

L18 Deadlock

(based on Ch. 32)

Common Non-Deadlock Bugs


We have seen race condition bug (e.g., counter++ in two threads)

Atomicity violation and **order-violation** are closely related to race condition

Thread 1 assumes `thd->proc_info` will not change between lines 2 and 3
- it assumes its operations are atomic

Atomicity violation

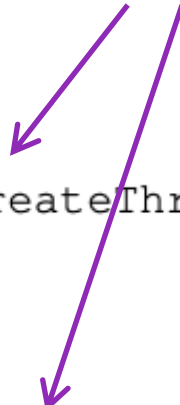
```
1 Thread 1::  
2 if (thd->proc_info) {  
3     fputs(thd->proc_info, ...);  
4 }  
5  
6 Thread 2::  
7 thd->proc_info = NULL;
```



The programmer expected thread 1 to execute before thread 2, there is no guarantee that will be the case

Order-violation

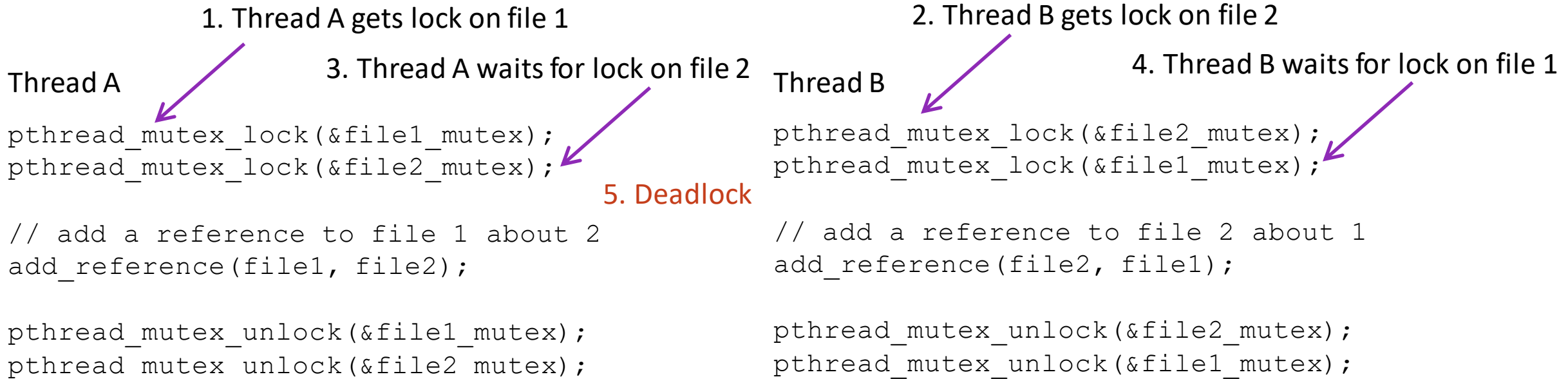
```
1 Thread 1::  
2 void init() {  
3     mThread = PR_CreateThread(mMain, ...);  
4 }  
5  
6 Thread 2::  
7 void mMain(...) {  
8     mState = mThread->State;  
9 }
```



Deadlock Example

Suppose threads A and B both require exclusive access to two files, each protected by a mutex lock

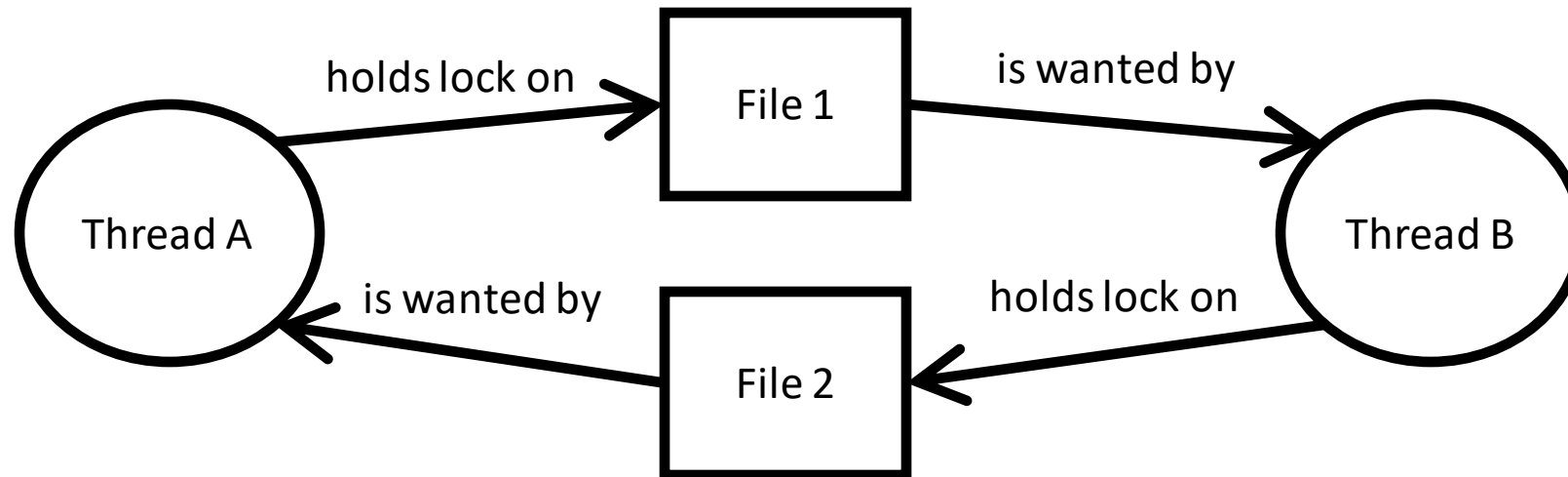
If thread A locks file1 and thread B locks file2, neither can proceed, they are deadlocked



Why deadlock happens? – Circular Wait

Why was there deadlock in the previous example?

First observation, there is a **circular wait** for resources



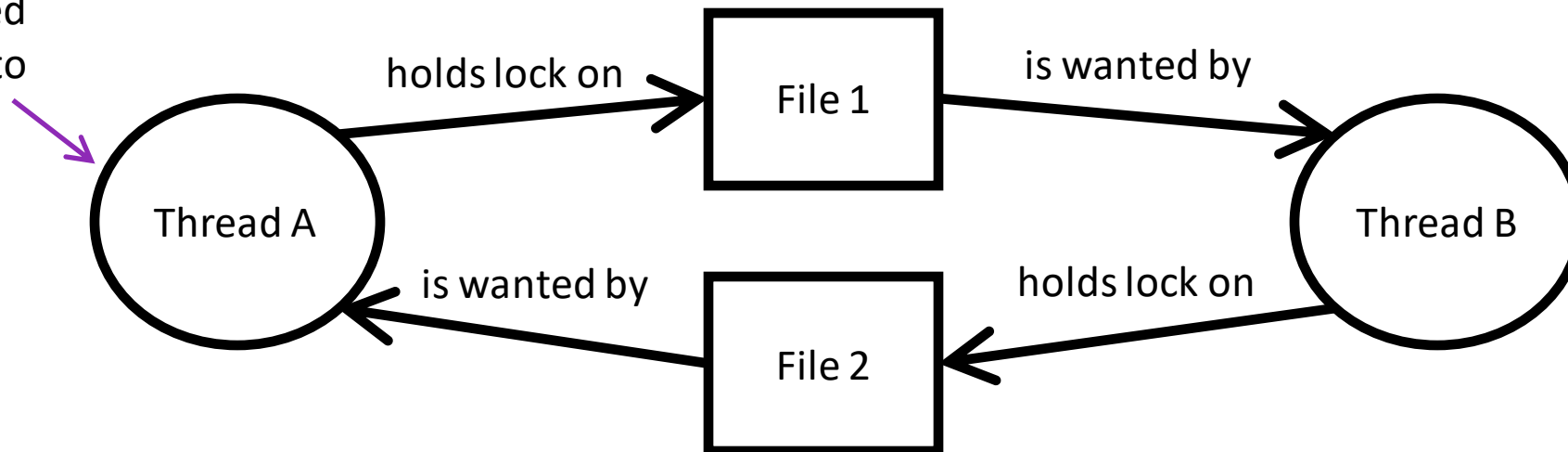
Why deadlock happens? – Mutual Exclusion

In the code both threads require **mutual exclusion** over the resources they need

If it were possible for Thread A and B to be using file 1 at the same time, there would be no deadlock

Remember that mutual exclusion is required to prevent race condition bugs

Both threads need
exclusive access to
both files



Why deadlock happens? – Hold and Wait

Deadlock is possible because both threads in the example hold some resource and then wait for another resource

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);
```

```
add_reference(file1, file2);
```

```
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Holding a resource

Now waiting for another one, if there is
deadlock this is where it will happen

Why deadlock happens? – No Preemption

Recall concept of preemption when one thread takes the CPU (a resource) from another

We can generalize preemption to any type of resource

If the OS could just force one of the threads to give up its resource, then there would be no deadlock possible

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);
```

```
add_reference(file1, file2);
```

```
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Holding a resource

If Thread A could steal the lock from
Thread B on file 2, the deadlock would not
be possible

The Four Conditions for Deadlock

All four conditions must hold for deadlock to be possible

- **Mutual Exclusion:** threads claim exclusive control of resources that they require (e.g., a thread grabs a lock)
- **Hold-and-wait:** threads hold resources allocated to them (*e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire)
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain

We just need to stop one of these to prevent deadlocks

Preventing Circular Wait

One strategy to prevent circular wait is to enforce a **total ordering** of locks

For example, suppose all threads are required to only lock file 2 after file 1

Deadlock is no longer possible

Thread A

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);  
  
// add a reference to file 1 about 2  
add_reference(file1, file2);  
  
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Thread B

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);  
  
// add a reference to file 2 about 1  
add_reference(file2, file1);  
  
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Removing Hold and Wait


When possible, it is a good idea not to hold multiple locks at the same time (i.e., don't hold a lock and wait for another)

Sometimes it is more efficient to have multiple resources locked at the same time

The example below shows avoiding hold and wait by adding an extra buffer


```
pthread_mutex_lock(&file2_mutex);  
read(file2, buffer);  
pthread_mutex_unlock(&file2_mutex);
```

Perfectly fine to hold a lock as long as the thread doesn't wait for another one while still holding the lock



```
pthread_mutex_lock(&file1_mutex);  
add_reference(file1, buffer);  
pthread_mutex_unlock(&file1_mutex);
```

Not holding the lock of file2, so fine to wait for lock on file 1



Adding Preemption

What if a thread can have its resource preempted? Then the deadlock would be broken

If not careful, preemption can lead to race condition, one strategy is restart the thread or process (or in extreme case reboot the machine)

Another alternative is a trylock – try to get the lock but don't block (wait) if it is not available

```
1  top:
2      pthread_mutex_lock(L1);
3      if (pthread_mutex_trylock(L2) != 0) {
4          pthread_mutex_unlock(L1);
5          goto top;
6      }
```

Removing Mutual Exclusion

Another solution would be to remove the need for mutual exclusion

Atomic instruction: compare-and-swap

```
int CompareAndSwap (int *address, int expected, int new){  
    if (*address == expected) {  
        *address = new;  
        Return 1; //success  
    }  
    return 0;  
}
```

Example 1: concurrent threads atomically increment a value by an amount (remove the need of lock)

```
void AtomicIncrement (int *value, int amount) {  
    do {  
        int old = *value;  
    }while(CompareAndSwap(value, old, old+amount)==0);  
}
```

Removing Mutual Exclusion

Example 2: concurrent threads atomically insert to a list

Solution with lock

```
int insert (int address){
    node_t *n = malloc(sizeof(node_t));
    n->value = value;
    pthread_mutex_lock(listlock);
    n->next = head;
    head = n;
    pthread_mutex_unlock(listlock);
}
```

Solution without lock

```
int insert (int address){
    node_t *n = malloc(sizeof(node_t));
    n->value = value;
    do{
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n)==0);
}
```

Deadlock Avoidance and Detect and Recover

Preventing deadlocks is a difficult problem, there is no really good general solution

Alternatives to preventing deadlocks

Avoid deadlocks by modifying the scheduler, a clever scheduler might know that two threads should never be scheduled concurrently because it can result in hold and wait

Detect and recover from a dead lock, some deadlocks are so difficult to prevent or avoid that OSes use the fallback strategy of detect the deadlock and reboot the system when it happens

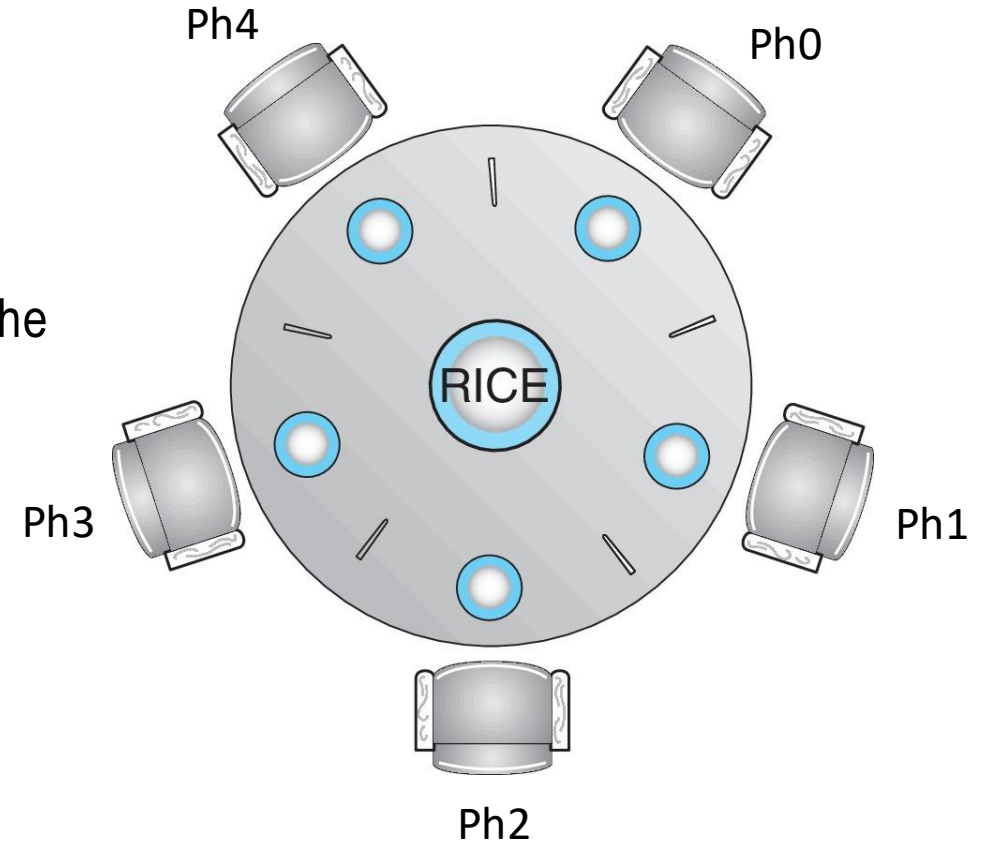
Classic Problem: Dinning Philosophers

5 philosophers alternate between thinking and eating

There are 5 chopsticks placed between the philosophers

To eat, a philosopher must pick up two chopsticks (the ones to the right and left of them), to think they put down both chopsticks

```
void * thread_start(void *arg) {  
    // ph is the philosopher's id (0 to 4)  
    int *ph = (int *)arg;  
    while(1) {  
        think();  
        get_chopsticks(ph);  
        eat();  
        put_chopsticks(ph);  
    }  
}
```



A Broken Solution

How to implement get() and put() functions using semaphores?

Each chopstick is a type of resources, create a semaphore for each chopstick initialized to 1

A philosopher must sem_wait (decrement) a chopstick semaphore to pick it up and sem_post (increment) a chopstick semaphore to put it back

```
sem_t chopsticks[5];

void init_chopsticks() {
    for (int i=0; i<5; i++) {
        sem_init(&chopsticks[i], pshared, 1);
    }
}

void get_chopsticks(int ph) {
    sem_wait(&chopsticks[left(ph)]);
    sem_wait(&chopsticks[right(ph)]);
}

void put_chopsticks(int ph) {
    sem_post(&chopsticks[left(ph)]);
    sem_post(&chopsticks[right(ph)]);
}
```


The Problem - Deadlock

What is wrong with the solution? Consider the following execution:

- Ph0 takes left chopstick
- Context switch to Ph1
- Ph1 takes left chopstick
- Context switch to Ph2
- Ph2 takes left chopstick
- Context switch to Ph3
- Ph3 takes left chopstick
- Context switch to Ph4
- Ph4 takes left chopstick

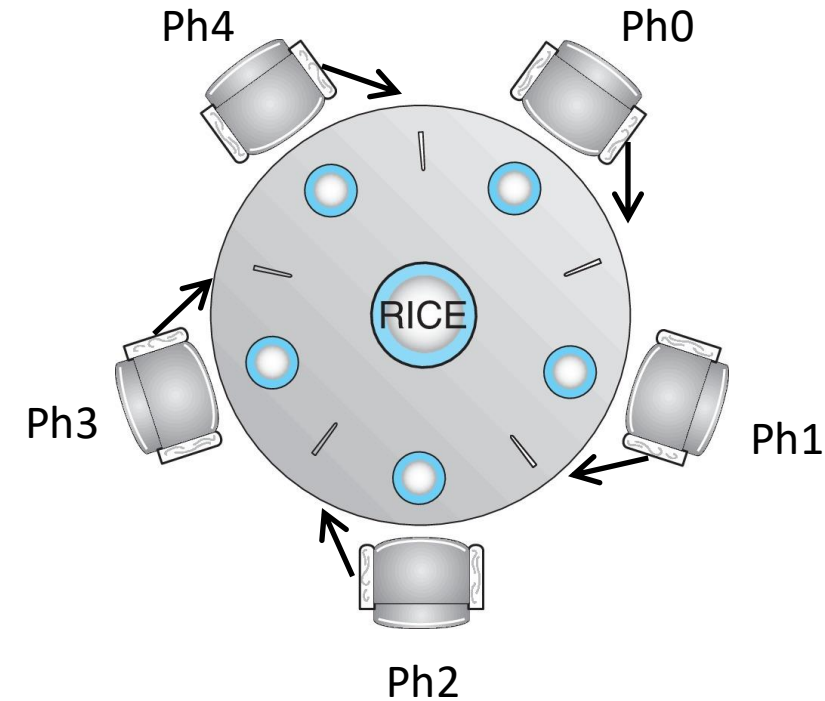
Now what? All threads are waiting for right chopstick, the system is stuck, forever!

When a group of threads are each waiting on another forming a cycle a **deadlock** has formed

```
// assume chopsticks[] is an array of 5 semaphores initialized to 1
```

```
void get_chopsticks(int ph) {  
    sem_wait(&chopsticks[left(ph)]);  
    sem_wait(&chopsticks[right(ph)]);  
}
```

```
void put_chopsticks(int ph) {  
    sem_post(&chopsticks[left(ph)]);  
    sem_post(&chopsticks[right(ph)]);  
}
```



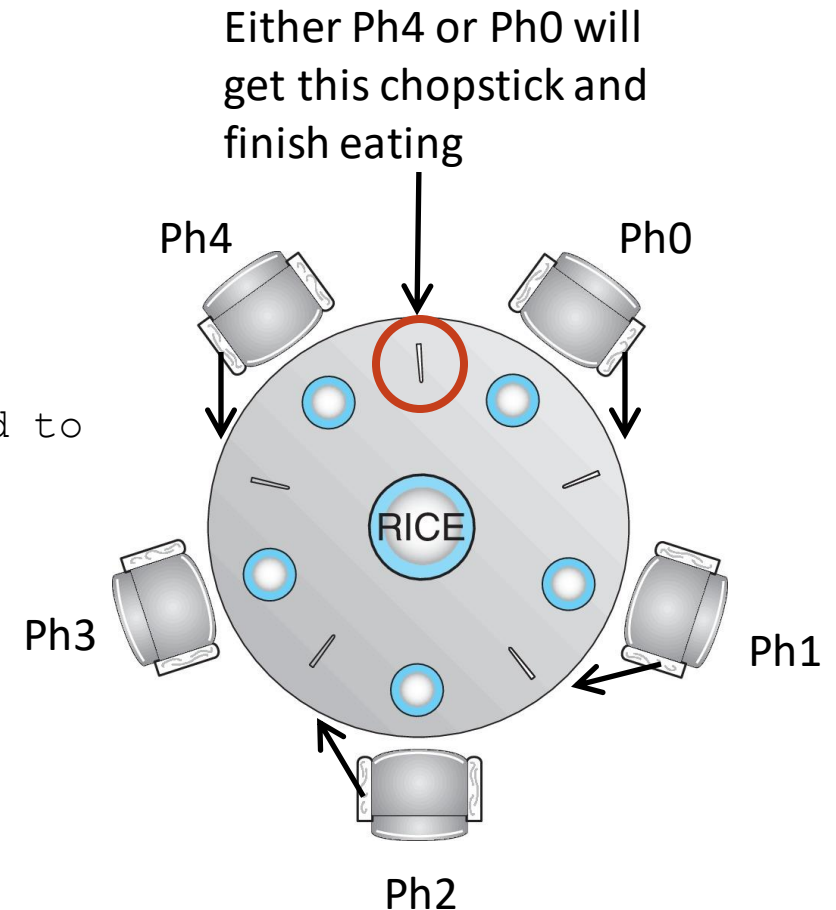
One Solution

Consider a simple rule: Ph4 must pick up the right chopstick before the left

No longer possible for cycle to form, therefore deadlock is not possible

```
// assume chopsticks[] is an array of 5 semaphores initialized to
```

```
void get_chopsticks(int ph) {  
    if (ph == 4) {  
        sem_wait(&chopsticks[right(ph)]);  
        sem_wait(&chopsticks[left(ph)]);  
    } else {  
        sem_wait(&chopsticks[left(ph)]);  
        sem_wait(&chopsticks[right(ph)]);  
    }  
}
```



Slightly Different Problem

Consider a second chopstick placed between Ph4 and Ph0

Is deadlock possible?

```
sem_t chopsticks[5];  
  
void init_chopsticks() {  
    sem_init(&chopsticks[0], pshared, 2);  
    for (int i=1; i<5; i++) {  
        sem_init(&chopsticks[i], pshared, 1);  
    }  
}
```

