

# CS240 Operating Systems, Communications and Concurrency

## Classical Coordination Problems

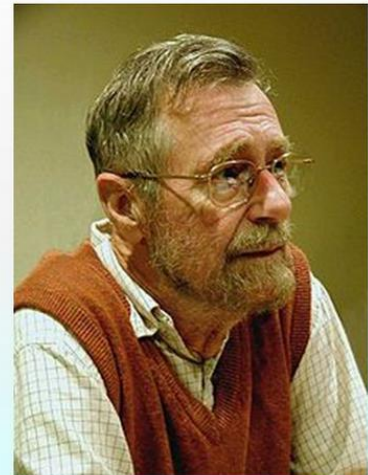
This synchronisation problem has to do with the allocation of limited resources among processes in a way that avoids deadlock.

## Dining Philosophers Problem

Originally formulated in 1965 by Dutch computer scientist Edsger Dijkstra as a student exercise and later as this analogy by English computer scientist Tony Hoare (associated with Quicksort, CSP, Occam language).

### Edsger Wybe Dijkstra

- Dijkstra was born in Rotterdam, Netherlands.
- He received the [1972 Turing Award](#) for fundamental contributions to developing programming languages.
- In 2002, he received the [ACM PODC Influential Paper Award](#) in distributed computing for his work on [self-stabilization](#) of program computation. This annual award was renamed the [Dijkstra Prize](#) the following year, in his honor.



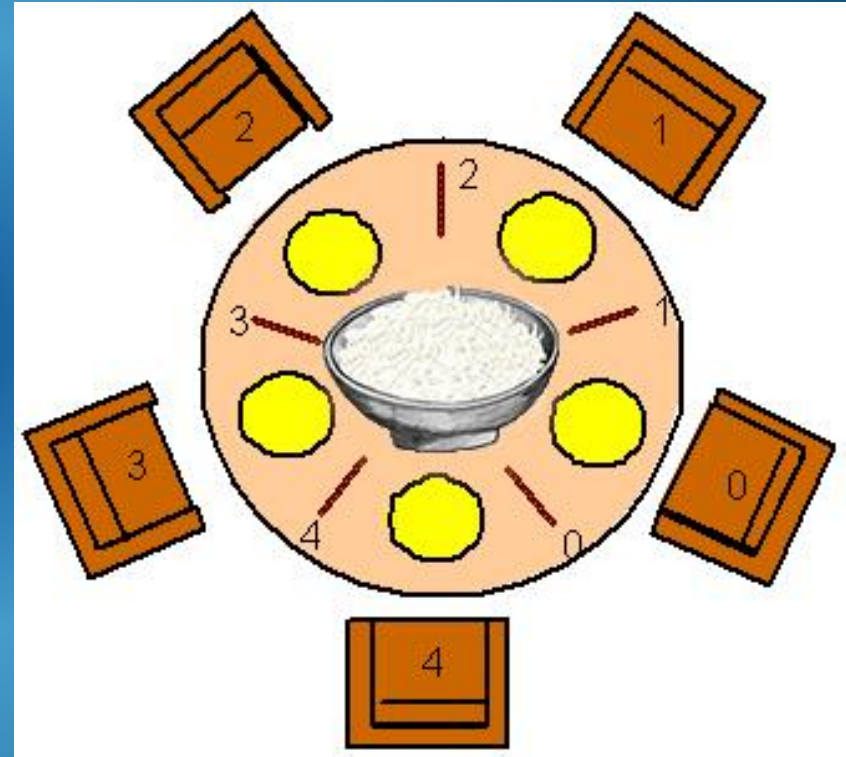
(1930~2002)

# CS240 Operating Systems, Communications and Concurrency

## Dining Philosophers Problem

Let's say that we have a table on which there are five plates and five chopsticks with a bowl of rice in the centre. Sitting around the table are five philosophers whose behaviour is described as follows:-

```
Philosopher
do {
    think();
    eat();
} while (true);
```



To eat, a philosopher requires his left and right chopstick.

# CS240 Operating Systems, Communications and Concurrency

## Recall Semaphore Class

```
class Semaphore {  
    private int value;  
  
    // Constructor  
    public Semaphore(int value) {  
        this.value = value;  
    }  
    public synchronized void acquire()  
    public synchronized void release()  
}
```



# CS240 Operating Systems, Communications and Concurrency

## Initial implementation of philosopher synchronisation

Each chopstick is represented by a binary semaphore, initially free. (We use the Semaphore class we defined previously)

Each philosopher  $i$  could then have the following behaviour:-

```
while (alive) do {  
    chopstick[i].acquire();    {wait for left chopstick }  
    chopstick[(i+1) % 5].acquire();    {wait for right}  
    eat();  
    chopstick[i].release();  
    chopstick[(i+1) % 5].release();  
    think();  
}
```

# CS240 Operating Systems, Communications and Concurrency

## Implementation of Dining Philosophers Simulation

```
class DiningPhilosophers {  
    public static void main(String args[]) {  
        Semaphore chopSticks[];  
  
        // Create an array of five Semaphore Object Reference Handles  
        chopSticks = new Semaphore[5];  
  
        // Create five Binary Semaphore Objects and assign to the array  
        for (int i=0; i<5; i++)  
            chopSticks[i] = new Semaphore(1);  
    }  
}
```

# CS240 Operating Systems, Communications and Concurrency

## Initial implementation of Philosopher Behaviour

```
class Philosopher extends Thread {  
    private int myName;  
    private Semaphore chopSticks[];  
  
    // This is the constructor function which is executed when a  
    // Philosopher thread is first created. It takes in handles for  
    // the shared objects and stores a reference to those in local  
    // variables.  
  
    public Philosopher(int myName, Semaphore chopSticks[]) {  
        this.myName = myName;  
        this.chopSticks = chopSticks;  
    }  
}
```

```
class Philosopher extends Thread {  
    public void run() {  
        while (true) {  
            System.out.println("Philosopher "+myName+" thinking.");  
            try { // Simulate thinking with a random sleep period  
                sleep ((int) (Math.random()*20000));  
            } catch (InterruptedException e) {}  
  
            System.out.println("Philosopher "+myName+" hungry.");  
            chopSticks[myName].acquire(); // Acquire left  
            chopSticks[(myName+1)%5].acquire(); //Acquire right  
            System.out.println("Philosopher "+myName+" eating.");  
            try { // Simulate eating activity for a random time  
                sleep ((int) (Math.random()*10000));  
            } catch (InterruptedException e) {}  
            chopSticks[myName].release(); // Release left  
            chopSticks[(myName+1)%5].release(); // Release right  
        }  
    }  
}
```



# CS240 Operating Systems, Communications and Concurrency

## Implementation of Dining Philosopher Simulation

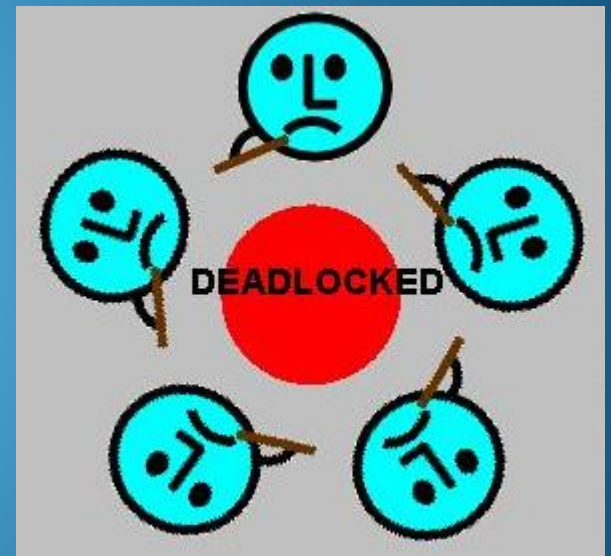
```
class DiningPhilosophers {  
    public static void main(String args[]) {  
        Philosopher workerThread[];  
  
        // Create an array of five Philosopher Object Reference Handles  
        workerThread = new Philosopher[5];  
  
        // Create and initiate five Philosopher Thread Objects  
        for (int i=0; i<5; i++) {  
            workerThread[i] = new Philosopher(i, chopSticks);  
            workerThread[i].start();  
        }  
    }  
}
```



# CS240 Operating Systems, Communications and Concurrency

This solution properly synchronises access to the chopsticks. However, what happens if every philosopher picks up one chopstick at the same time?

We have all processes waiting on an event (the freeing of some chopstick) but that event can only be generated by one of the waiting processes (i.e. never). This is called a **deadlock**.



# CS240 Operating Systems, Communications and Concurrency

In this example, we could eliminate the potential deadlock problem in a number of ways.

## **Method 1 - Deadlock Avoidance by Restricting Concurrency**

Allow at most four philosophers to sit at the table at the same time. This means at least one philosopher can always eat and when she's finished, another can eat and so on. There is progress.

```
class Philosopher extends Thread {  
    public void run() {  
        while (true) {
```

```
// A shared counting semaphore "room" with maximum value of 4  
// could be defined to prevent more than 4 philosophers at a time  
// from acquiring chopsticks, always leaving enough for progress  
        room.acquire();  
  
        chopSticks[myName].acquire(); // Acquire left  
        chopSticks[(myName+1)%5].acquire(); //Acquire right  
        eat();  
        chopSticks[myName].release(); // Release left  
        chopSticks[(myName+1)%5].release(); // Release right  
  
        room.release();  
    }  
}
```

# CS240 Operating Systems, Communications and Concurrency

## Method 1 - Note

This scheme reduces the potential concurrency in order to avoid a potential deadlock situation occurring and therefore system throughput is reduced on the **chance** that a deadlock may occur.



# CS240 Operating Systems, Communications and Concurrency

Eliminating deadlock from the solution:-


## Method 2 – Deadlock Prevention of Hold and Wait

Acquire both chopsticks together. We could implement the chopsticks as an integer array with a mutex semaphore to guard modifications to it.

A philosopher would acquire the mutex semaphore first and then check the array for the required two chopsticks, if they are both available, then the array is modified to reflect both being taken. If not, it should not take any chopsticks, but release mutex and wait again.

```
class Philosopher extends Thread {  
private int[] chopSticks;  
private Semaphore mutex;
```

These are initialised  
with values passed  
into the constructor



```
    public void run() {  
        while (true) {
```

```
        .....
```

```
        haveChopSticks = false;
```

```
        while (!haveChopSticks) {
```

```
            mutex.acquire();
```

```
            if ((chopSticks[myName]==1) && (chopSticks[(myName+1)%5]==1)) {
```

```
                chopSticks[myName]=0;
```

```
                chopSticks[(myName+1)%5]=0;
```

```
                haveChopSticks = true;
```

```
            }
```

```
            mutex.release();
```

```
        }
```

Semaphore mutex  
is shared by all  
Philosopher  
Threads



# CS240 Operating Systems, Communications and Concurrency

## Method 2 - Note

In reality, it would be difficult for a process, especially one with large resource requirements, to **acquire all of its resources at the same time** before it could do any work.

Also there is **busy waiting** in our solution where the while loop continually attempts to acquire the mutex and test the chopstick array until the thread has its chopsticks.

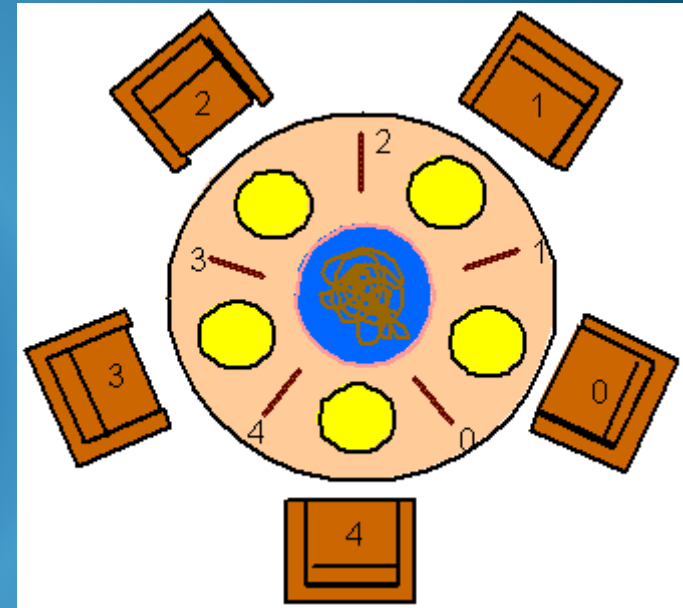
# CS240 Operating Systems, Communications and Concurrency

## Method 3 – Prevention of Hold and Wait

An ad hoc **asymmetrical solution** works quite well in this case to prevent some philosophers from holding resources while waiting for more.

If an Odd numbered philosopher picks up his left chopstick first and then his right, but an even philosopher picks up his right first then his left, then a deadlock situation is prevented. It stops all five from picking up simultaneously.

A philosopher cannot hold any chopstick initially if his neighbour already holds one.





```
class Philosopher extends Thread {  
    public void run() {  
        while (true) {  
  
.....  
        // determines if id is odd or even  
        if ((myName % 2) == 0) {  
            chopSticks[myName].acquire(); // Acquire left  
            chopSticks[(myName+1)%5].acquire(); // Acquire right  
        }  
        else {  
            chopSticks[(myName+1)%5].acquire(); // Acquire right  
            chopSticks[myName].acquire(); // Acquire left  
        }  
  
.....  
    }  
}
```

# CS240 Operating Systems, Communications and Concurrency

## Method 3 - Note

This is an ad hoc solution and can be seen to work for this concurrency scenario but is difficult to apply generally. A similar such solution may not be obvious in a typical resource scheduling situation.

# CS240 Operating Systems, Communications and Concurrency

## Final Thoughts on Dining Philosophers Problem

Our original solution shared the chopstick resources properly but was susceptible to **deadlock**.

Alternative methods were proposed to **avoid** or **prevent deadlock**, but came at the cost of restricting resource usage performance or discriminating against larger processes.

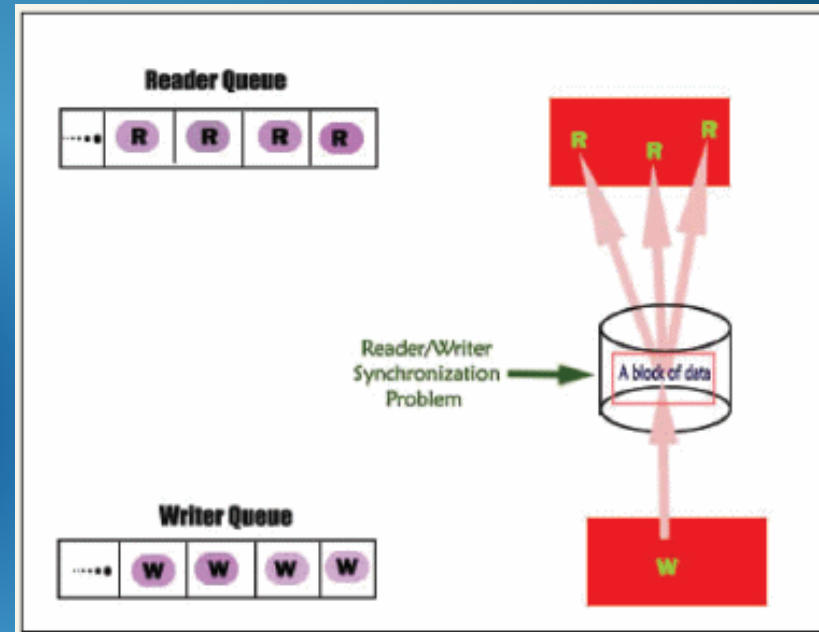
None of the solutions make any guarantees about **starvation** which may occur due to unfavourable scheduling or conspiracy.

An even better solution might attempt to deal with that and giving a **fair allocation** of chopstick time to each philosopher.

# CS240 Operating Systems, Communications and Concurrency

## Readers/Writers Problem

A number of concurrent threads wish to carry out operations on a shared data. Some operations may involve reading the data and others involve writing/updating it.



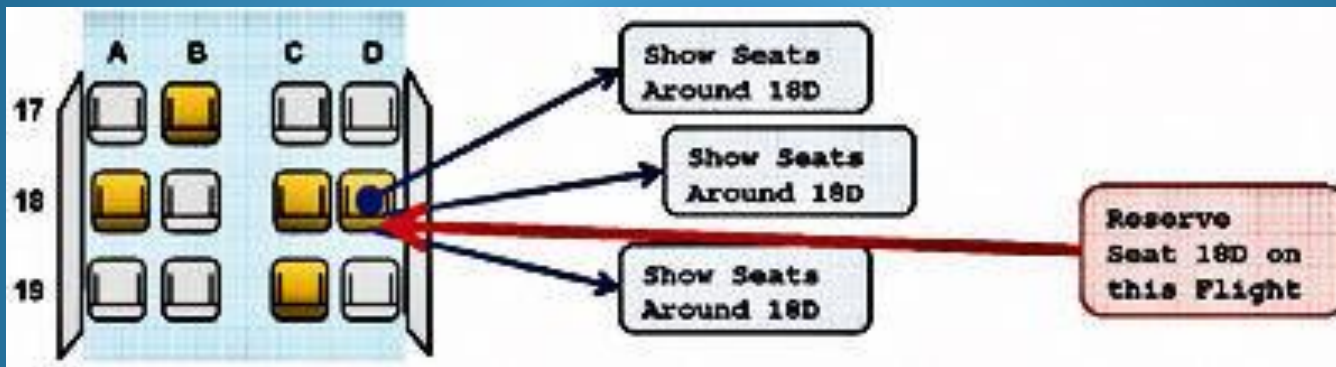
While reader threads can access the data concurrently without any adverse affects, a writer thread requires exclusive access to carry out its transaction.

The readers/writers problem has a number of alternative solutions, which try to address efficiency and fairness to reader and/or writer threads in different ways.



# CS240 Operating Systems, Communications and Concurrency

## Two Solutions to the Readers/Writers Problem



The **first readers/writers problem** prioritises readers and requires that no reader be kept waiting unless a writer has already obtained permission to use the shared item.

The **second readers/writers problem** prioritises writers and requires that once a writer is ready, that writer performs its write as soon as possible.

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Solution Design Strategy (Prioritising Readers)

Readers and writers are **mutually exclusive**. We will use a semaphore (**wrt**) to control this, i.e. whether readers or a writer can access the data item.

A writer must acquire the **wrt** semaphore before writing, and the **first reader** must acquire the **wrt** semaphore to prevent writers. Other readers can proceed immediately according to the policy, once at least one reader is accessing the shared item.

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Solution Design Strategy

If a reader is finished with the data, and there are no other readers using the item, then the **last reader can give up the semaphore** (`wrt`) to allow any waiting writers to enter.

In order to determine if a reader is the first or last reader, we will have to keep a count (`readerCount`) of the number of readers accessing the item at one time. This variable will be incremented as new readers access the data and decremented as new readers finish with the data.

We will use a semaphore `mutex` to ensure that only one reader thread at a time tests and modifies `readerCount`.

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Solution Implementation

### Java Implementation Approach

We will implement a `DataAccessPolicyManager` class to manage the data access with different access methods for readers and writers. An instance of this class can be used to protect a given data set using the policy approach above to the first readers/writers problem.

To use the item a reader will `acquireReadLock()` and when finished will `releaseReadLock()`.

To use the item a writer will `acquireWriteLock()` and when finished will `releaseWriteLock()`.



# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Solution Implementation

```
public class DataAccessPolicyManager
{
    private int readerCount;
    private Semaphore mutex;
    private Semaphore wrt;

    public DataAccessPolicyManager () {
        readerCount = 0;
        mutex = new Semaphore(1);
        wrt = new Semaphore(1);
    }
```

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Solution Implementation

```
public class DataAccessPolicyManager
{
    public void acquireReadLock() {
        mutex.acquire();
        ++readerCount;
        if (readerCount == 1) // first reader
            wrt.acquire();
        mutex.release();
    }
}
```

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Solution Implementation

```
public class DataAccessPolicyManager
{
    public void releaseReadLock() {
        mutex.acquire();
        --readerCount;
        if (readerCount == 0) // Last reader
            wrt.release();
        mutex.release();
    }
}
```

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Solution Implementation

```
public class DataAccessPolicyManager  
{
```

```
    public void acquireWriteLock() {  
        wrt.acquire();  
    }
```

```
    public void releaseWriteLock() {  
        wrt.release();  
    }
```



# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Problem Simulation

### Defining a Reader Thread Class

```
public class Reader extends Thread {
```

```
// All threads which use the data being synchronised should  
// share the same DataAccessPolicyManager object to  
// coordinate access. The instance could be passed in to the  
// constructor for the Reader class.
```

```
    public Reader (DataAccessPolicyManager accessManager)  
    {  
        }  
  
}
```

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Problem Simulation

### Defining a Reader Thread Class

#### Reader Thread Behaviour:-

```
public void run() {  
  
    accessManager.acquireReadLock();  
  
    Do reading activity  
  
    accessManager.releaseReadLock();  
}
```

# CS240 Operating Systems, Communications and Concurrency

## First Readers/Writers Problem Simulation

### Defining a Writer Thread Class

#### Writer Thread Behaviour:-

```
public void run() {  
  
    accessManager.acquireWriteLock();  
  
    Do writingactivity  
  
    accessManager.releaseWriteLock();  
}
```

# CS240 Operating Systems, Communications and Concurrency

```
public class ReadersWritersSimulation {  
    public static void main (String args[]) {  
        DataAccessPolicyManager accessManager =  
            new DataAccessPolicyManager();  
  
        Reader reader1 = new Reader(accessManager);  
        Reader reader2 = new Reader(accessManager);  
        Reader reader3 = new Reader(accessManager);  
        Writer writer = new Writer(accessManager);  
  
        reader1.start();  
        reader2.start();  
        reader3.start();  
        writer.start();  
    }  
}
```

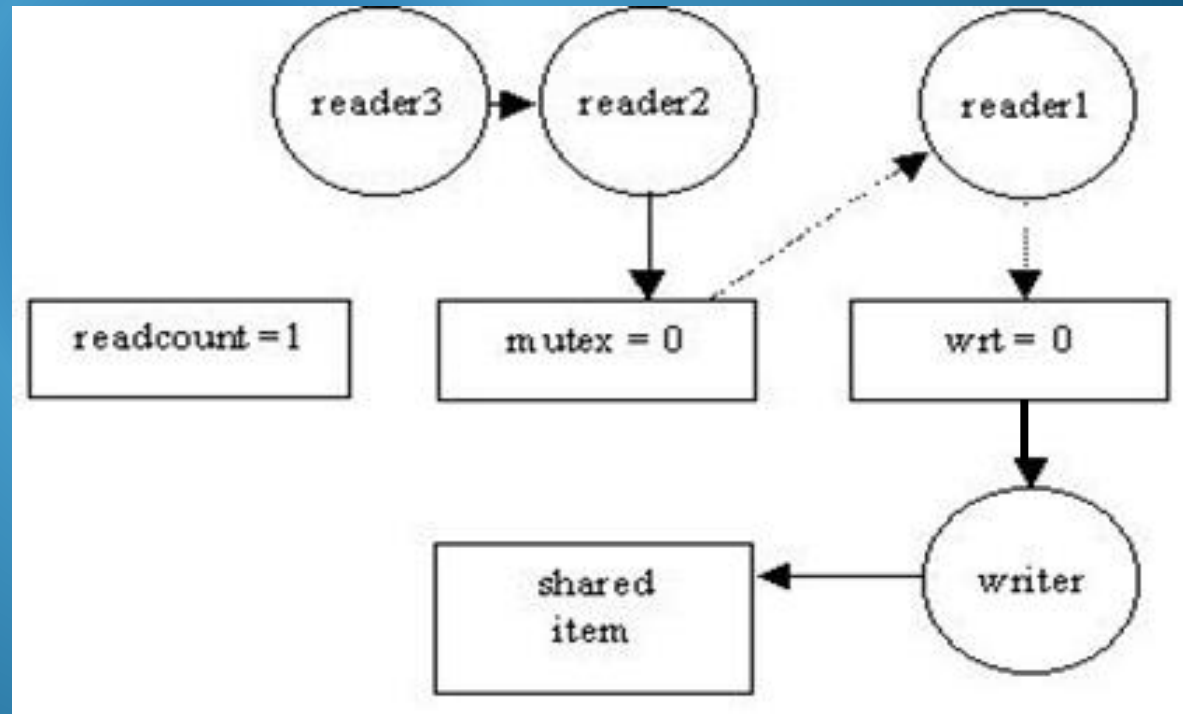


# CS240 Operating Systems, Communications and Concurrency

As an example, to demonstrate a snapshot of this system, let's say that the shared data is in use by a writer and there are three readers waiting for it to finish. The state of the structures would be as follows:-

```
public void acquireWriteLock()
{
    wrt.acquire();
}

public void acquireReadLock()
{
    mutex.acquire();
    ++readerCount;
    if (readerCount == 1)
        wrt.acquire();
    mutex.release();
}
```

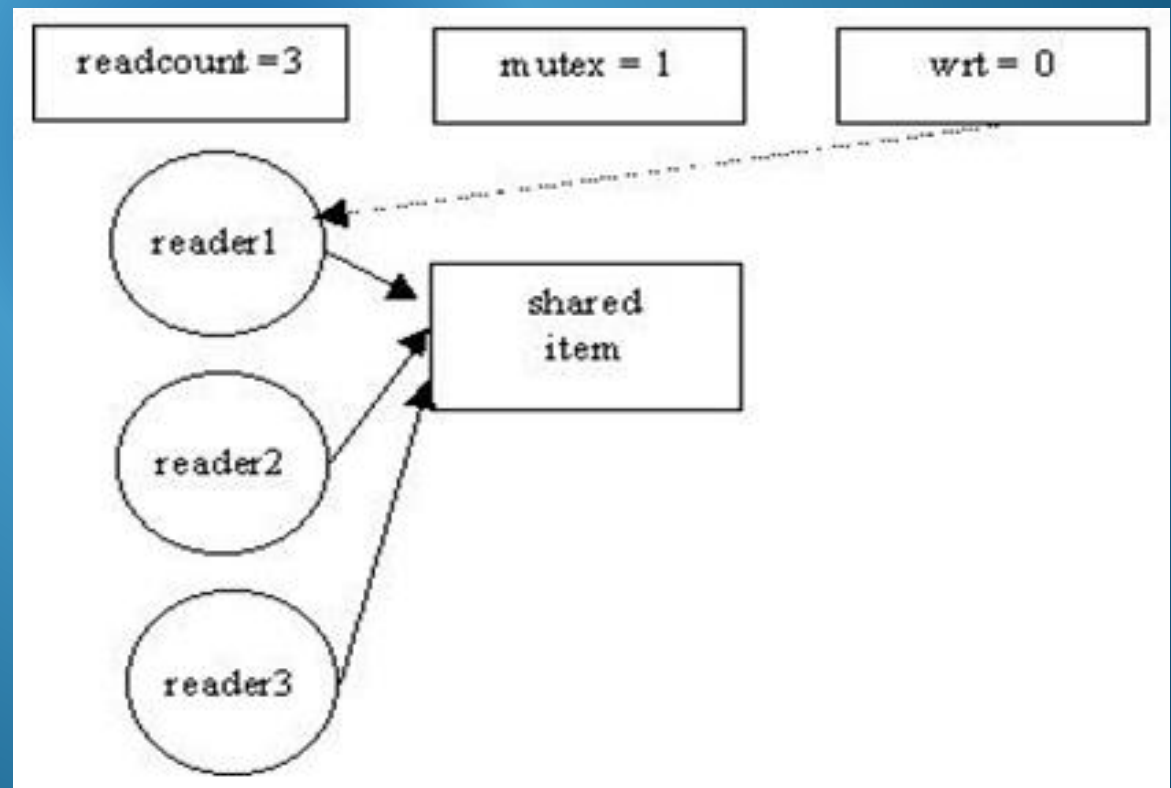


# CS240 Operating Systems, Communications and Concurrency

When the writer exits and releases wrt, the representation of the structures would be as follows:-

```
public void acquireWriteLock()  
{  
    wrt.acquire();  
}
```

```
public void acquireReadLock()  
{  
    mutex.acquire();  
    ++readerCount;  
    if (readerCount == 1)  
        wrt.acquire();  
    mutex.release();  
}
```



# CS240 Operating Systems, Communications and Concurrency

## Second Readers Writers Solution Design Strategy

The second readers writers policy prioritises writers. Writers only have to wait for current readers to finish. New readers must wait until there are no writers. Need a way for first writer to block new readers from competing for data item and last writer to unblock readers. The implementation is an extension of the first one.

## Data structures required for solution

```
public class DataAccessPolicyManager2
{
```

```
    private int readCount, writeCount;
```

```
    private Semaphore mutexReadCount, mutexWriteCount;
```

```
    private Semaphore wrt, rdr;
```

For  
blocking  
writers



For  
blocking  
readers

# CS240 Operating Systems, Communications and Concurrency

## Second Readers Writers Solution Implementation

### Constructor for access manager class

```
public DataAccessPolicyManager2 () {  
    readCount = 0;  
    writeCount = 0;  
    mutexReadCount = new Semaphore(1);  
    mutexWriteCount = new Semaphore(1);  
    wrt = new Semaphore(1); // to block writers  
    rdr = new Semaphore (1); // to block readers  
}
```



# CS240 Operating Systems, Communications and Concurrency

## Second Readers Writers Solution Implementation

The second readers writers policy prioritises writers. Writers only have to wait for current readers to finish. New readers must wait until there are no writers.

```
public void acquireReadLock() {  
    rdr.acquire(); // Reader can enter if no writers  
    mutexReadCount.acquire();  
    readCount = readCount + 1;  
    if (readCount == 1) wrt.acquire(); // block writers  
    mutexReadCount.release();  
    rdr.release(); // allow another in  
}
```

# CS240 Operating Systems, Communications and Concurrency

## Second Readers Writers Solution Implementation

The second readers writers policy prioritises writers. Writers only have to wait for current readers to finish. New readers must wait until there are no writers.

```
public void releaseReadLock() {  
    mutexReadCount.acquire();  
    readCount = readCount - 1;  
    if (readCount == 0) wrt.release(); //last reader  
    mutexReadCount.release();  
}
```

# CS240 Operating Systems, Communications and Concurrency

## Second Readers Writers Solution Implementation

The second readers writers policy prioritises writers. Writers only have to wait for current readers to finish. New readers must wait until there are no writers.

```
public void acquireWriteLock() {  
    mutexWriteCount.acquire();  
    writeCount = writeCount+1;  
    if (writeCount == 1) rdr.acquire(); //Block new readers  
    mutexWriteCount.release();  
    wrt.acquire(); // wait for existing readers/writer  
}
```

# CS240 Operating Systems, Communications and Concurrency

## Second Readers Writers Solution Implementation

The second readers writers policy prioritises writers. Writers only have to wait for current readers to finish. New readers must wait until there are no writers.

```
public void releaseWriteLock() {  
    mutexWriteCount.acquire();  
    writeCount = writeCount - 1;  
    if (writeCount==0) rdr.release();//no more writers  
    mutexWriteCount.release();  
    wrt.release();  
}
```



# CS240 Operating Systems, Communications and Concurrency

One issue with this code is that writers and readers both compete for the `rdr` semaphore in the `acquireReadLock()` and `acquireWriteLock()` methods.

The writer needs to acquire the `rdr` semaphore to block new readers. If there are many readers competing for the `rdr.acquire()` method the writer may have trouble in getting it and as a result it may experience delay, against the policy.

```
public void acquireReadLock() {  
    rdr.acquire(); // Contention for this method  
    mutexReadCount.acquire();  
    readCount = readCount + 1;  
    if (readCount == 1) wrt.acquire(); // block writers  
    mutexReadCount.release();  
    rdr.release(); // allow another in  
}
```

# CS240 Operating Systems, Communications and Concurrency

We could reduce contention for the `rdr` semaphore by first requiring readers to acquire another binary semaphore (say `mutexContentionGate`) before seeking to acquire `rdr`.

```
public void acquireReadLock() {  
    mutexContentionGate.acquire();  
  
    rdr.acquire();  
    mutexReadCount.acquire();  
    readCount = readCount + 1;  
    if (readCount == 1) wrt.acquire();  
    mutexReadCount.release();  
    rdr.release();  
  
    mutexContentionGate.release();  
}
```

Now never more  
than 1 reader thread  
in the wait set

