Condition Variables

Deadlock

CS 241

March 17, 2014

University of Illinois

Slides adapted in part from material accompanying Bryant & O'Hallaron,
"Computer Systems: A Programmer's Perspective", 2/E

Today

Condition Variables (reminder)

- Reader-Writer Problem: a better solution

Deadlock

- Dining Philosophers Problem

Condition Variables (Reminder)

Synchronization primitives

Mutex locks

- Used for exclusive access to a shared resource (critical section)
- Operations: Lock, unlock

Sempahores

- Generalization of mutexes: Count number of available “resources”
- Wait for an available resource (decrement), notify availability (increment)
- Example: wait for free buffer space, signal more buffer space

Condition variables

- Represent an arbitrary event
- Operations: Wait for event, signal occurrence of event
- Tied to a mutex for mutual exclusion

Condition variables

Goal: Wait for a specific event to happen

- Event depends on state shared with multiple threads

Solution: condition variables

- “Names” an event
- Internally, is a queue of threads waiting for the event

Basic operations

- Wait for event
- Signal occurrence of event to one waiting thread
- Signal occurrence of event to all waiting threads

Signaling, not mutual exclusion

- Condition variable is intimately tied to a mutex

Condition variable "Hello world?"

[see code, in class and on web site]

Readers-Writers with Condition Variables

Readers-Writers Problem

Generalization of the mutual exclusion problem

Problem statement:

- *Reader* threads only read the object
- *Writer* threads modify the object
- Writers must have exclusive access to the object
- Unlimited number of readers can access the object

Thread 2

Thread 1	Reader	Writer
Reader	OK	No
Writer	No	No

Recall: Semaphore solution

Shared:

```
int readcnt;    /* Initially = 0 */
sem_t mutex, w; /* Both initially = 1 */
```

Writers:

```
sem_wait(&w);

/* Critical section */
/* Writing here */

sem_post(&w);
```

Readers:

```
sem_wait(&mutex);
readcnt++;
if (readcnt == 1) /* First reader in */
  sem_wait(&w); /* Lock out writers */
sem_post(&mutex);

/* Main critical section */
/* Reading would happen here */

sem_wait(&mutex);
readcnt--;
if (readcnt == 0) /* Last out */
  sem_post(&w); /* Let in writers */
sem_post(&mutex);
```

(full code online)

Condition variable solution

Idea:

- If it's safe, just go ahead and read or write
- Otherwise, wait for my "turn"

Initialization:

```
/* Global variables */
pthread_mutex_t m;
pthread_cond_t  turn; /* Event: it's our turn */
int writing;
int reading;

void init(void) {
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&turn, NULL);
    reading = 0;
    writing = 0;
}
```

Condition variable solution

```
void reader(void)
{
    mutex_lock(&m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

(Note: "pthread_" prefix removed from all synchronization calls for compactness)

Familiar problem: Starvation

```
void reader(void)
{
    mutex_lock(&m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

(Note: "pthread_" prefix removed from all synchronization calls for compactness)

Idea: take turns

If a writer is waiting, then reader should wait its turn

- Even if it's safe to proceed (only readers are in critical section)

Requires keeping track of waiting writers

```
/* Global variables */
pthread_mutex_t m;
pthread_cond_t  turn; /* Event: someone else's turn */
int reading;
int writing;
int writers;

void init(void) {
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&turn, NULL);
    reading = 0;
    writing = 0;
    writers = 0;
}
```

Taking turns

```c
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```c
void writer(void)
{
    mutex_lock(&m);
    writers++;
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    writers--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

Another problem :-(

```
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    writers++;
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    writers--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

Only unblocks one thread at a time;
Inefficient if many readers are waiting

Easy solution: Wake everyone

```c
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_broadcast(&turn);
    mutex_unlock(&m);
}
```

```c
void writer(void)
{
    mutex_lock(&m);
    writers++;
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    writers--;
    cond_broadcast(&turn);
    mutex_unlock(&m);
}
```

Semaphores vs. Condition Variables

Semaphore

- Integer value (≥ 0)
- Wait doesn't always block
- Signal either un-blocks thread or increments counter
- If signal releases thread, both may continue concurrently

Condition Variable

- No value
- Wait always blocks
- Signal either un-blocks thread or is lost
- If signal releases thread, only one continues
  - Need to hold mutex lock to proceed
  - Other thread is released from waiting on condition, but still has to wait to obtain the mutex again

Conclusion

Condition variables

- convenient way of signaling general-purpose events between threads

Common implementation: “monitors”

- An object which does the locking/unlocking for you when its methods are called
- See `synchronized` keyword in Java

Beware pitfalls...

Pitfalls

signal() before wait()

- Waiting thread will miss the signal

Fail to lock mutex before calling wait()

- Might return error, or simply not block

`if (!condition) wait();` instead of `while (!condition) wait();`

- condition may still be false when wait returns!
- can lead to arbitrary errors (e.g., following NULL pointer, memory corruption, ...)

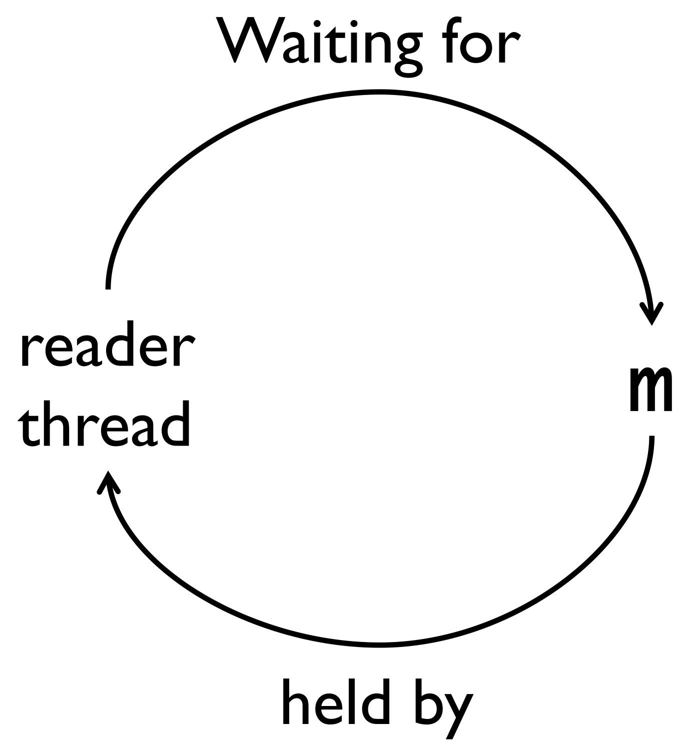
Forget to unlock mutex

- uh oh...

Forgetting to unlock the mutex

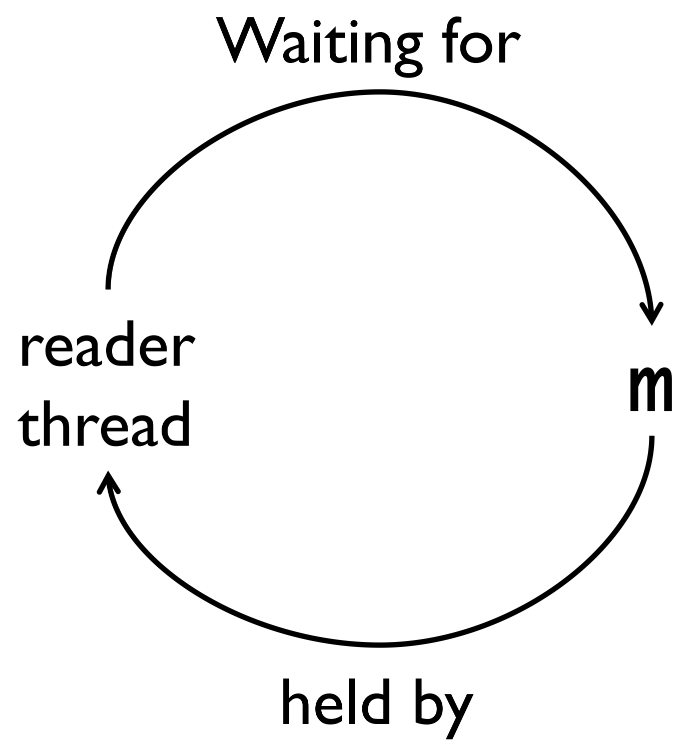
```
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_broadcast(&turn);
    mutex_unlock(&m);
}

while (1) { reader() };
```

After running once, next time reader calls `mutex_lock(&m)`:



Forgetting to unlock the mutex

After running once,
next time reader calls
`mutex_lock(&m)`:



DEADLOCK

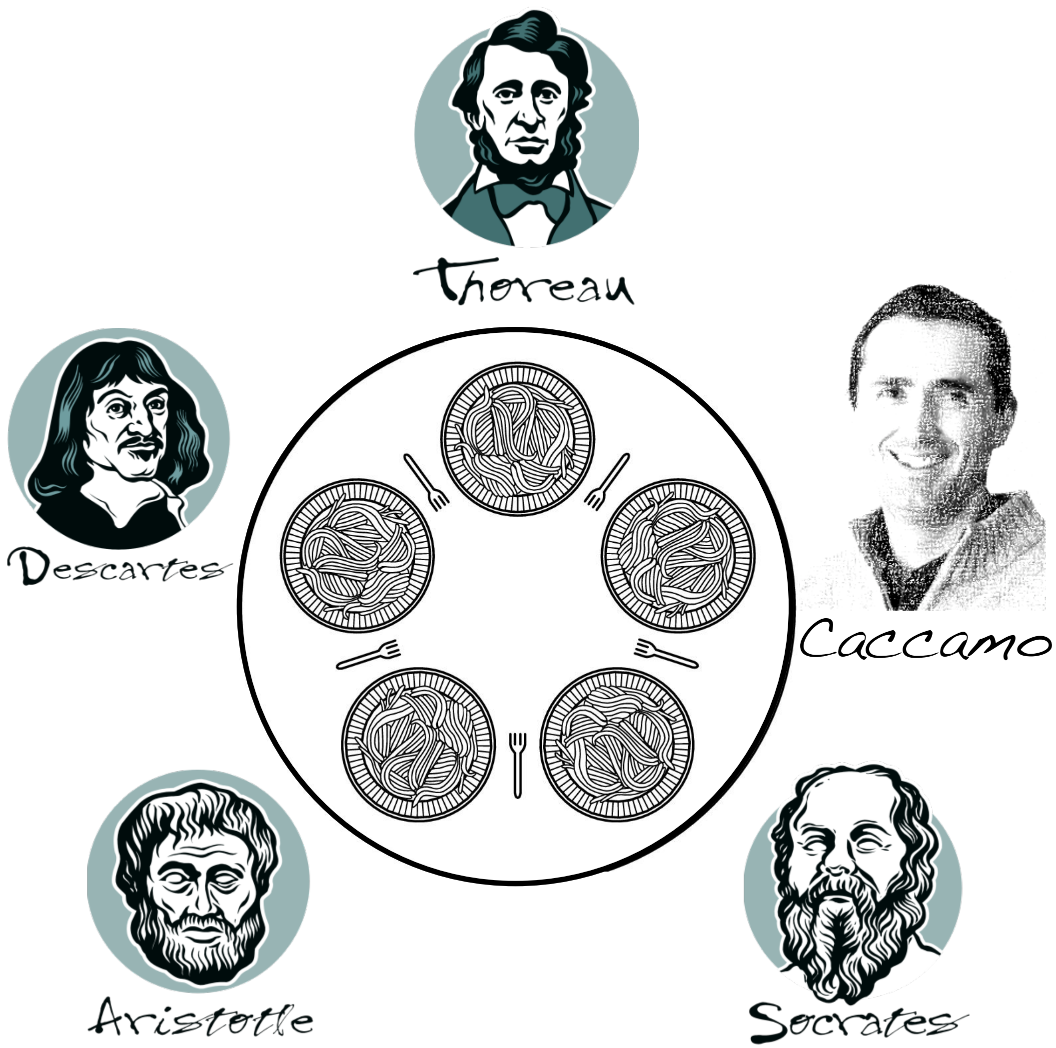
thread waits forever
for event that will
never happen

The
Dining Philosophers
Problem

Drinking Philosophers

[illegible] t N[illegible]
of [illegible]ov[illegible]ee[illegible]ill.
C[illegible]nal[illegible]int[illegible]f[illegible]a[illegible]
was p[illegible]icul[illegible]ill ...

Dining Philosophers



Dining Philosophers

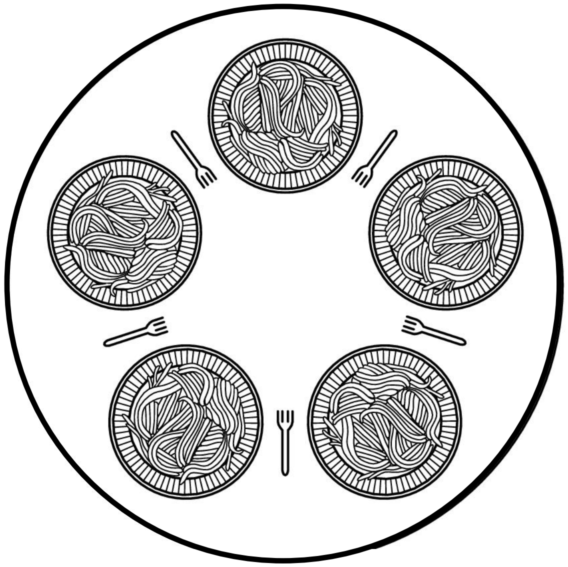
N philosophers and N forks

Philosophers eat, think

Eating needs 2 forks

Pick up one fork at a time

Each fork used by one person at a time



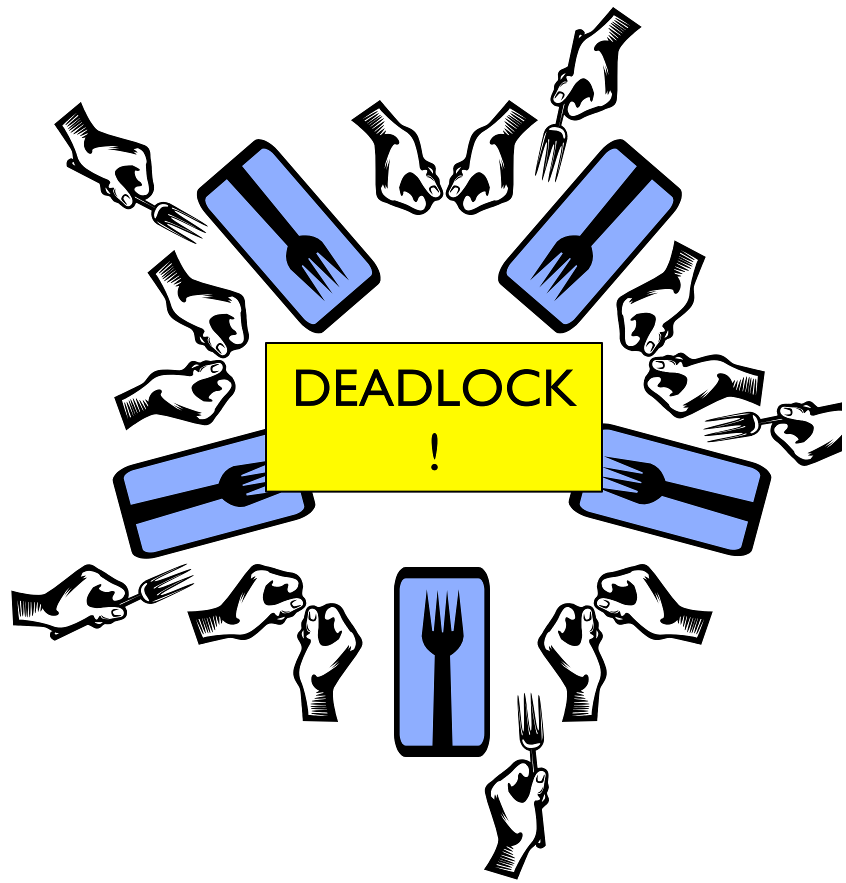
Dining Philosophers: Take 1

```
# define N 5

void philosopher (int i) {
   while (TRUE) {
      think();
      lock_fork(i);
      lock_fork((i+1)%N);

      eat(); /* yummy */

      unlock_fork(i);
      unlock_fork((i+1)%N);
   }
}
```

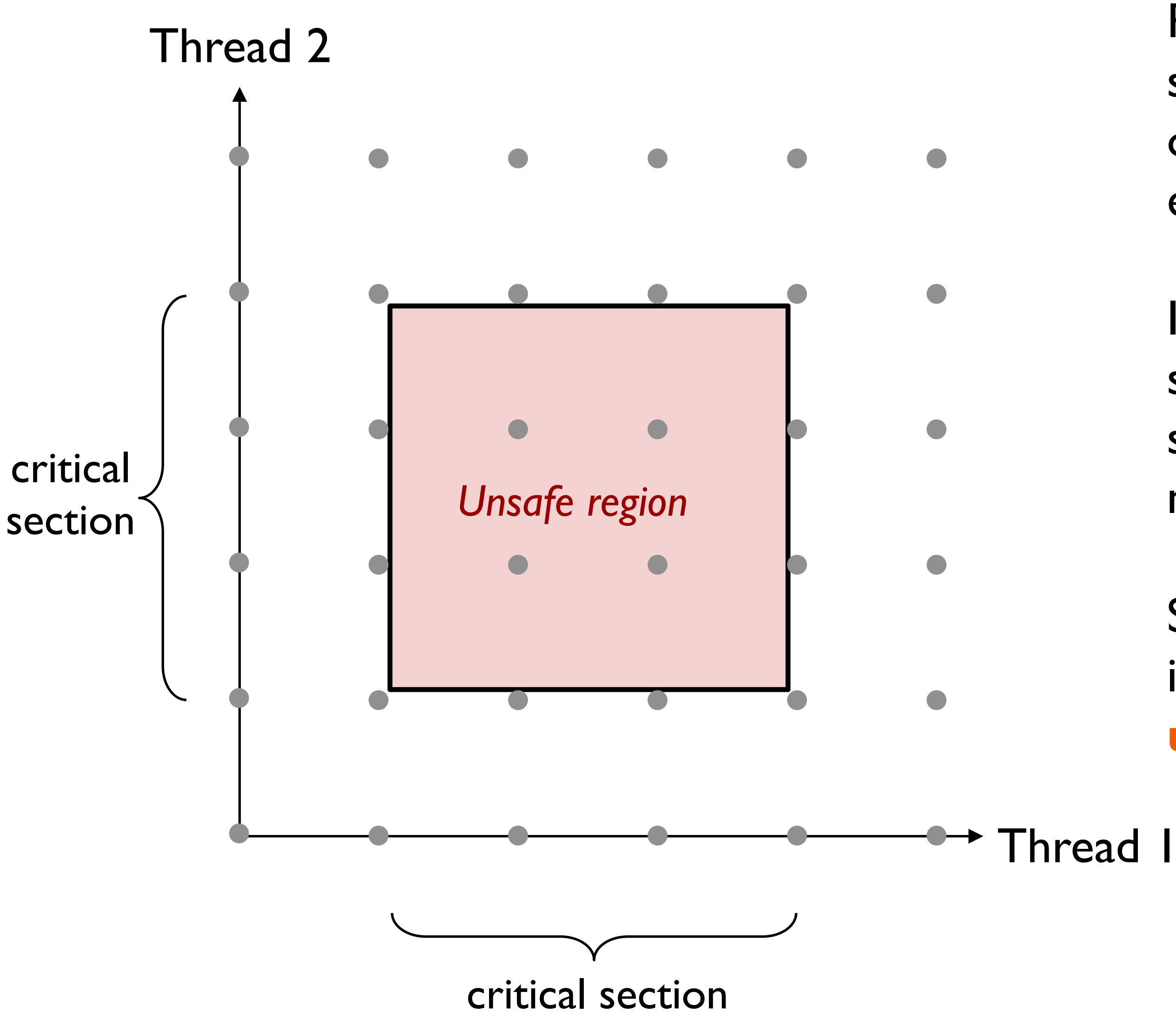
Does this work?

Dining Philosophers: Take 1

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        lock_fork(i);
        lock_fork((i+1)%N);

        eat(); /* yummy */

        unlock_fork(i);
        unlock_fork((i+1)%N);
    }
}
```



Progress diagram

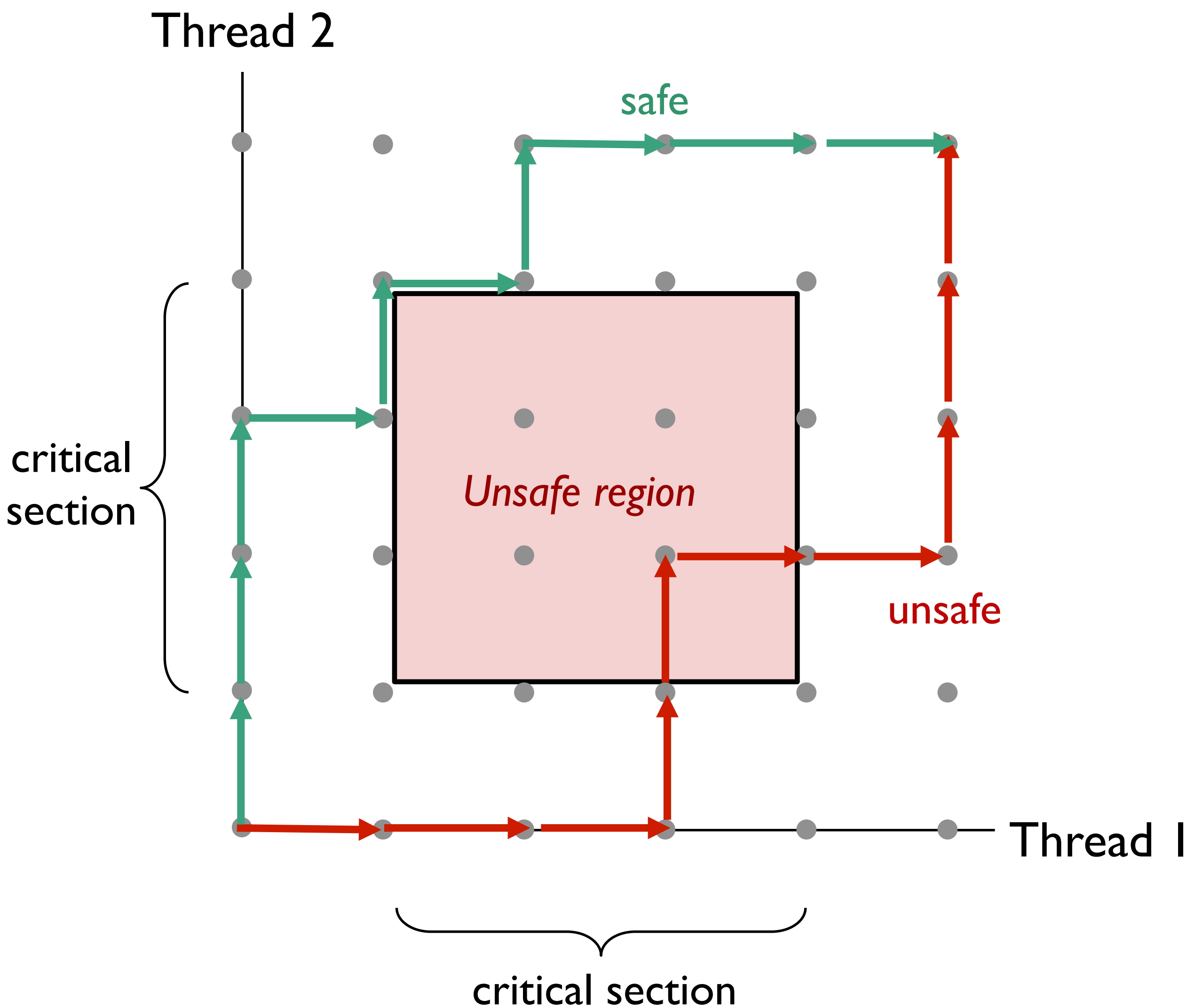


Represents state of system (position of each of the two threads in their executions)

Instructions in critical sections (wrt to some shared variable) should not be interleaved

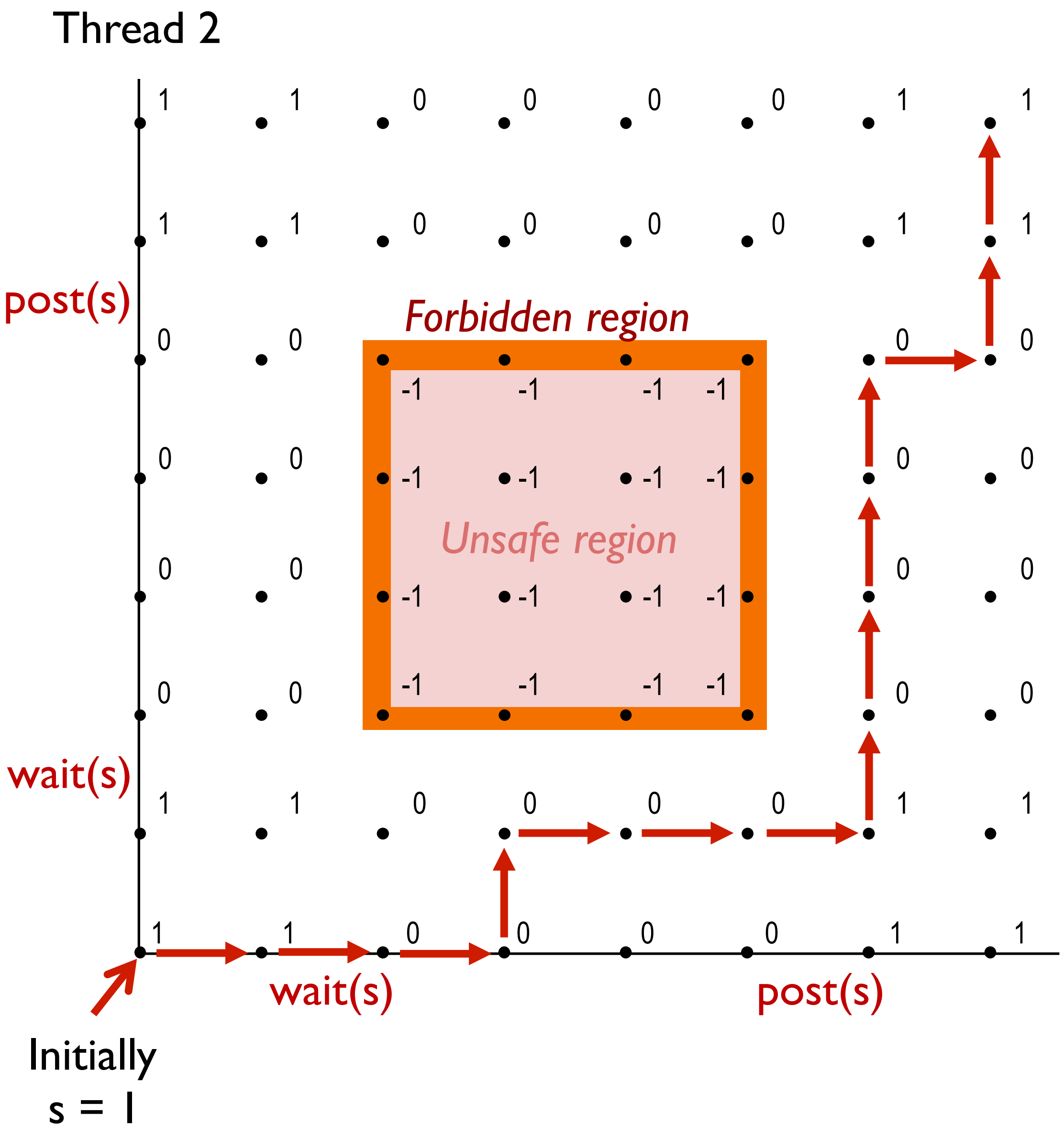
Sets of states where such interleaving occurs form **unsafe regions**

Progress diagram



But, any trajectory that goes up and to the right might occur...

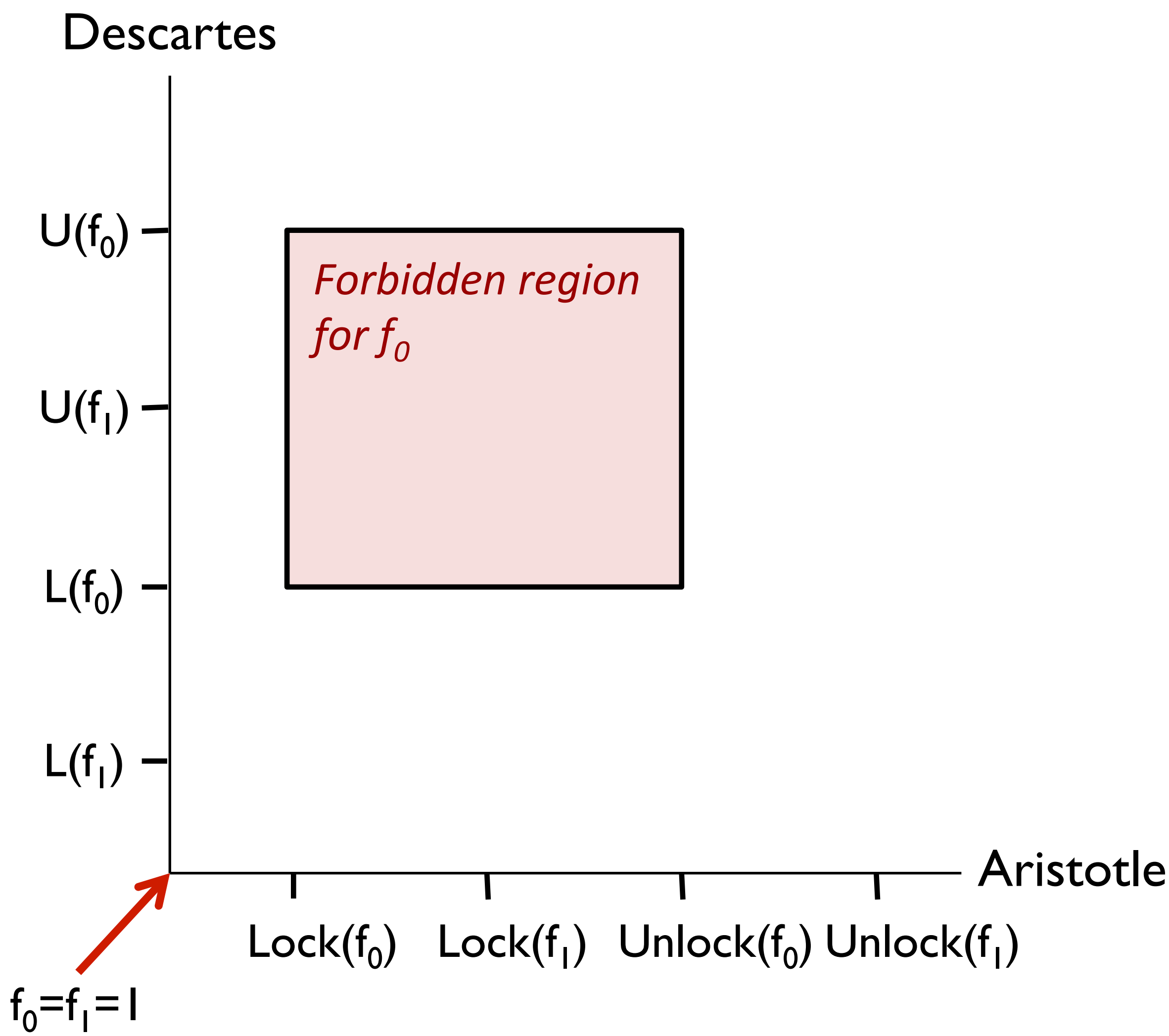
Reminder: process diagram



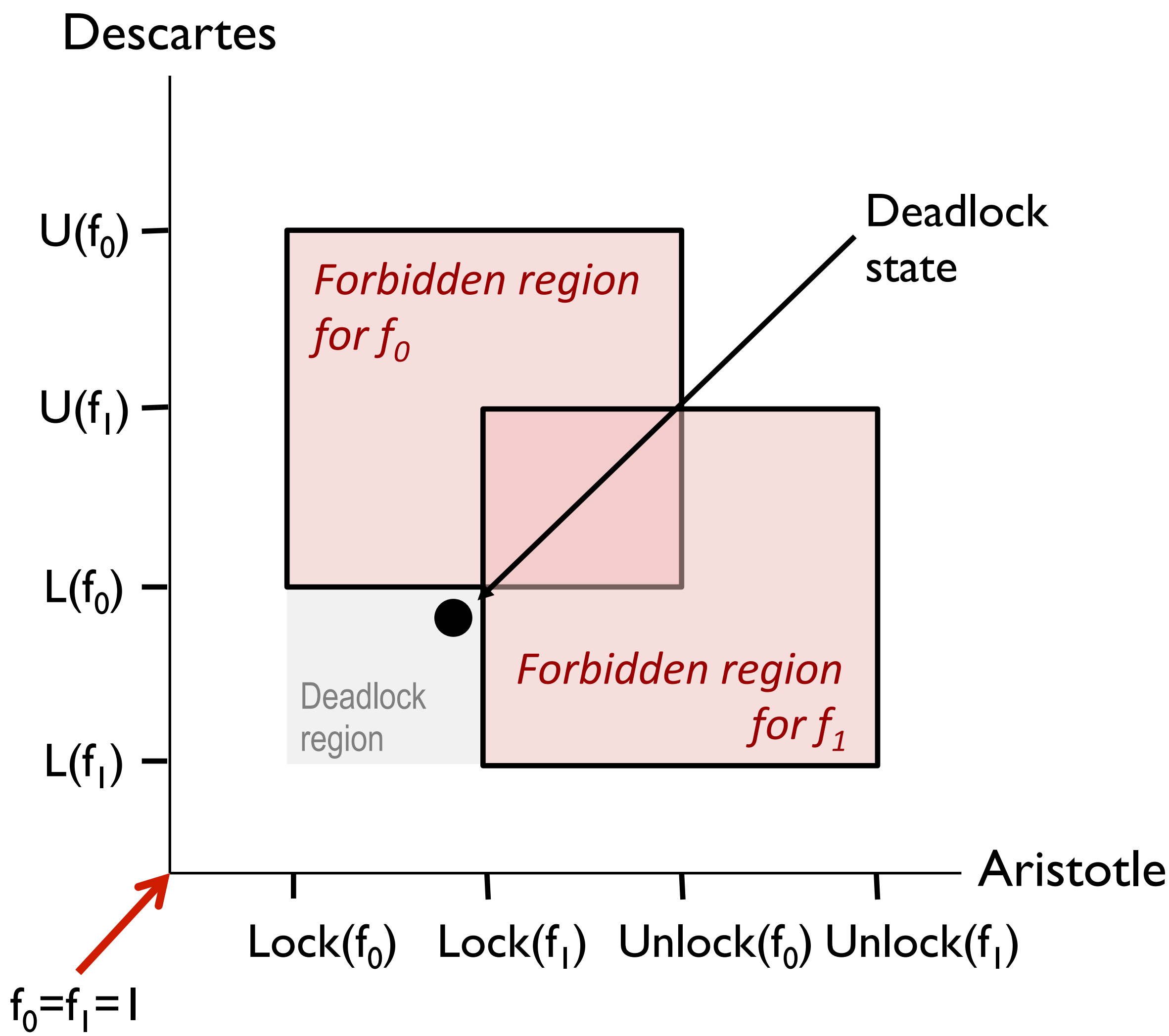
Mutexes provide mutually exclusive access to shared variable by surrounding critical section with *wait* and *post* operations on semaphore s (initially set to 1)

Semaphore invariant creates a **forbidden region** that encloses the unsafe region that must not be entered by any trajectory.

Two shared resources



Two shared resources



Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state,* waiting for either f_0 or f_1 to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often nondeterministic (race)

Deadlock: definition

There exists a cycle of processes such that each process cannot proceed until the next process takes some specific action.

Result: all processes in the cycle are stuck!

Example:

- P1 holds resource R1 & is waiting to acquire R2 before unlocking them
- P2 holds resource R2 & is waiting to acquire R1 before unlocking them

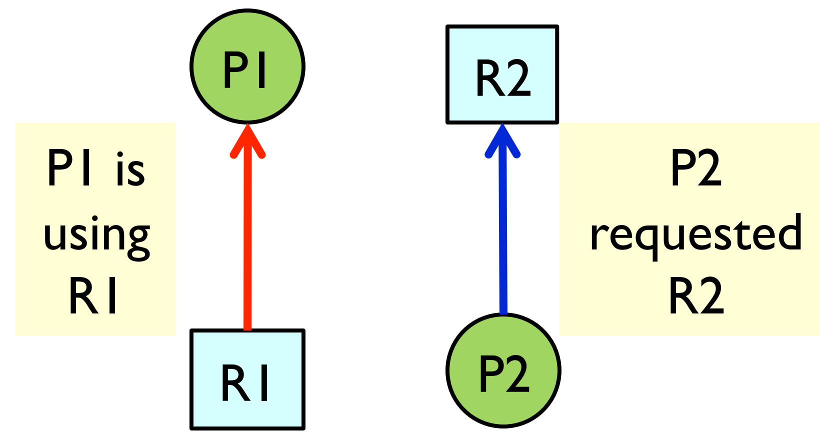
Resource allocation graphs

Nodes

- Circle: Processes
- Square: Resources

Arcs

- From resource to process = resource assigned to process
- From process to resource = process requests (and is waiting for) resource



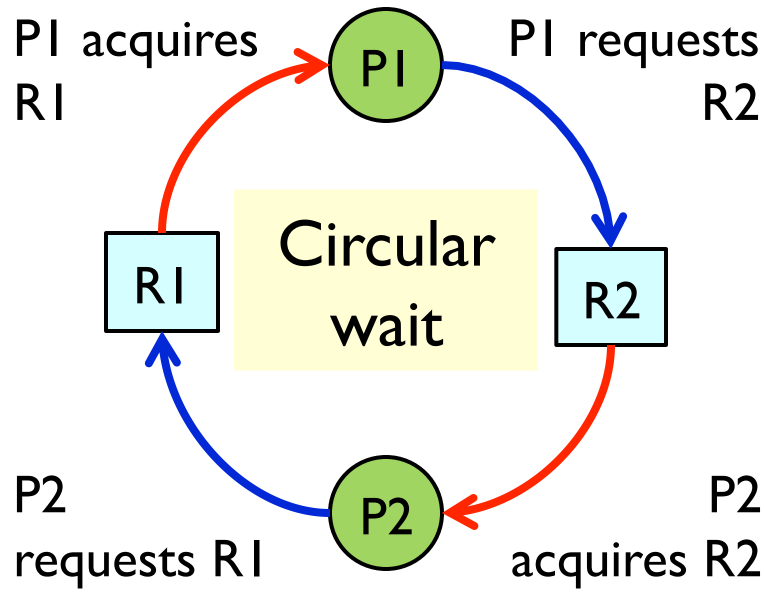
Resource allocation graphs

Nodes

- Circle: Processes
- Square: Resources

Deadlock

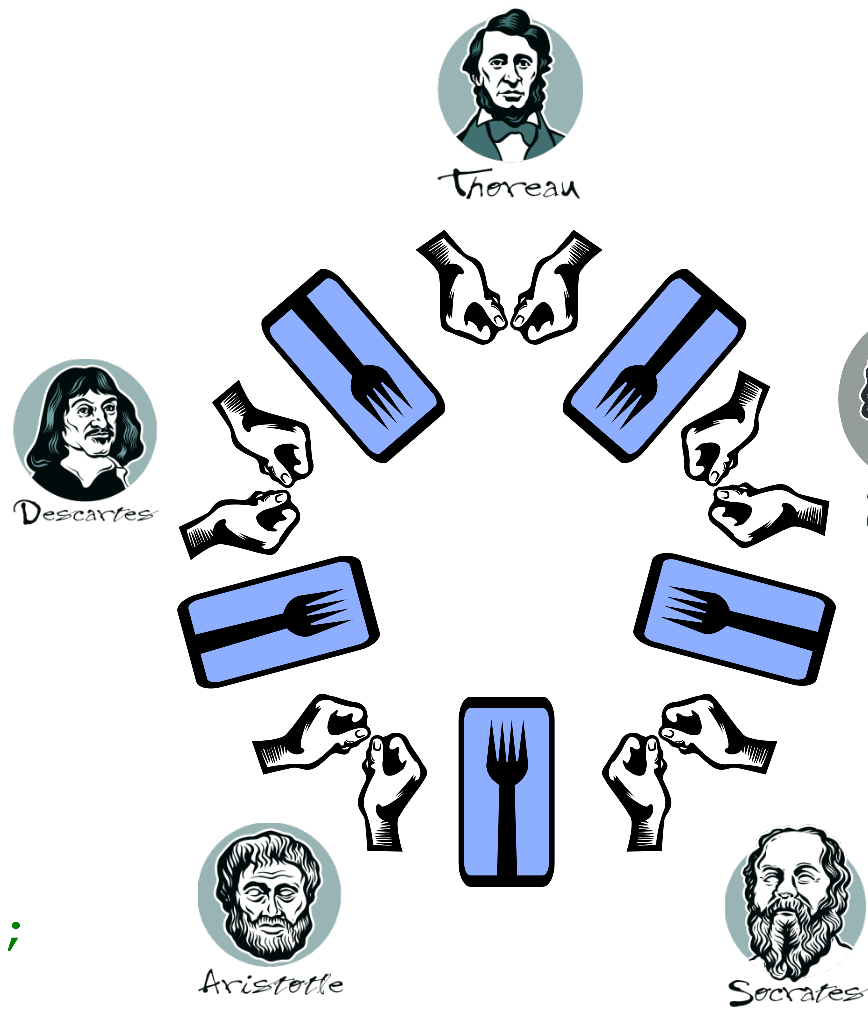
- Processes P1 and P2 are in deadlock over resources R1 and r2



Dining Philosophers resource allocation graph

If we use the trivial broken "solution"...

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        lock_fork(i);
        lock_fork((i+1)%N);
        eat(); /* yummy */
        unlock_fork(i);
        unlock_fork((i+1)%N);
   }
}
```



Dining Philosophers resource allocation graph

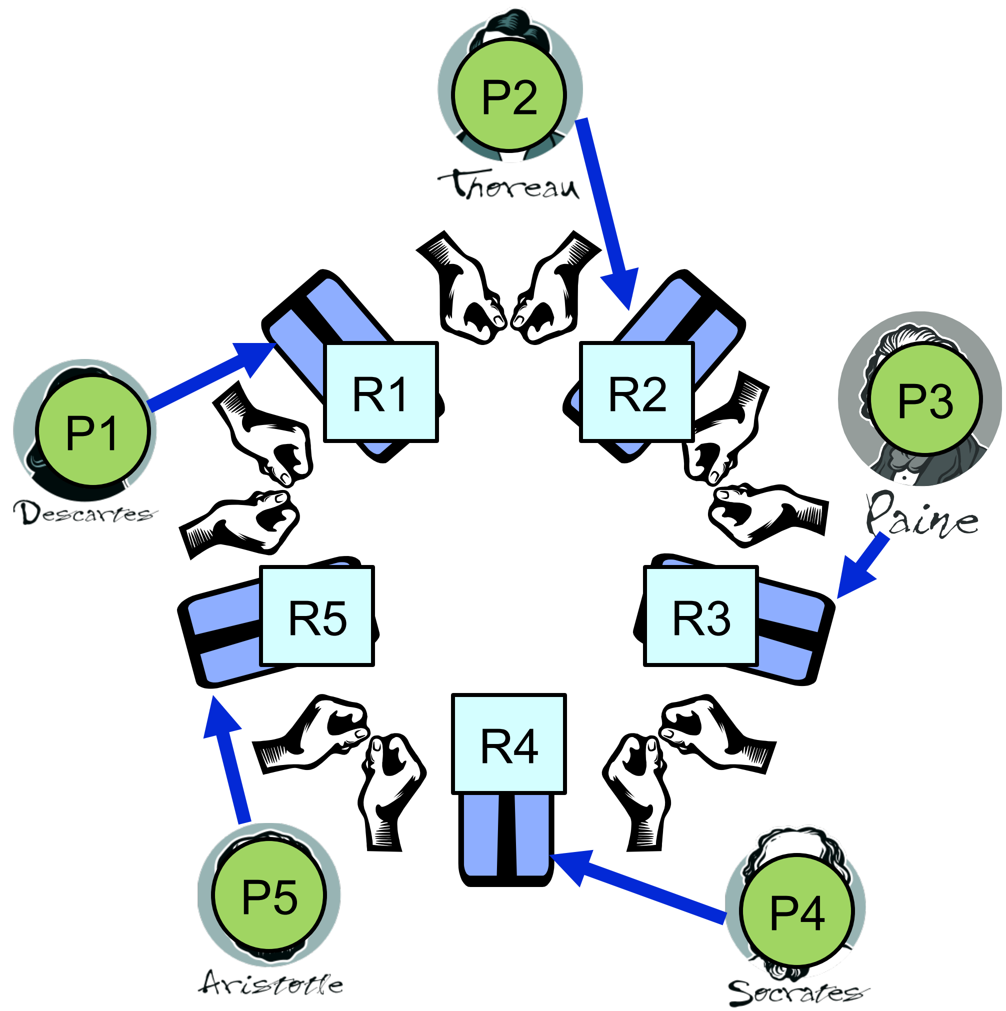
If we use the trivial broken “solution”...

One node per philosopher

One node per fork

Everyone tries to pick up left fork

- Result: Request edges



Dining Philosophers resource allocation graph

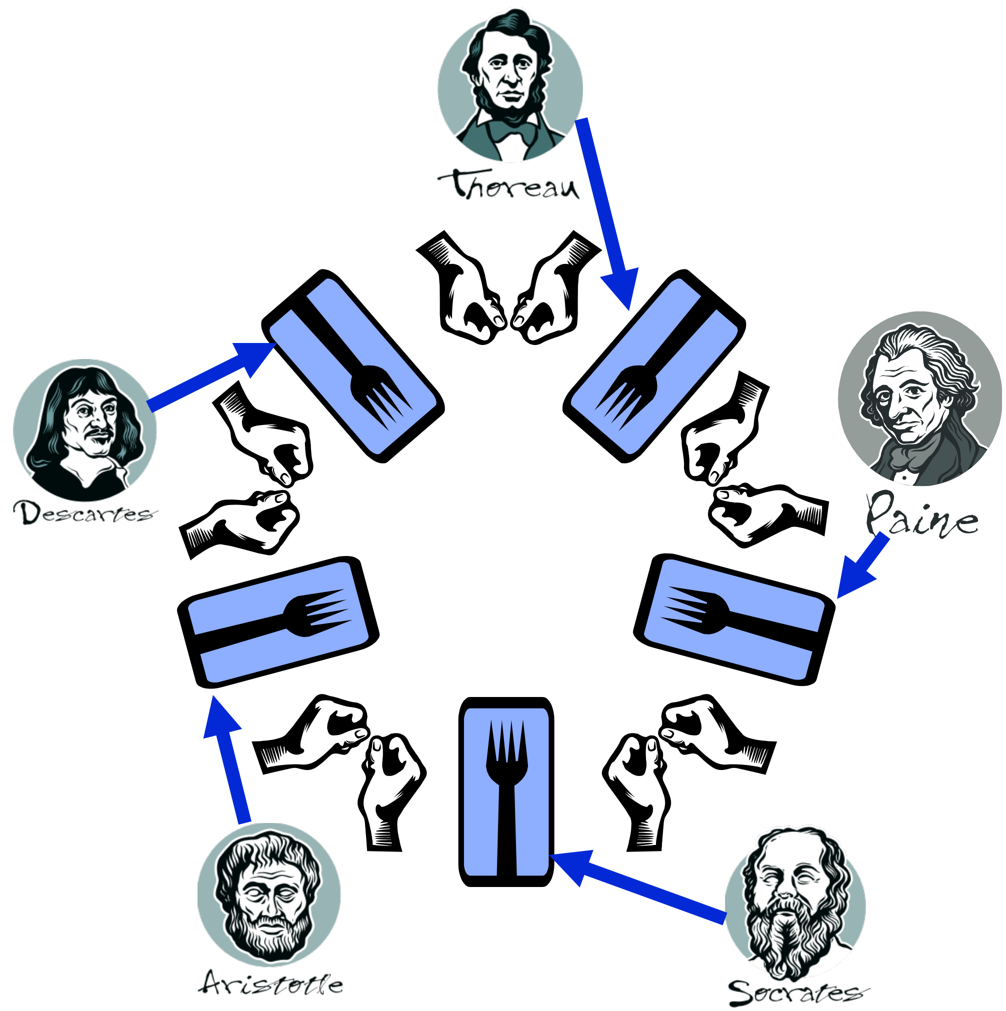
If we use the trivial broken "solution"...

One node per philosopher

One node per fork

Everyone tries to pick up left fork

- Result: Request edges
- Everyone succeeds!



Dining Philosophers resource allocation graph

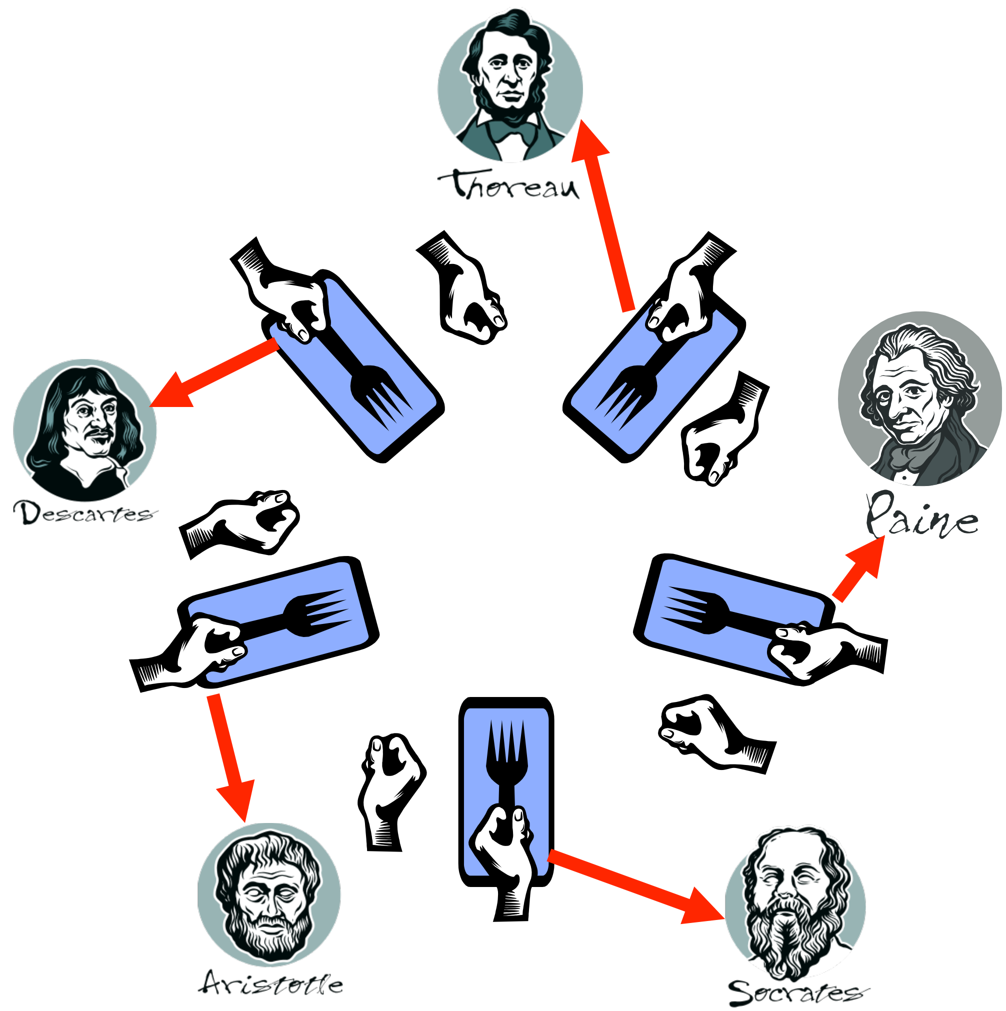
If we use the trivial broken "solution"...

One node per philosopher

One node per fork

Everyone tries to pick up left fork

- Result: Request edges
- Everyone succeeds!
- Result: Assignment edges



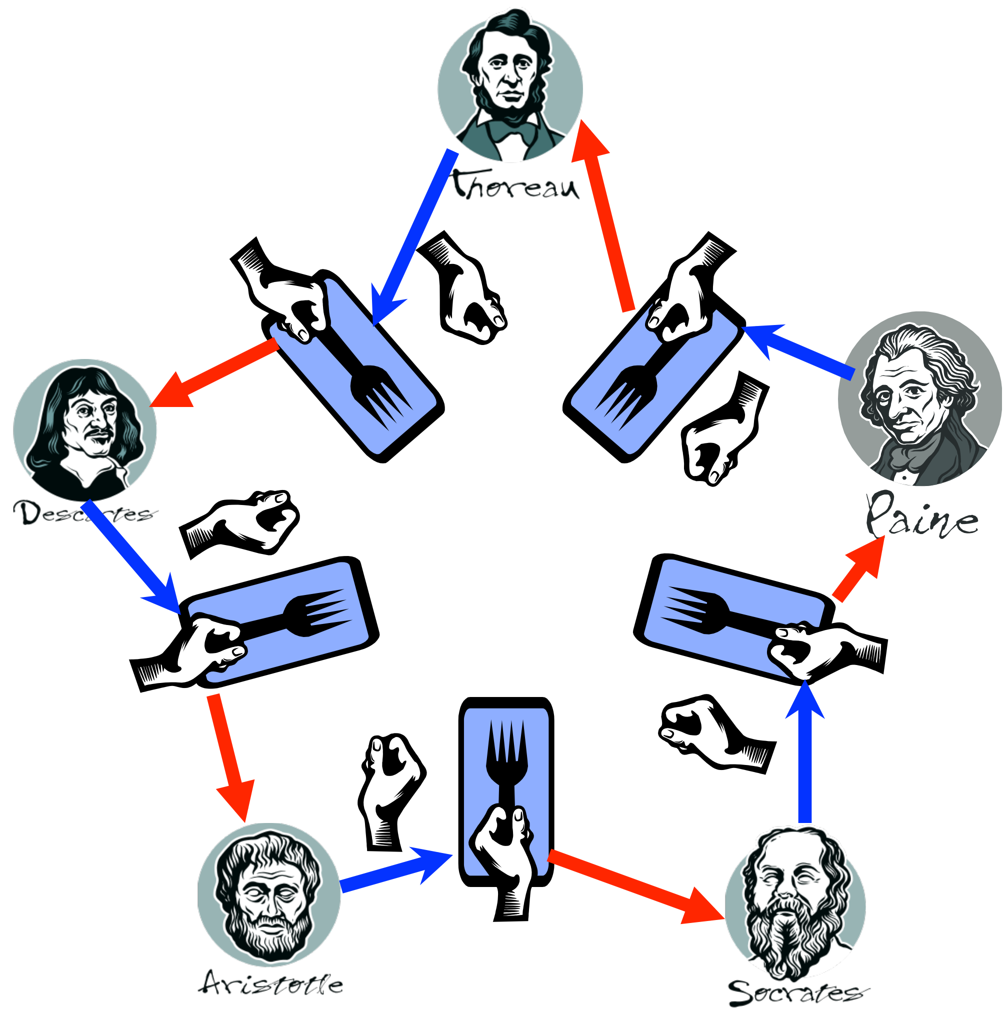
Dining Philosophers resource allocation graph

Everyone tries to pick up left fork

- Result: Request edges
- Everyone succeeds!
- Result: Assignment edges

Everyone tries to pick up right fork

- Result: Request edges



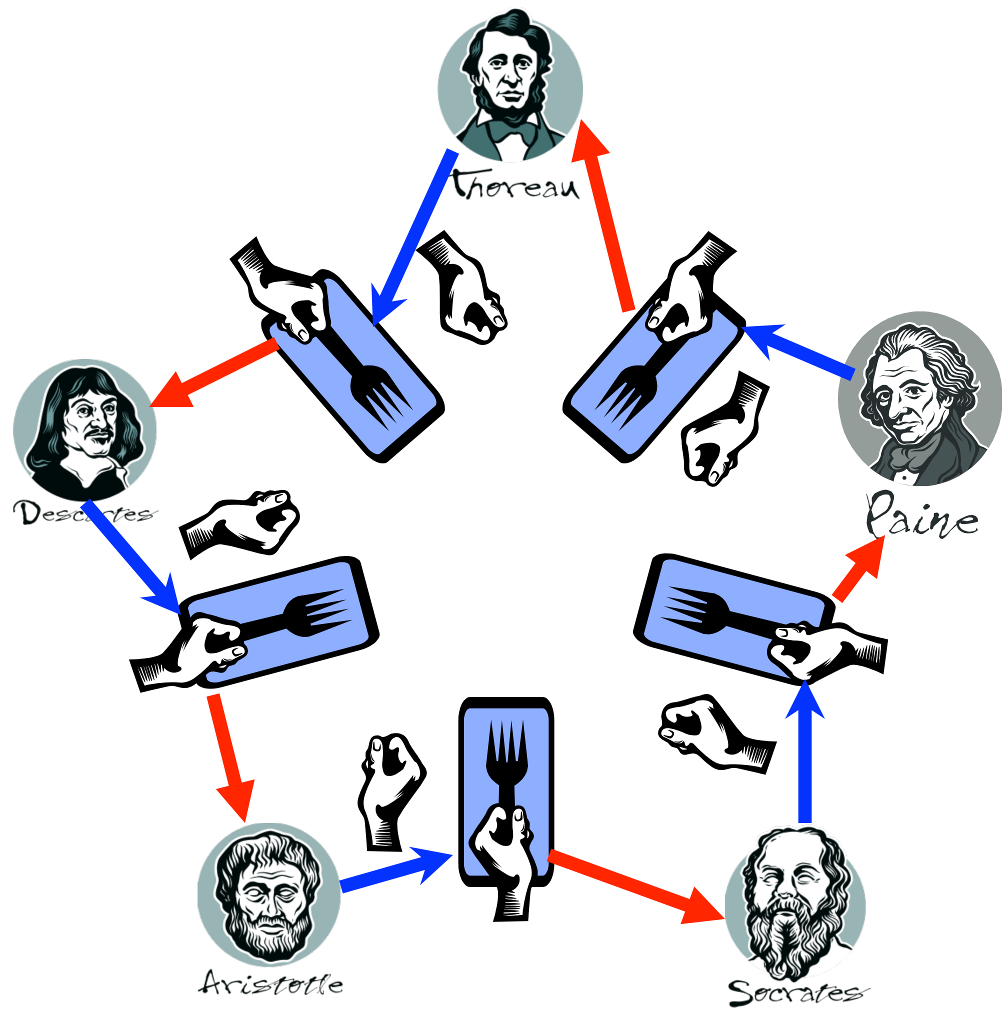
Dining Philosophers resource allocation graph

Everyone tries to pick up left fork

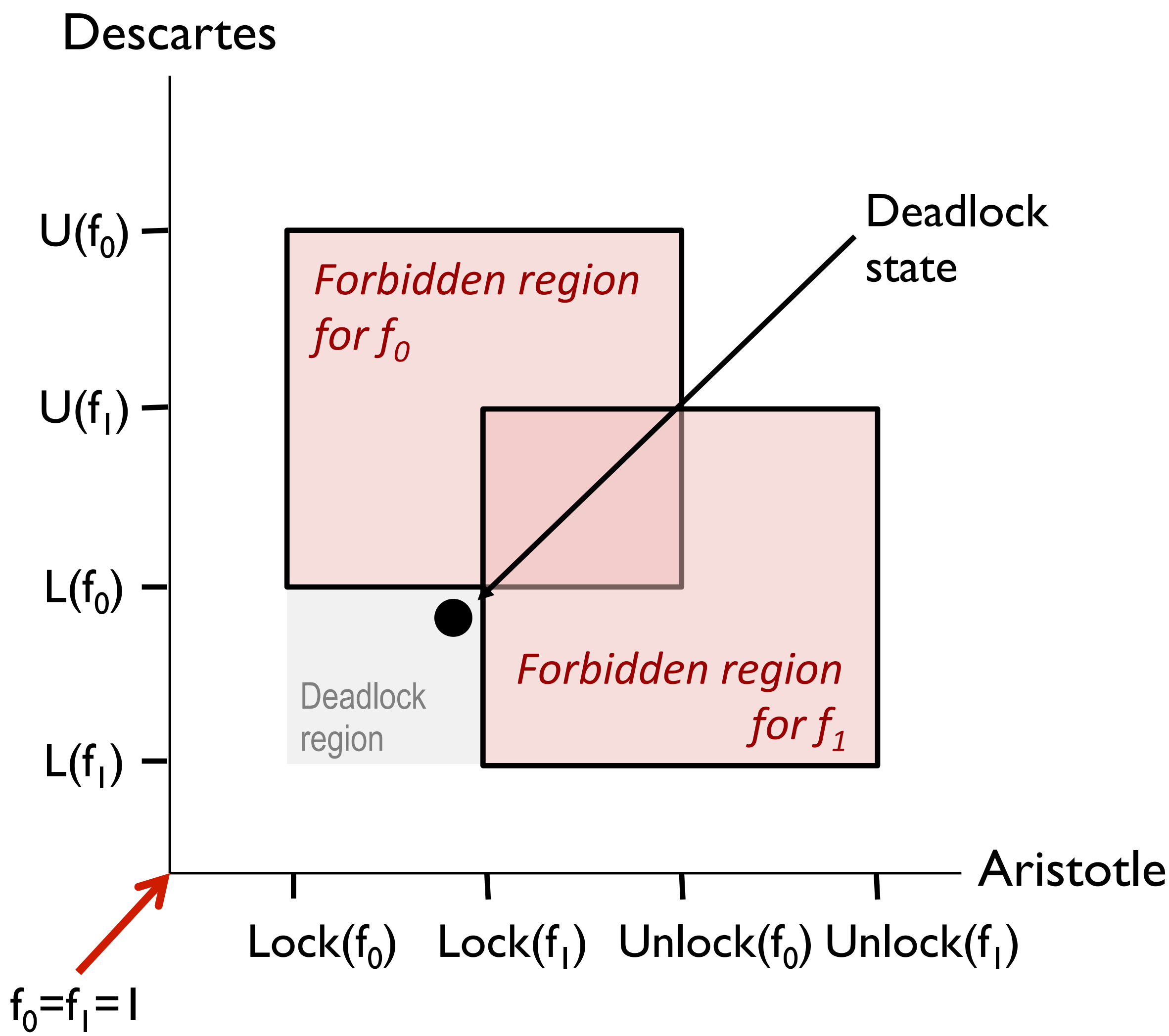
- Result: Request edges
- Everyone succeeds!
- Result: Assignment edges

Everyone tries to pick up right fork

- Result: Request edges
- DEADLOCK

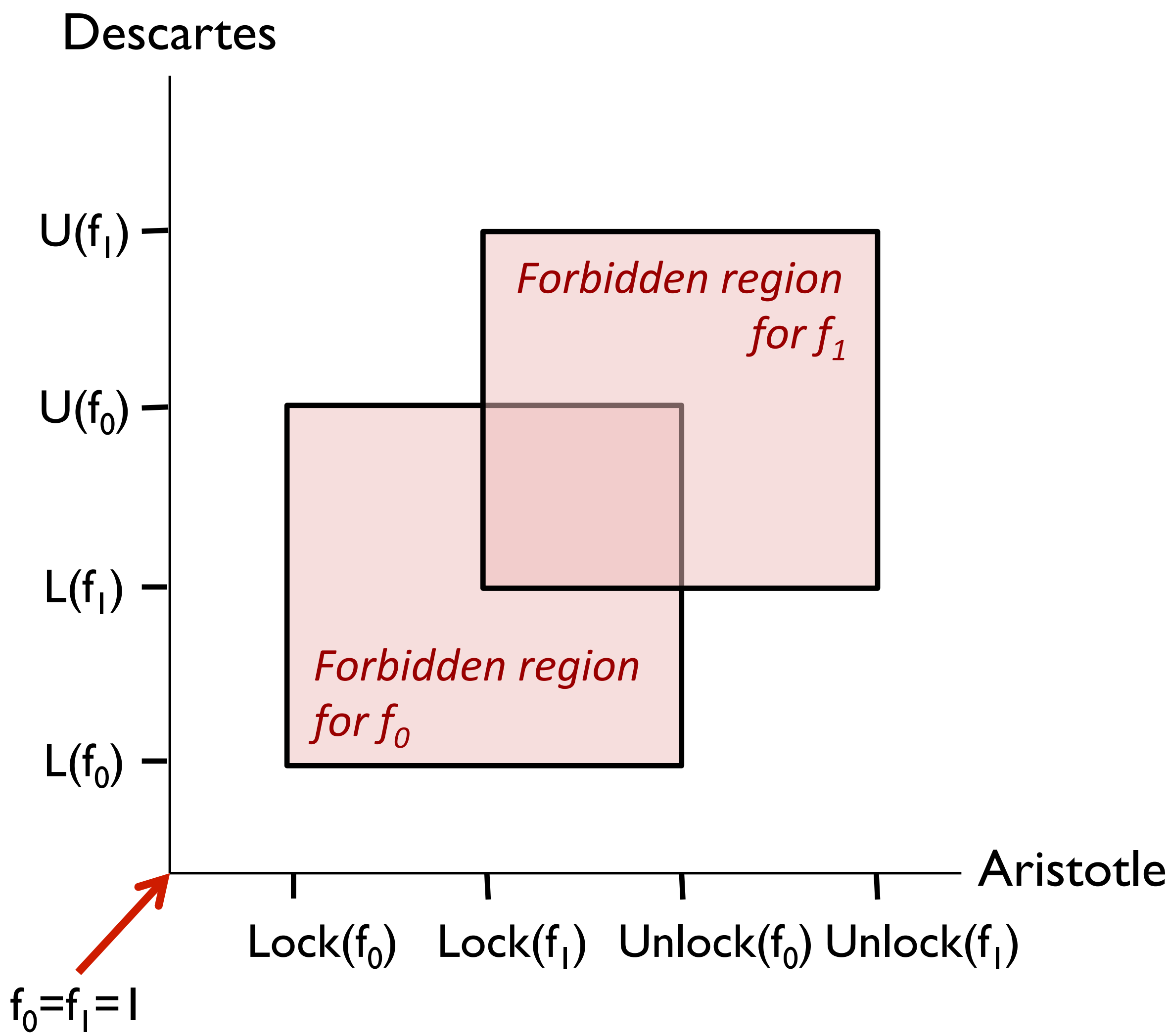


Idea: change the order



Old order

Idea: change the order



New order

Deadlock impossible!

Summary

Deadlock

- Cycle of processes / threads, each waiting on the next
- Modeled by cycle in resource allocation graph
- Often nondeterministic, tricky to debug

Next: dealing with deadlocks

- "change the order" was a nice trick... but why did it work?
- Is there a simple technique that will work always?
- Are there other ways of avoiding deadlocks?