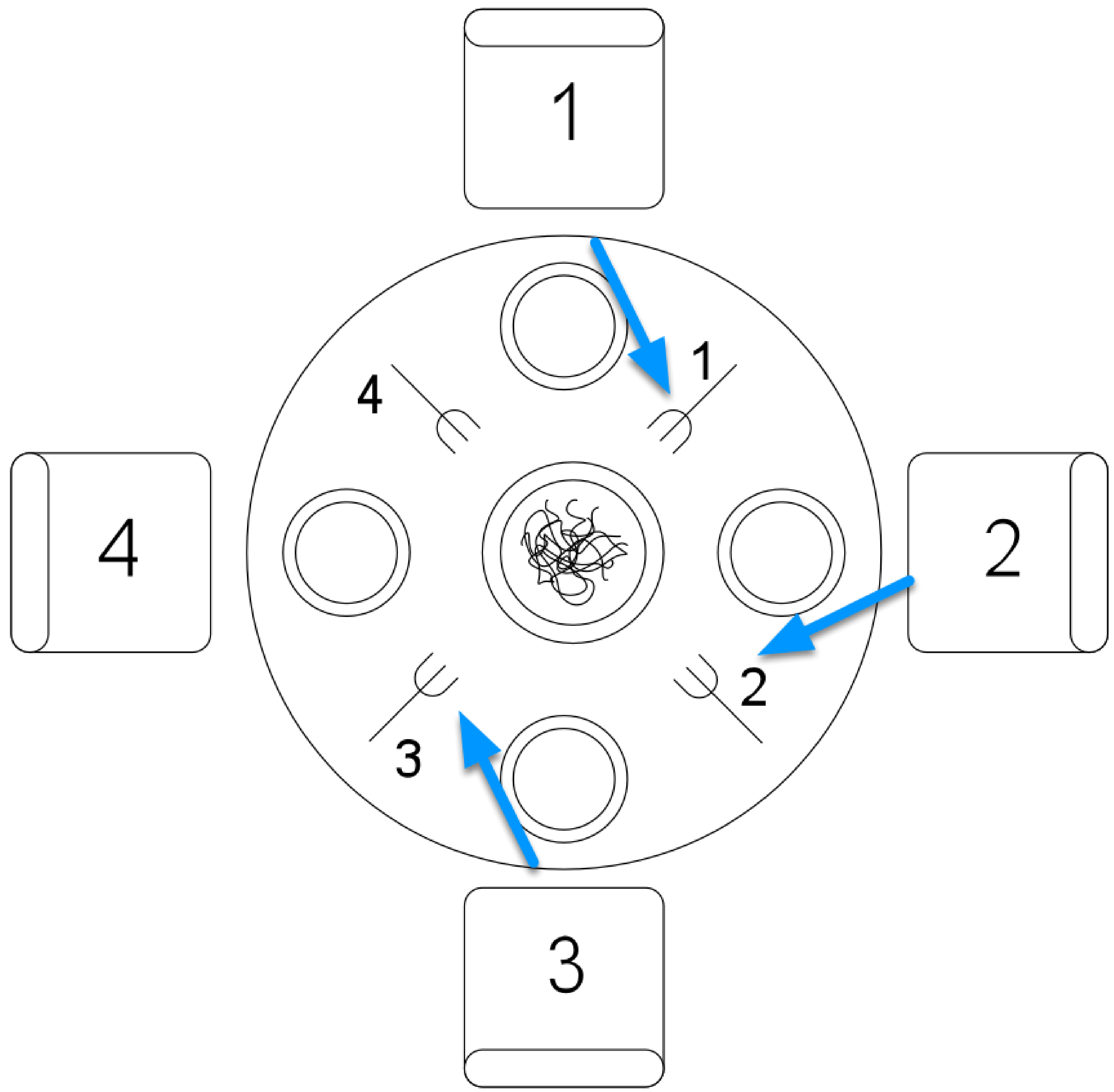
DEADLOCKS: DINING PHILOSOPHERS



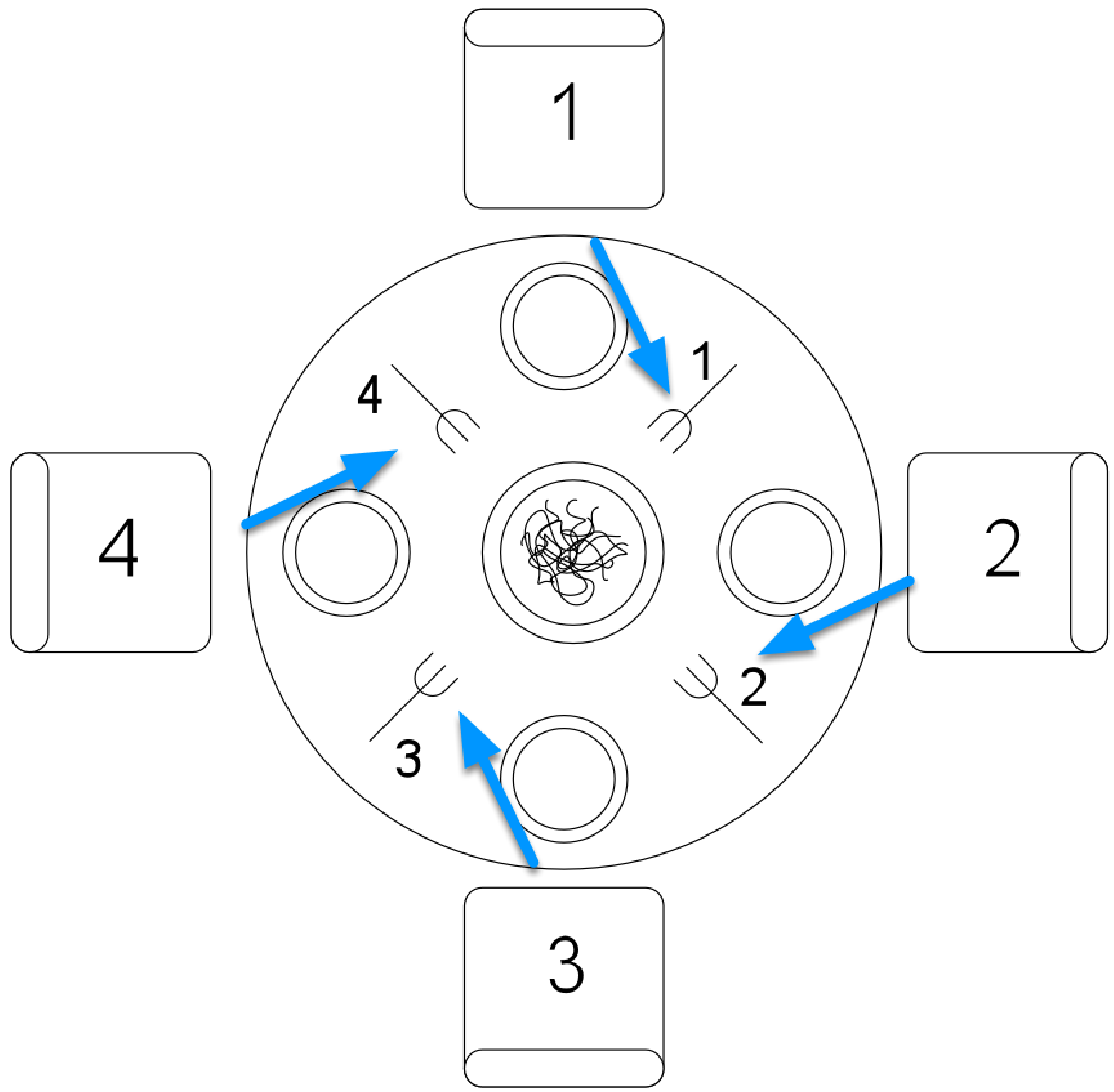
DEADLOCKS: DINING PHILOSOPHERS



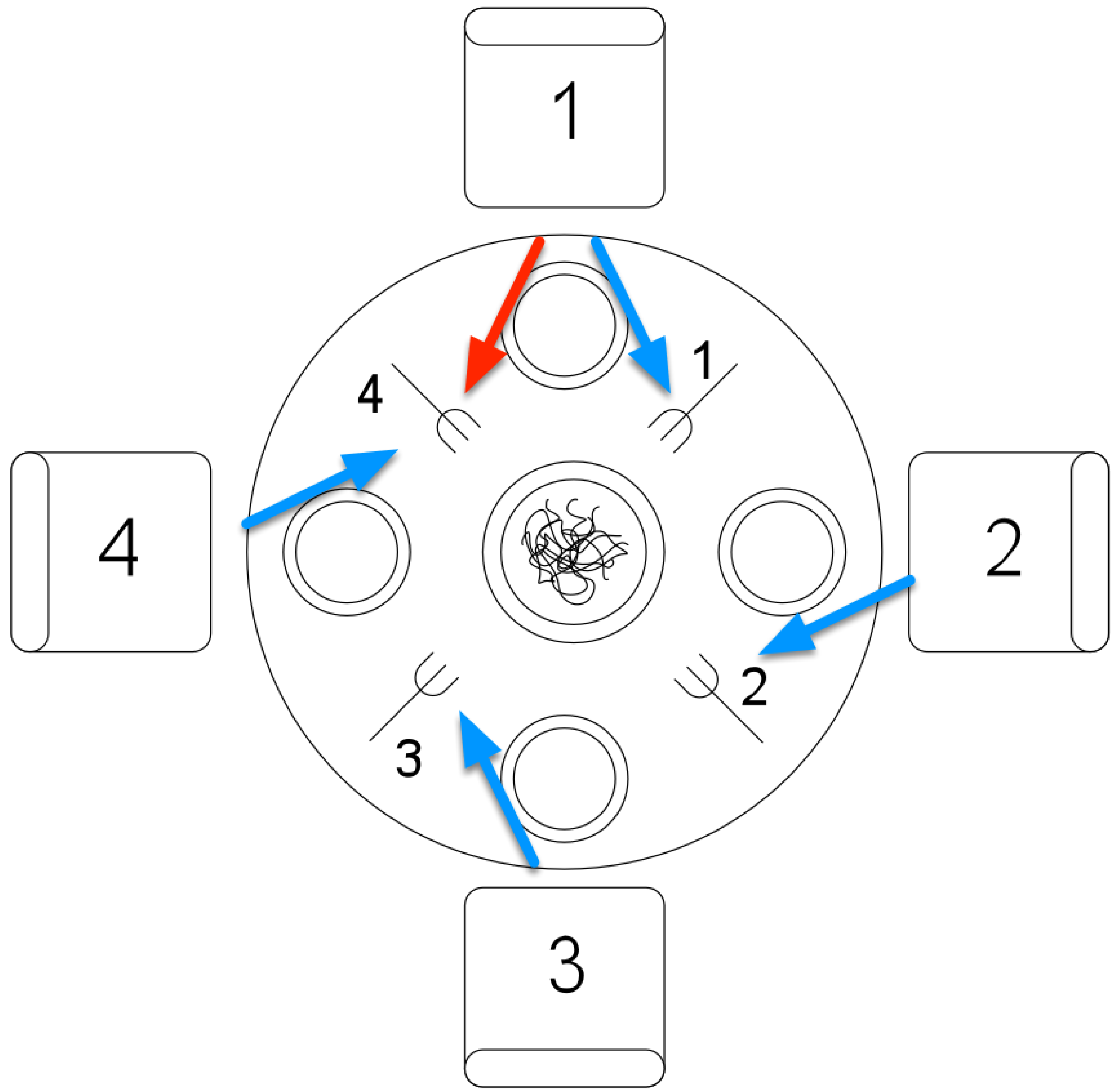
DEADLOCKS: DINING PHILOSOPHERS



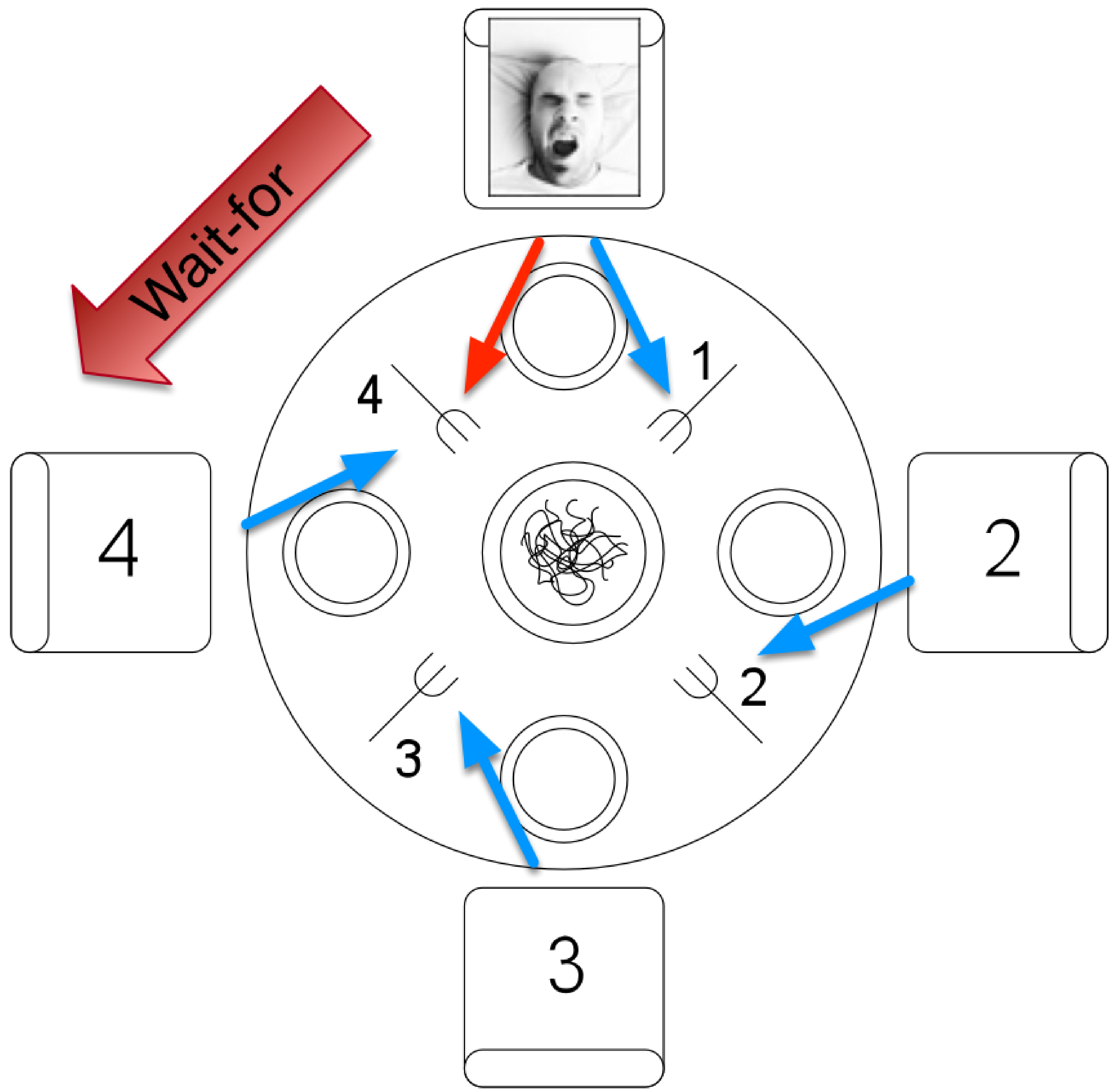
DEADLOCKS: DINING PHILOSOPHERS



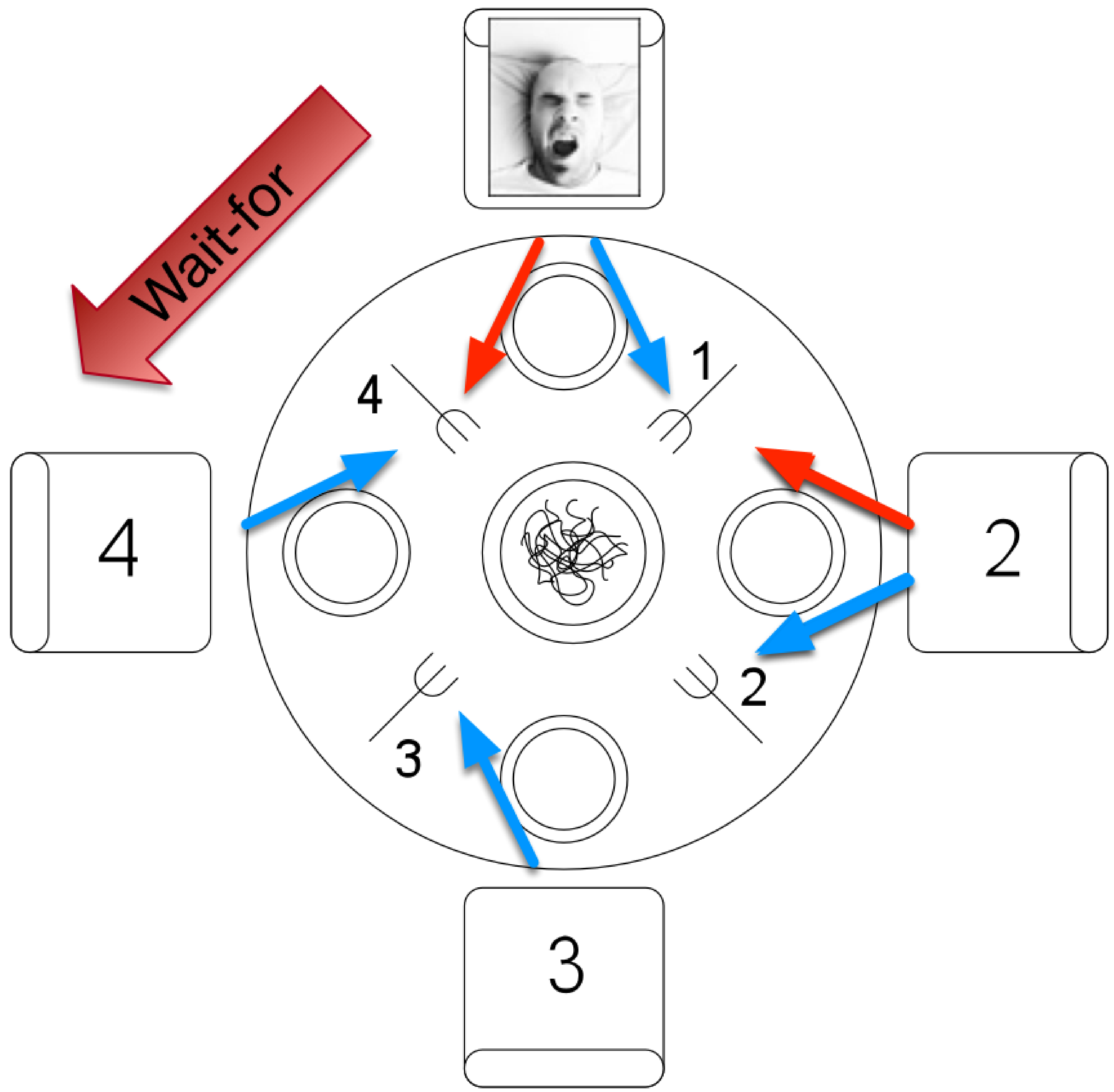
DEADLOCKS: DINING PHILOSOPHERS



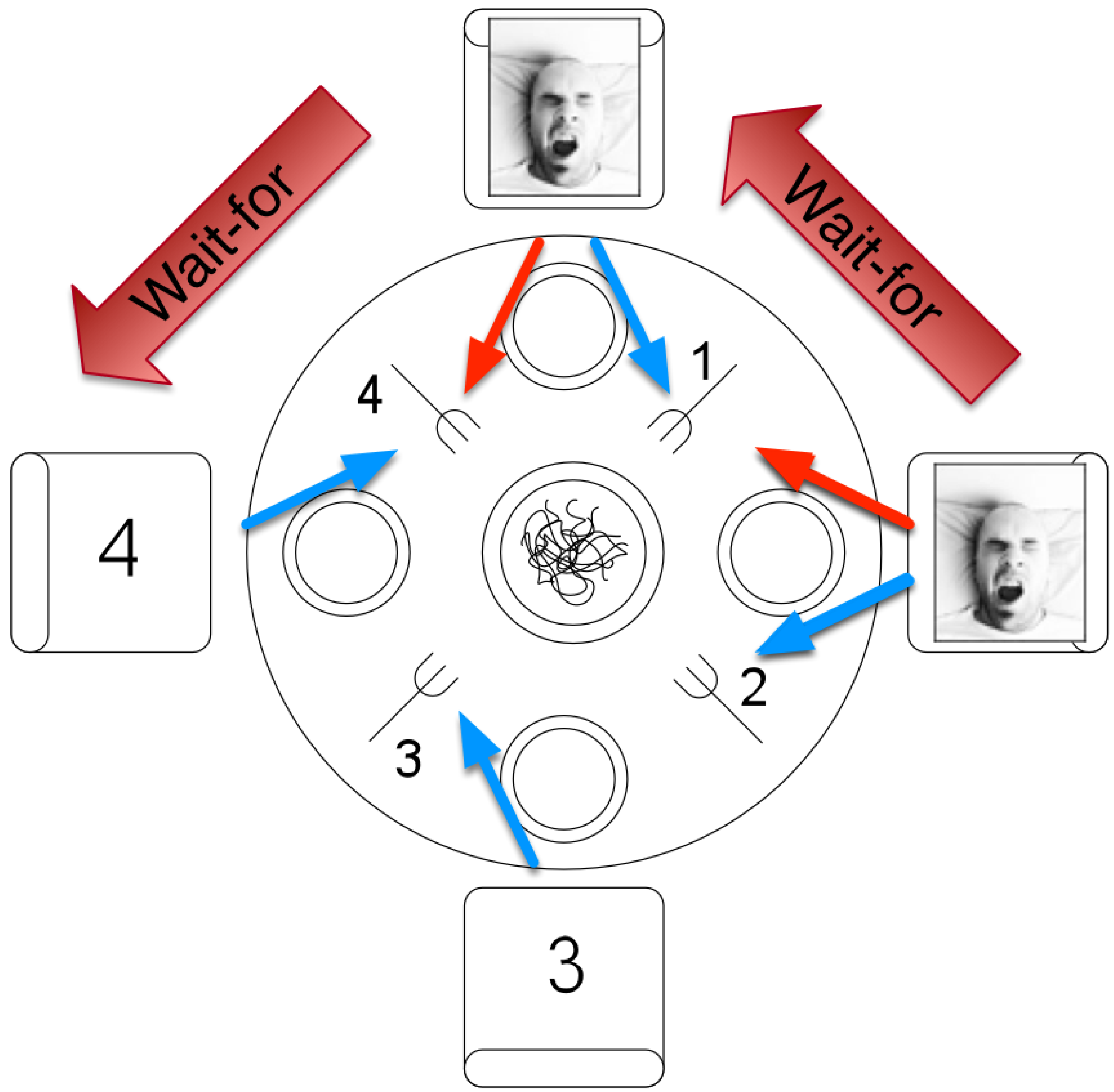
DEADLOCKS: DINING PHILOSOPHERS



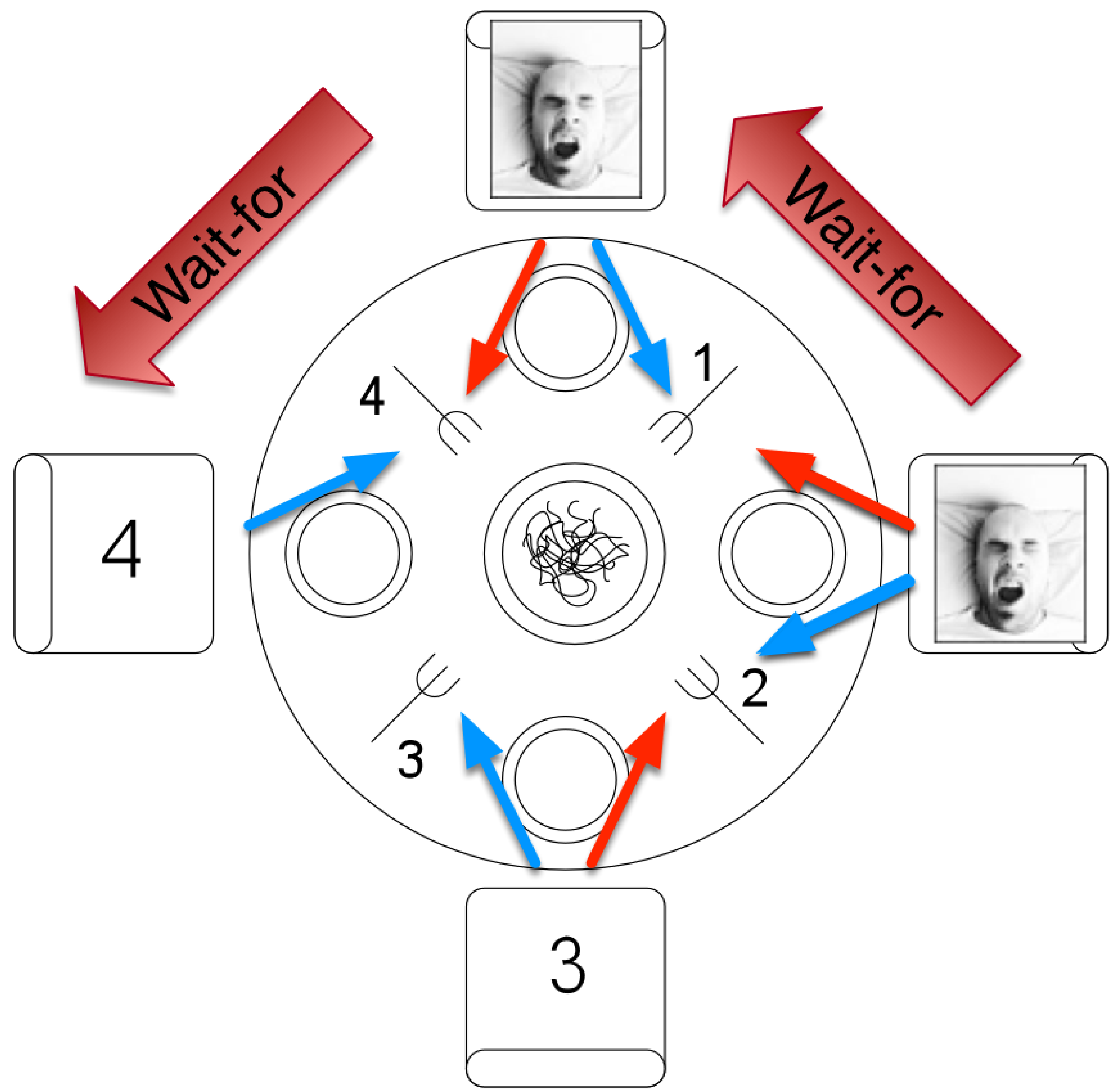
DEADLOCKS: DINING PHILOSOPHERS



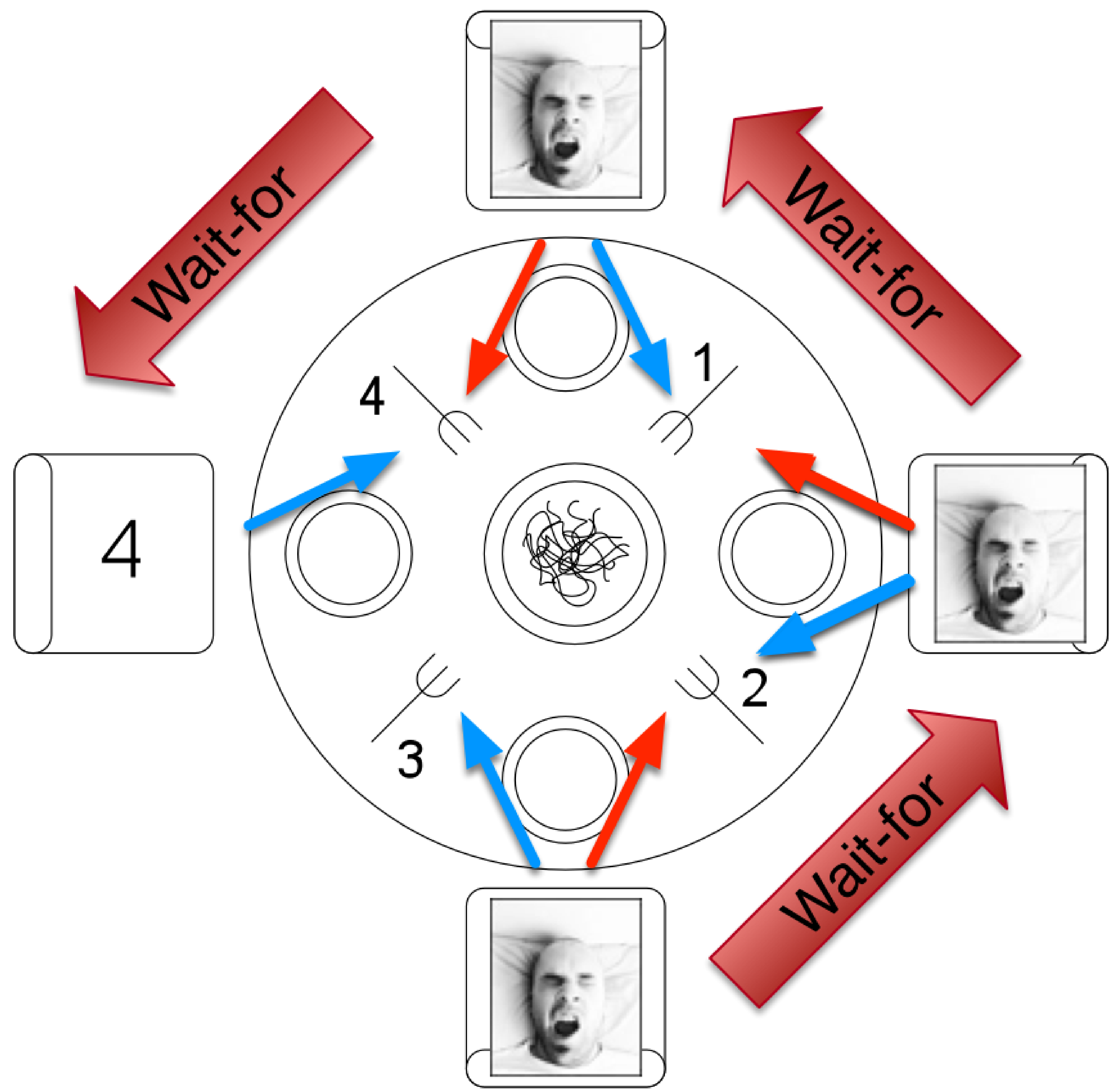
DEADLOCKS: DINING PHILOSOPHERS



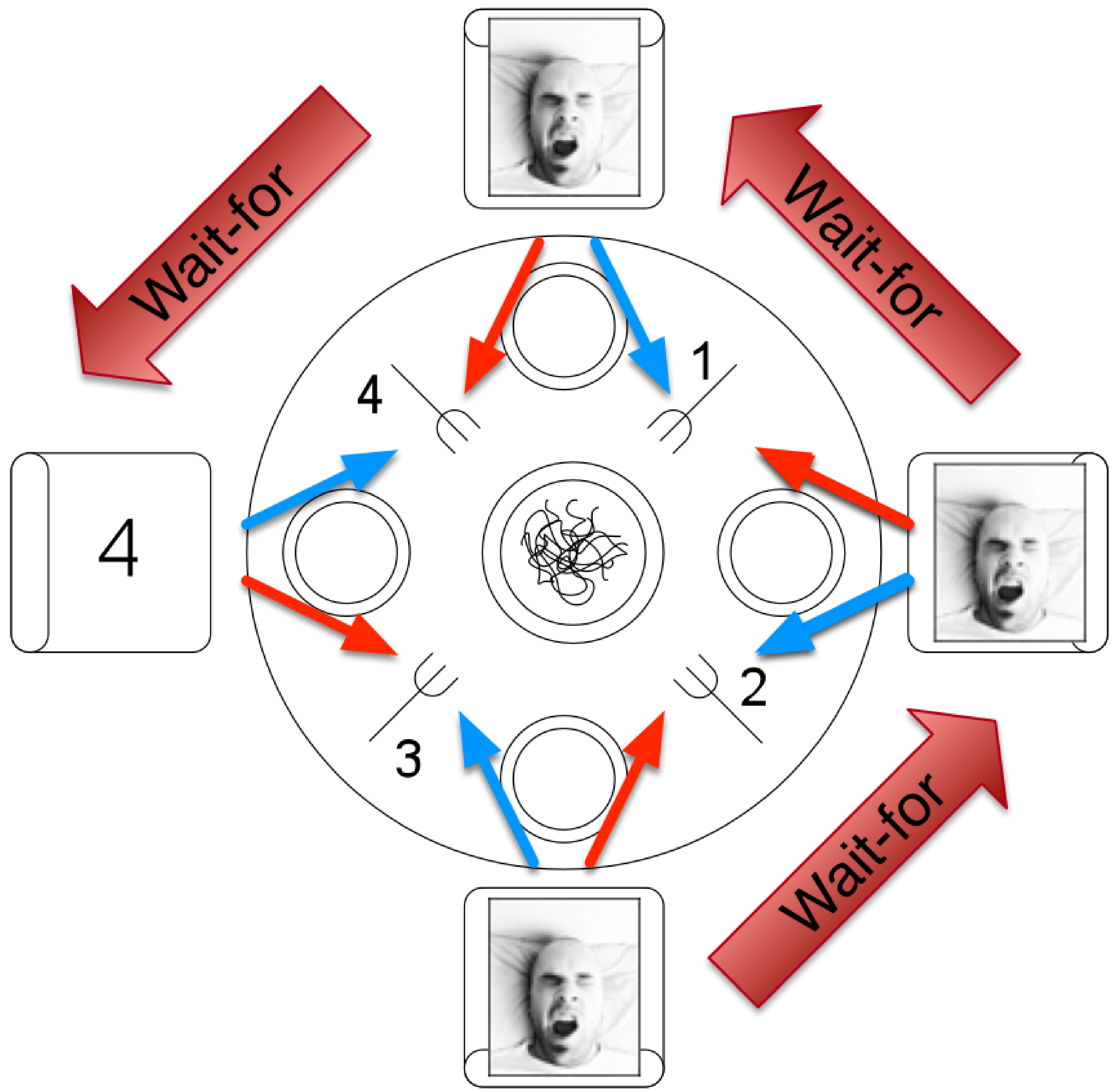
DEADLOCKS: DINING PHILOSOPHERS



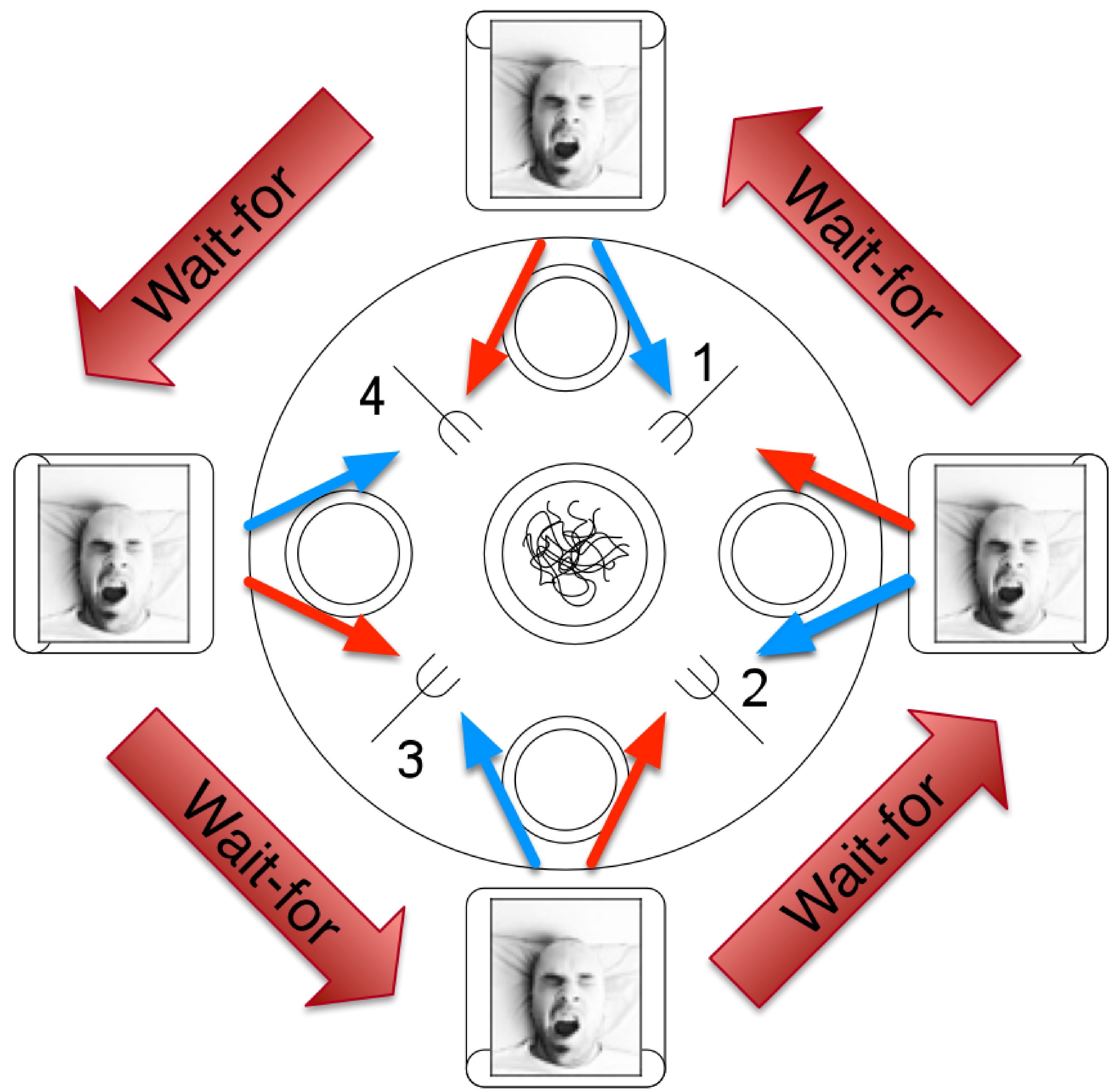
DEADLOCKS: DINING PHILOSOPHERS



DEADLOCKS: DINING PHILOSOPHERS



DEADLOCKS: DINING PHILOSOPHERS



DEADLOCKS: SOLUTIONS

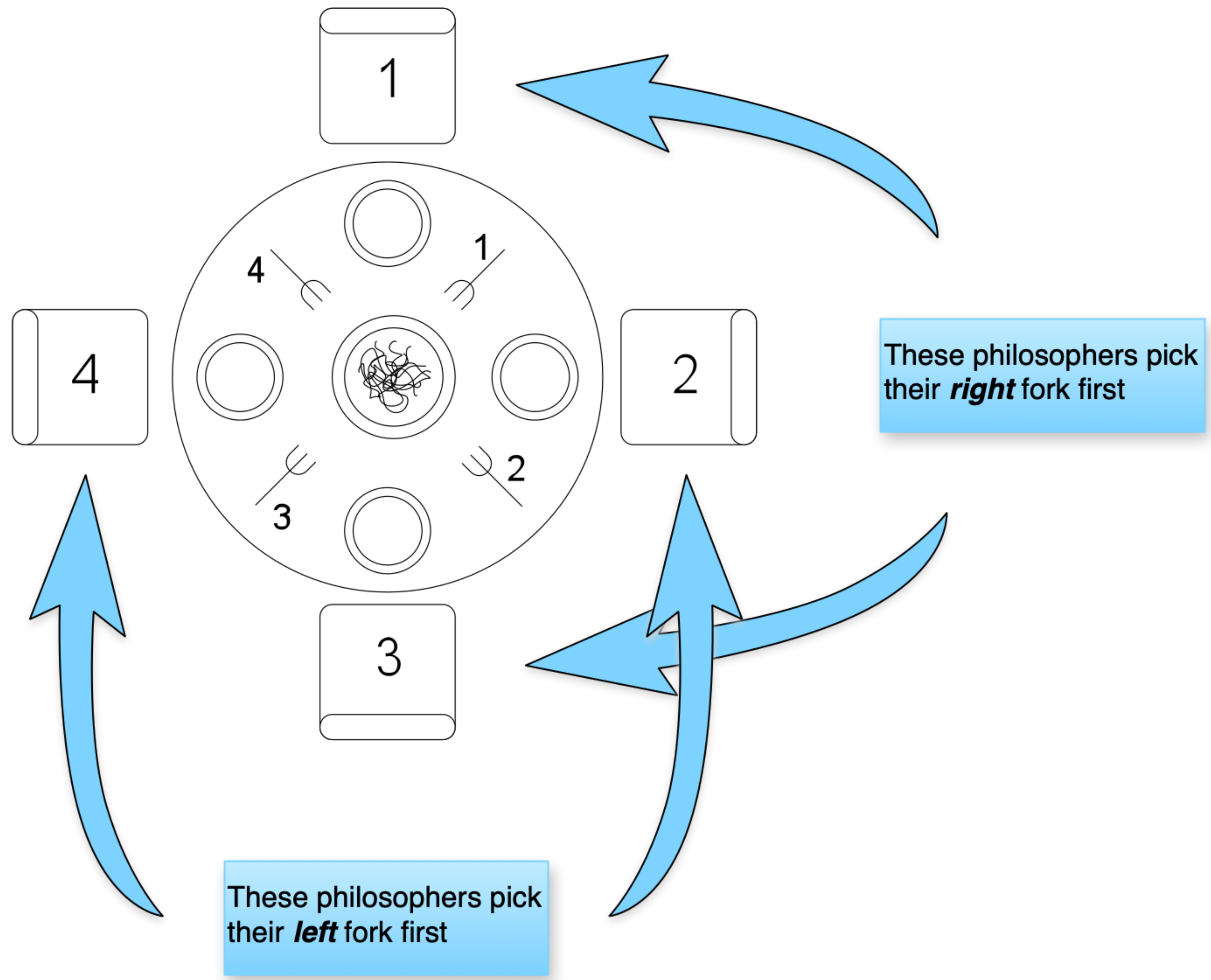
THE SOLUTION TO DEADLOCKS IN GENERAL IS TO REMOVE ONE OF THE FOUR NECESSARY CONDITIONS:

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

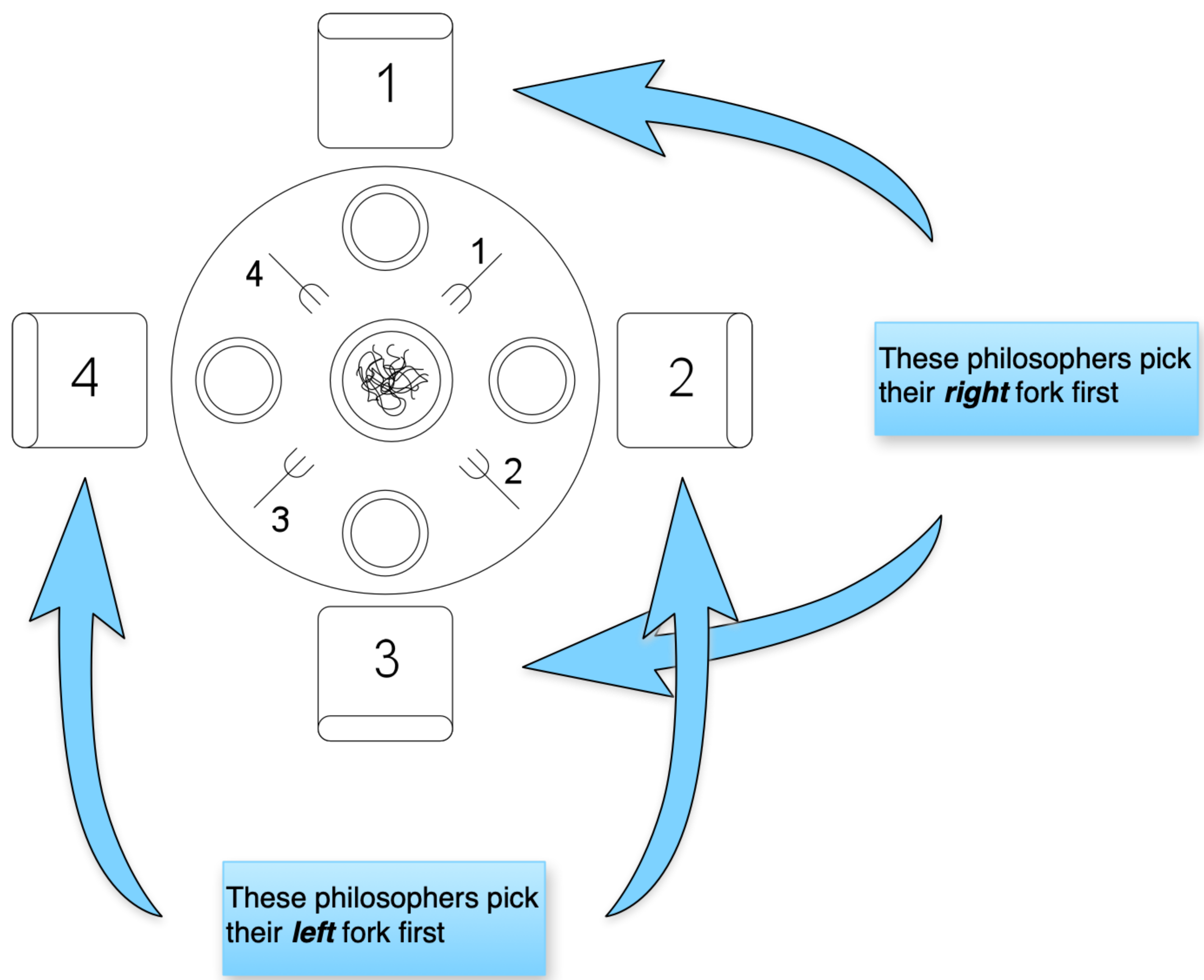
APPLIED TO THE DINING PHILOSOPHER'S PROBLEM: CAN WE REMOVE

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 - A SOLUTION



DINING PHILOSOPHERS - A SOLUTION



Try for yourselves :-D