# Semaphores

Edsger Dijkstra

P or wait

Dutch word *proberen*, to test

*probeer te verlagen*, try to decrease

**`OS_Wait`** **`OSSemPend`**

V or signal

Dutch word *verhogen*, to increase

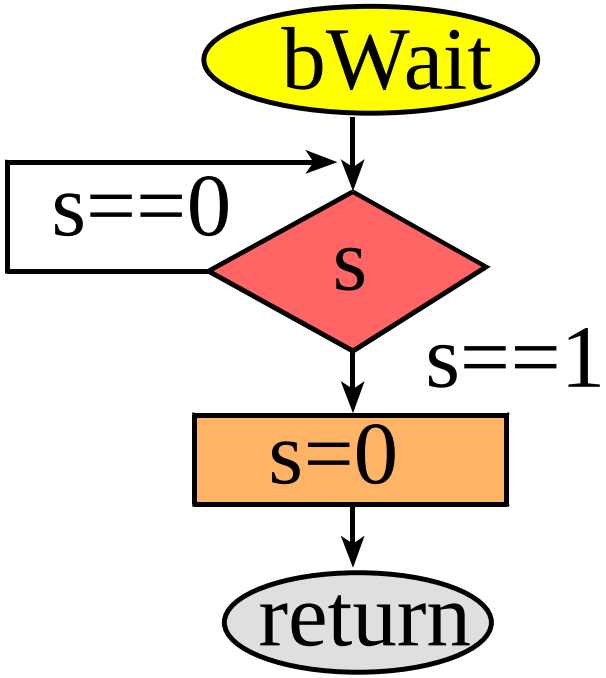
**`OS_Signal`** **`OSSemPost`**

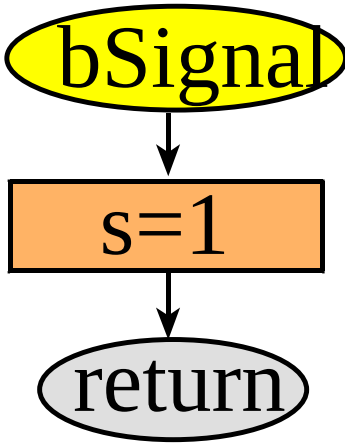
Reference Book, chapter 4

# Semaphore means something

- Counter
  - Number of elements stored in FIFO
  - Space left in the FIFO
  - Number of printers available
- Binary
  - Free (1), busy (0)
  - Event occurred (1), not occurred (0)

# Spin-lock binary




**Mutual exclusion**

```
bWait(&s);
// access LCD
bSignal(&s);
```

How do we use this to solve critical sections?

Why is this a good solution for critical sections?

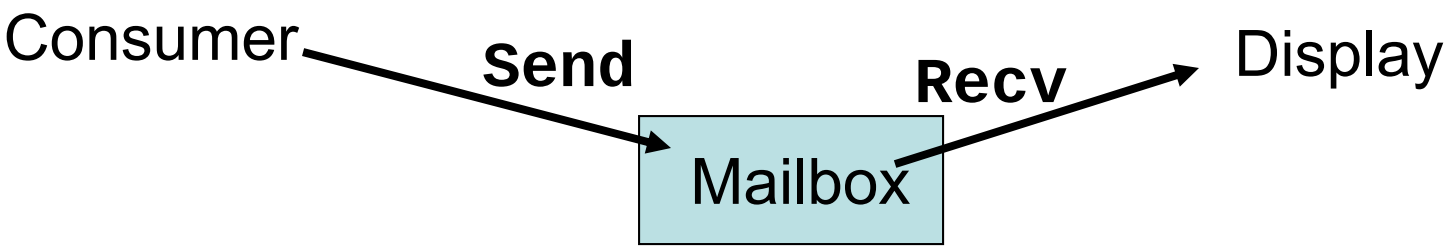
What does the semaphores mean?

What would be a better name for s?

# Mailbox

**MailBox_Send**
- **bWait(&BoxFree)**
- **Put data into Mailbox**
- **bSignal(&DataValid)**

**MailBox_Recv**
- **bWait(&DataValid)**
- **Retrieve data from Mailbox**
- **bSignal(&BoxFree)**

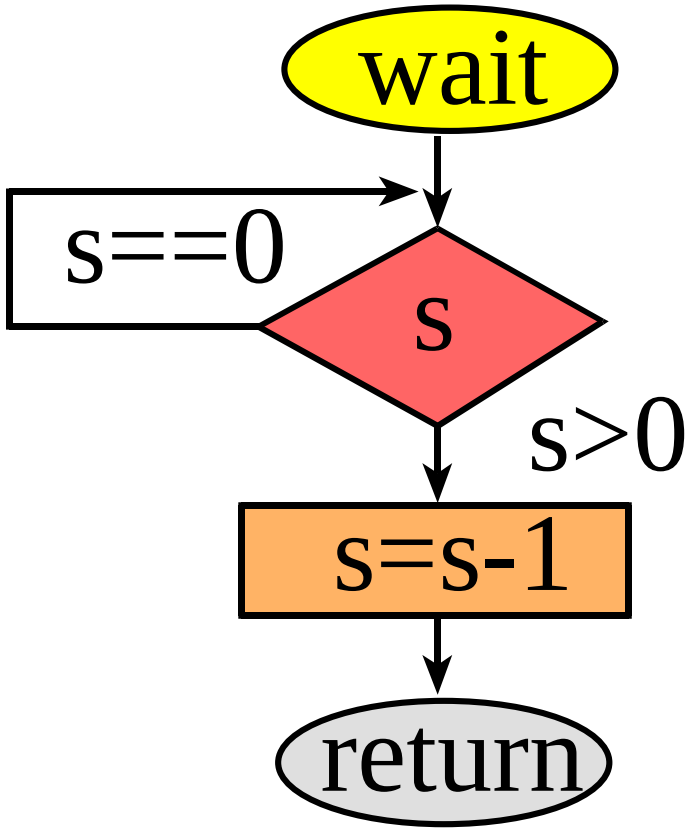


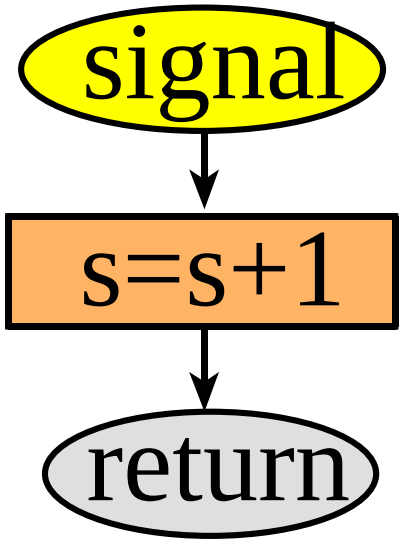

What do the semaphores mean?

What are the initial values?

What if we remove **bWait(&BoxFree)** and **bSignal(&BoxFree)**?

# Spin-lock counting




What does the semaphore mean?

What to do with the I bit?

# Spin-lock semaphores

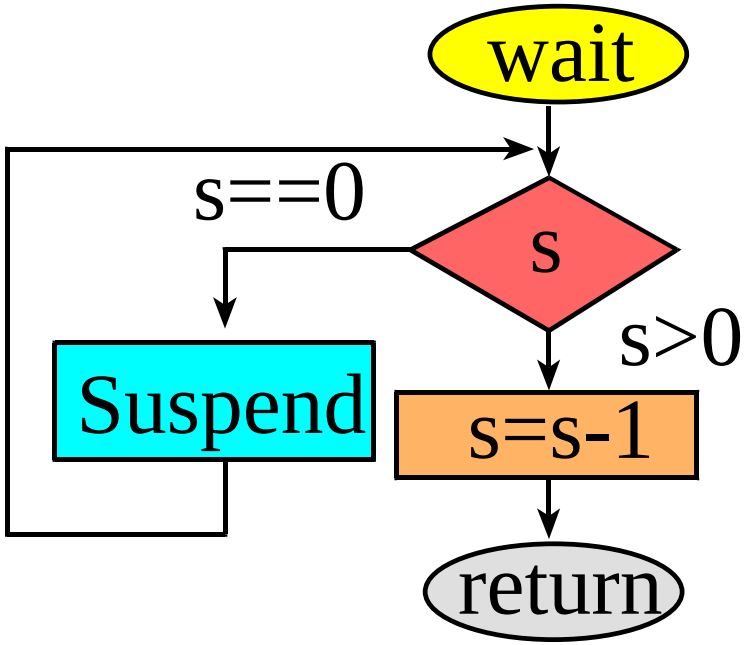
```
OS_Wait ;R0 points to counter
   LDREX   R1, [R0] ; counter
   SUBS    R1, #1   ; counter -1,
   ITT     PL       ; ok if >= 0
   STREXPL R2,R1,[R0]  ; try update
   CMPPL   R2, #0   ; succeed?
   BNE     OS_Wait  ; no, try again
   BX      LR
OS_Signal ; R0 points to counter
   LDREX   R1, [R0]  ; counter
   ADD     R1, #1    ; counter + 1
   STREX   R2,R1,[R0]  ; try update
   CMP     R2, #0    ; succeed?
   BNE     OS_Signal ;no, try again
   BX      LR
```

```
void OS_Wait(long *s){
  DisableInterrupts();
  while((*s) <= 0){
    EnableInterrupts();
    DisableInterrupts();
  }
  (*s) = (*s) - 1;
  EnableInterrupts();
}
void OS_Signal(long *s){
  long status;
  status = StartCritical();
  (*s) = (*s) + 1;
  EndCritical(status);
}
```

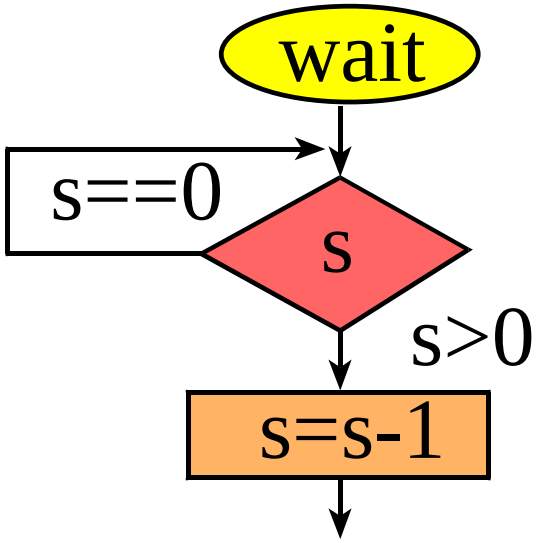
**Program 4.11**

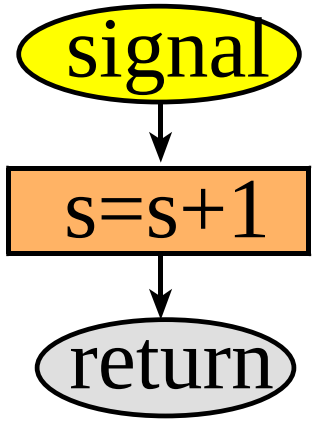
# Cooperative spin-lock

## Cooperative spin-lock




## Regular spin-lock




Why would you want a timeout error?
How would you implement timeout?

```
if(OS_Wait(&free,T100ms)){
  // use it
  OS_Signal(&free);
  } else{
   // error
}
```

*Could be implemented with a catch and throw*

# Cooperative semaphores

```
void OS_Wait(long *s){
  DisableInterrupts();
  while((*s) <= 0){
    EnableInterrupts();
    OS_Suspend();
    DisableInterrupts();
  }
  (*s) = (*s) - 1;
  EnableInterrupts();
}
void OS_Signal(long *s){
  long status;
  status = StartCritical();
  (*s) = (*s) + 1;
  EndCritical(status);
}
```


Do an experiment of Lab 2 with and without cooperation

# FIFO, queue, or Pipe

**FIFO_Put**

```
Wait(&DataRoomLeft)
Disable Interrupts
Enter data into Fifo
Enable Interrupts
Signal(&DataAvailable)
```

**FIFO_Get**

```
Wait(&DataAvailable)
Disable Interrupts
Remove data from Fifo
Enable Interrupts
Signal(&DataRoomLeft)
```

---

**FIFO_Put**

```
Wait(&DataRoomLeft)
bWait(&Mutex)
Enter data into Fifo
bSignal(&Mutex)
Signal(&DataAvailable)
```

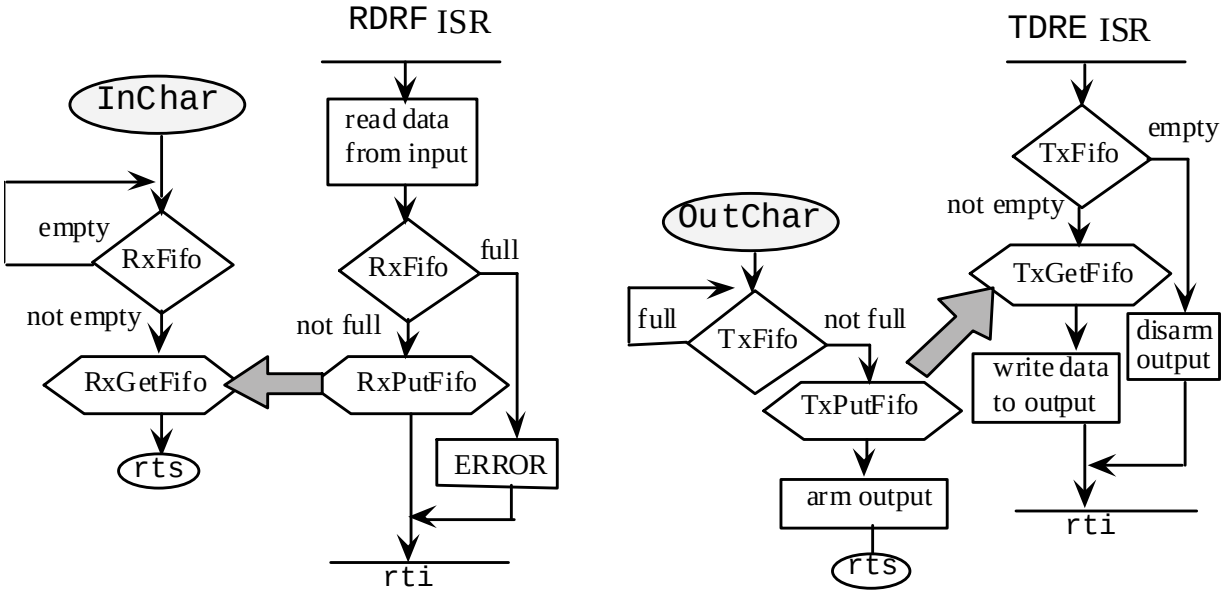
**FIFO_Get**

```
Wait(&DataAvailable)
bWait(&Mutex)
Remove data from Fifo
bSignal(&Mutex)
Signal(&DataRoomLeft)
```

What do the semaphores mean?
What if the FIFO never fills?

# Can’t wait from background

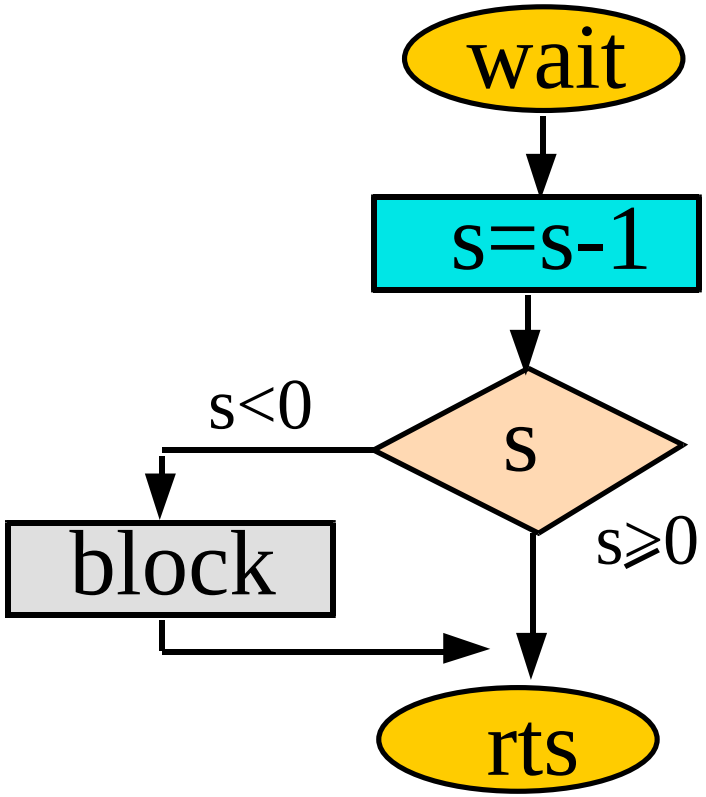
- Redo Mailbox if **`Send`** in background
- Redo Fifo if **`Put`** in background (RX)
- Redo Fifo if **`Get`** in background (TX)

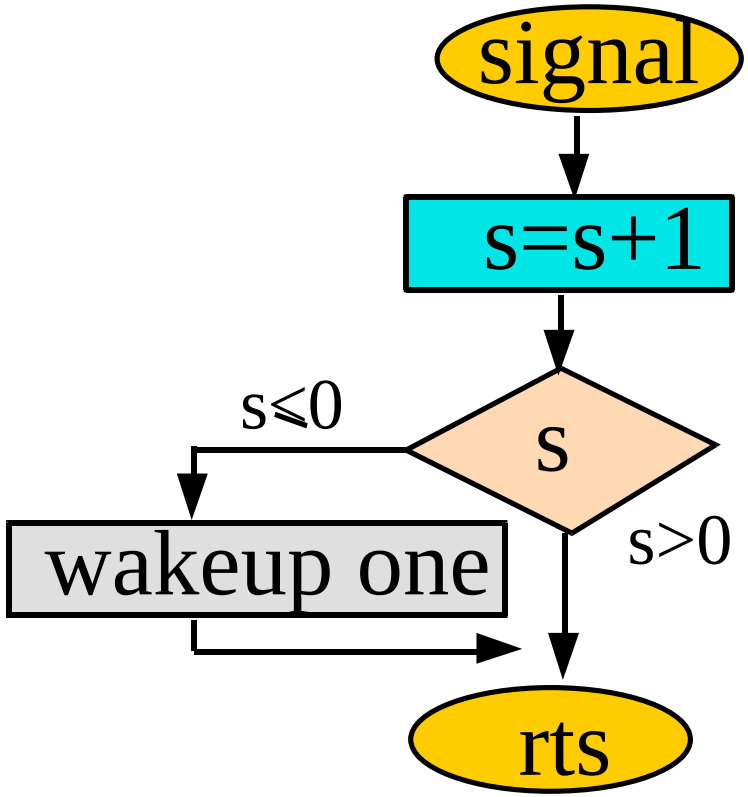


# Blocking semaphore (Lab 3)

- Recapture time lost in the spin operation of spin-lock
- Eliminate wasted time running threads that are not doing work (e.g.., waiting)
- Implement **bounded waiting**
  - once thread calls **Wait** and is not serviced,
  - there are a finite number of threads that will go ahead

# Blocking semaphore




What does the semaphores mean?

What to do with I bit?

# 1) Blocking semaphore

**OS_Wait(Sema4Type *semaPt)**

**1) Save the I bit and disable interrupts**

**2) Decrement the semaphore counter, S=S-1**

```
(semaPt->Value)--;
```

**3) If the Value < 0 then this thread will be blocked**

**set the status of this thread to blocked,**

**specify this thread blocked on this semaphore**

**suspend thread**

**4) Restore the I bit**

# 1) Blocking semaphore

**OS_Signal (Sema4Type *semaPt)**

**1) Save I bit, then disable interrupts**

**2) Increment the semaphore counter, S=S+1**

```
(semaPt->Value)++;
```

**3) If the Value ≤ 0 then**

**Wake up one thread from the TCB linked list**

- **Bounded waiting -> the one waiting the longest**
- **Priority -> the one with highest priority**

**Move TCB of the "wakeup" thread from the blocked list to the active list**

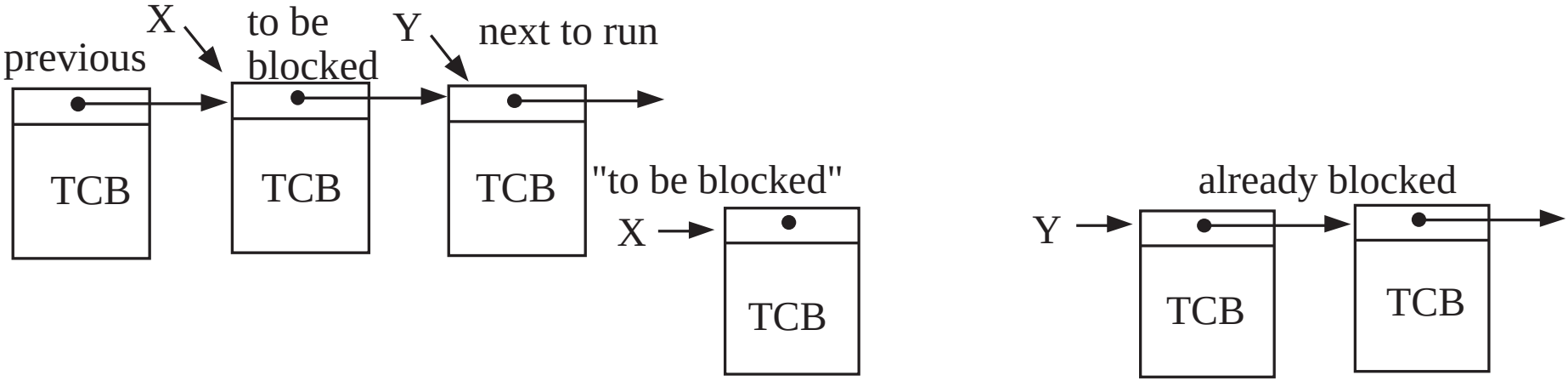
***What to do with the thread that called OS_Signal?***

- **Round robin -> do not suspend**
- **Priority -> suspend if wakeup thread is higher priority**

**4) Restore I bit**

# 1) Blocking semaphore

Each semaphore has a blocked TCB linked list
- contains the threads that are blocked
- empty if semaphore **Value** ≥ 0
- e.g., if **Value** == -2, then two threads are blocked
- order on blocked list determine sequence of blocking
- sequence of blocking determine which to wake up




How is the scheduler different?

# 2) Blocking semaphore

-All threads exist on circular TCB list: active and blocked
-Each semaphore simply has a **Value**
-No blocked threads if semaphore **Value** ≥ 0
  e.g., if **Value** is -2, then two threads are blocked
-No information about which thread has waited longest
-Add to TCB, a **BlockPt**, of type **Sema4Type**
    initially, this pointer is **null**
    **null** means this thread is active and ready to run
-If blocked, this pointer contains the semaphore address

**New Scheduler**

Find the next active thread from the TCB list
only run threads with **BlockPt** equal to **null**

# 2) Blocking semaphore

**OS_Wait(Sema4Type *semaPt)**

1) Disable interrupts, I=1
2) Decrement the semaphore counter, S=S-1
   **(semaPt->Value)--;**
3) If the **Value<0** then this thread will be blocked
   specify this thread is blocked to this semaphore
   **RunPt->BlockPt = semaPt;**
   suspend thread**;**
4) Enable interrupts, I=0

# 2) Blocking semaphore

**OS_Signal(Sema4Type *semaPt)**

1) Save I bit, then disable interrupts
2) Increment the semaphore Value, S=S+1
   **(semaPt->Value)++;**
3) If **Value ≤ 0** then
   wake up one thread from the **TCB** linked list
   (no bounded waiting)
   do not suspend the thread that called **OS_Signal**
   search TCBs for thread with **BlockPt == semaPt**
   set the **BlockPt** of this **TCB** to **null**
4) Restore I bit

How is the scheduler different?

# Applications

- Sequential execution
  - **Run-A** then **Run-B** then **Run-C**
- Rendezvous
- Event trigger
  - **Event-A** and **Event-B**
  - **Event-A** or **Event-B**
- Fork and join
- Readers-Writers Problem

Look at old exams

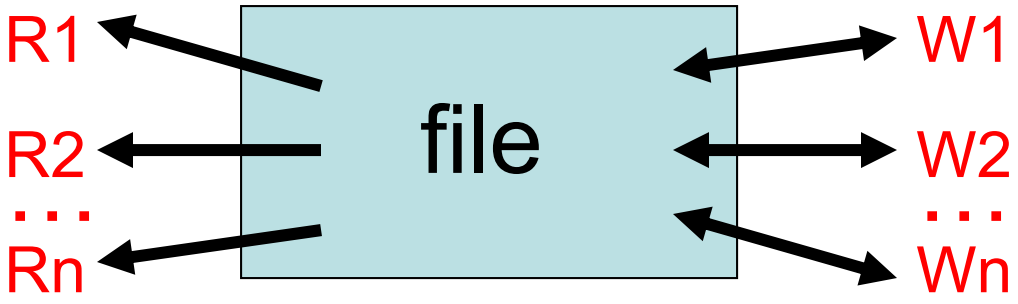
# Readers-Writers Problem

**Reader Threads**

1) Execute **ROpen(file)**
2) Read information from **file**
3) Execute **RClose(file)**

**Writer Threads**

1) Execute **WOpen(file)**
2) Read information from **file**
3) Write information to **file**
4) Execute **WClose(file)**




**ReadCount=0, number**
**mutex=1, semaphore**
**wrt=1, semaphore**

# Readers-Writers Problem

**ReadCount, number of Readers that are open**
**mutex, semaphore controlling access to ReadCount**
**wrt, semaphore is true if a writer is allowed access**

**ROpen**

```
wait(&mutex);
ReadCount++;
if(ReadCount==1) wait(&wrt)
signal(&mutex);
```

**WOpen**

```
wait(&wrt);
```

**RClose**

```
wait(&mutex);
ReadCount--;
if(ReadCount==0) signal(&wrt)
signal(&mutex);
```

**WClose**

```
signal(&wrt);
```

# Cool stuff we'll make the graduate students do

- Bounded waiting
- Time-out
- Deadlock detection
  - Wait-for-graph
  - Resource allocation graph

Two names for the same thing

Works for single instance resources

- Two types of boxes
  Threads, resources
- Two types of arrows
  Assignment, request