Chapter 2 (Second Part)

Interprocess Communication and Synchronization

Slide Credits:
Jonathan Walpole
Andrew Tanenbaum
Outline

Race Conditions
Mutual Exclusion and Critical Regions
Mutex Locks
Test-And-Set Instruction
Sleep, Yield, Wakeup
Disabling Interrupts in the Kernel
Classical IPC Problems:
  • Producer-Consumer
  • Readers-Writers
  • Dining Philosophers
  • Sleeping Barber
Multiple Processes will Cooperate

Assumptions:
- Two or more threads (or processes)
- Each executes in (pseudo) parallel
- Cannot predict exact running speeds
- The threads can interact
  - Example: Access to a shared variable

Example:
- One thread writes a variable
- The other thread reads from the same variable

Problem:
- The order of READs and WRITEs can make a difference!!!
Race Condition: An Example

Incrementing a counter (load, increment, store)
Context switch can occur after load and before increment!

```
Spooler directory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>abc</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>prog.c</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>prog.n</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

out = 4
in = 7
Race Conditions

Whenever the output depends on the precise execution order of the processes!!!

Why do race conditions occur?
- values of memory locations replicated in registers during execution
- context switches at arbitrary times during execution
- threads can see “stale” memory values in registers

What solutions can we apply?
- prevent context switches by preventing interrupts
- make threads coordinate with each other to ensure mutual exclusion in accessing “critical sections” of code
Mutual Exclusion

**Critical Region (Critical Section):**

The part of the code accessing shared data

**Desired Conditions:**

(1) No two threads simultaneously in critical region.
(2) No assumptions made about speeds or numbers of CPUs.
(3) No thread running outside its critical region may block another thread.
(4) No thread must wait forever to enter its critical region (no “starvation”).
Critical regions with mutual exclusion

A enters critical region
A leaves critical region

Process A

Process B

B attempts to enter critical region
B enters critical region
B leaves critical region

B blocked

Time

T₁ T₂ T₃ T₄
How can we enforce mutual exclusion?

*What about using a binary “lock” variable in memory and having threads check it and set it before entry to critical regions?*

Solves the problem of exclusive access to shared data. Expresses intention to enter *Critical Section*.

Acquiring a lock prevents concurrent access.

**Assumptions:**

*Every thread sets lock before accessing shared data!*
*Every thread releases the lock after it is done!*
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Free Lock
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Lock

Free
Acquiring and Releasing Locks

Thread A

Lock

Set

Thread B

Thread C

Thread D
Acquiring and Releasing Locks
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Set

Lock
Acquiring and Releasing Locks

Thread A

Thread B

Lock

Thread C

Thread D

Set

Lock
Acquiring and Releasing Locks

Thread A

Thread B

Lock

Thread C

Thread D

Set

Lock
Acquiring and Releasing Locks

Thread A

Thread B

Lock

Thread C

Lock

Thread D

Lock

Set

Lock
Acquiring and Releasing Locks
Acquiring and Releasing Locks
Acquiring and Releasing Locks

Thread A

Lock

Thread B

Unlock

Thread C

Lock

Thread D

Lock

Set

Lock
Acquiring and Releasing Locks

Thread A

Thread B

Lock

Thread C

Lock

Thread D

Lock

Free
Acquiring and Releasing Locks
Acquiring and Releasing Locks
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock

Lock

Lock
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Set Lock

Lock

Lock
Mutex Locks

- An abstract data type
- Can be used for synchronization and mutual exclusion
- The “mutex” is either:
  - Locked ("the lock is held")
  - Unlocked ("the lock is free")

- Two operations:

<table>
<thead>
<tr>
<th><strong>Lock</strong> (<strong>mutex</strong>)</th>
<th>Acquire the lock, if it is free</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If the lock is not free, then wait until it can be acquired</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Unlock</strong> (<strong>mutex</strong>)</th>
<th>Release the lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If there are waiting threads, then wake up one of them</td>
</tr>
</tbody>
</table>

Both **Lock** and **Unlock** are assumed to be *atomic***!!
(A kernel implementation will ensure atomicity)
An Example using a Mutex

Shared data:

Mutex myLock;

1 repeat
2   Lock(myLock);
3   critical section
4   Unlock(myLock);
5   remainder section
6 until FALSE

1 repeat
2   Lock(myLock);
3   critical section
4   Unlock(myLock);
5   remainder section
6 until FALSE
How can we implement mutual exclusion?

Many computers have *some limited* hardware support for setting locks...

- Atomic “Test and Set Lock” instruction
- Atomic “Compare and Swap” operation

Can be used to implement “Mutex” locks
The “Test-And-Set” Instruction (TSL, tset)

A lock is a variable with two values
• One word:
  0=FALSE=not locked
  1=TRUE=locked

Test-and-set does the following atomically:
• Get the (old) value
• Set the lock to TRUE
• Return the old value

If the returned value was FALSE...
Then you got the lock!!!
If the returned value was TRUE...
Then someone else already has the lock.
(so try again later)
Critical section entry code with TSL

This code ensures that...

Only one thread at a time will enter its “critical section”.

```
1 repeat
2   while(TSL(lock))
3     no-op;
4   critical section
5   lock = FALSE;
6   remainder section
7 until FALSE
```

```
1 repeat
2   while(TSL(lock))
3     no-op;
4   critical section
5   lock = FALSE;
6   remainder section
7 until FALSE
```
Busy Waiting

Also called “Polling”

*The thread consumes CPU cycles to evaluate when lock becomes free!!!*

• Shortcoming:
  On a single CPU system...
  A busy-waiting thread can prevent the lock holder from running & completing its critical section & releasing the lock!

**Better: Block instead of busy wait**
  *(on a single CPU system)*
Synchronization Primitives

**Sleep**
- Put a thread to sleep
- Thread becomes BLOCKed

**Wakeup**
- Move a BLOCKed thread back onto “Ready List”
- Thread becomes READY (or RUNNING)

**Yield**
- Move to another thread
- Does not BLOCK thread
- Just gives up the current time-slice

*But how can these be implemented?*
Synchronization Primitives

Sleep
Wakeup
Yield
ThreadCreateAndStart
ThreadKill
...etc...

Implementation:

In User Programs:
   Syscalls to kernel
In Kernel:
   Calls to the thread “Scheduler” routines
Concurrency Control in the Kernel

Different threads call Yield, Sleep, ...
Scheduler routines manipulate the “Ready List”
Ready List is shared data

Problem:
How can scheduler routines be programmed correctly?

Solution:
• Scheduler can disable interrupts, or
• Scheduler can use “Test And Set Lock” instruction
Disabling interrupts

Disabling interrupts in the OS

vs

Disabling interrupts in user processes

• Why not allow user processes to disable interrupts?
• Is it ok to disable interrupts in the OS?
• What precautions should you take?
Disabling interrupts in the Kernel

Scenario
- A thread is running; wants to access shared data
- Disable interrupts
- Access shared data ("critical section")
- Enable interrupts
Disabling interrupts in the Kernel

Scenario
A thread is running; wants to access shared data
Disable interrupts
Access shared data ("critical section")
Enable interrupts

Scenario #2
Interrupts are already disabled
Thread wants to access critical section
...using the above sequence...
Disabling interrupts in the Kernel

Scenario
A thread is running; wants to access shared data
Save previous interrupt status (enabled/disabled)
Disable interrupts
Access shared data (“critical section”)
Restore interrupt status to what it was before

Scenario #2
Interrupts are already disabled
Thread wants to access critical section
...using the above sequence...
Classical IPC Synchronization Problems

**Producer-Consumer**
- One thread produces data items
- Another thread consumes them
- Use a bounded buffer / queue between the threads
- The buffer is a shared resource
  - Must control access to it!!!
- Must suspend the producer thread if buffer is full
- Must suspend the consumer thread if buffer is empty

**Readers and Writers**

**Dining Philosophers**

**Sleeping Barber**
Producer/Consumer with Busy Waiting

```
thread producer {
    while(1){
        // Produce char c
        while (count==n) {
            no_op
        }
        buf[InP] = c
        InP = InP + 1 mod n
        count++
    }
}

thread consumer {
    while(1){
        while (count==0) {
            no_op
        }
        c = buf[OutP]
        OutP = OutP + 1 mod n
        count--
        // Consume char
    }
}
```

Global variables:
- char buf[n]
- int InP = 0  // place to add
- int OutP = 0  // place to get
- int count = 0
Problems with this code

- Count variable can be corrupted if context switch occurs at the wrong time
- A race condition exists! *Race bugs very difficult to track down*
- What if buffer is full?  
  Produce will busy-wait  
  Consumer will not be able to empty the buffer
- What if buffer is empty?  
  Consumer will busy-wait  
  Producer will not be able to fill the buffer
Problems with this code

• Count variable can be corrupted if context switch occurs at the wrong time
• A race condition exists!
  *Race bugs very difficult to track down*
• What if buffer is full?
  Produce will busy-wait
  Consumer will not be able to empty the buffer
• What if buffer is empty?
  Consumer will busy-wait
  Producer will not be able to fill the buffer

addressing these issues next...
Producer/Consumer with Blocking

```
0  thread producer {
1    while(1) {
2      // Produce char c
3      if (count==n) {
4        sleep(full)
5      }
6      buf[InP] = c;
7      InP = InP + 1 mod n
8      count++
9      if (count == 1)
10         wakeup(empty)
11    }
12  }

0  thread consumer {
1    while(1) {
2      while (count==0) {
3        sleep(empty)
4      }
5      c = buf[OutP]
6      OutP = OutP + 1 mod n
7      count--;
8      if (count == n-1)
9        wakeup(full)
10     // Consume char
11    }
12  }
```

Global variables:

- char buf[n]
- int InP = 0  // place to add
- int OutP = 0  // place to get
- int count = 0
This code is still incorrect!

The “count” variable can be corrupted:

Increments or decrements may be lost!
Possible Consequences:
• Both threads may sleep forever
• Buffer contents may be over-written

What is this problem called?
This code is still incorrect!

The “count” variable can be corrupted:

Increments or decrements may be lost!
Possible Consequences:
• Both threads may sleep forever
• Buffer contents may be over-written

What is this problem called? *Race Condition*

Code that manipulates count must be made into a ??? and protected using ???.
This code is still incorrect!

The “count” variable can be corrupted:

- Increments or decrements may be lost!
- Possible Consequences:
  - Both threads may sleep forever
  - Buffer contents may be over-written

What is this problem called? **Race Condition**

Code that manipulates count must be made into a **Critical Section** and protected using **Mutual Exclusion**.
Semaphores

- An abstract data type that can be used for condition synchronization and mutual exclusion

- Integer variable with two operations:

<table>
<thead>
<tr>
<th><strong>Down</strong> <em>(sem)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement <em>sem</em> by 1</td>
</tr>
<tr>
<td>if <em>sem</em> would go negative, “wait” until possible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Up</strong> <em>(sem)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment <em>sem</em> by 1</td>
</tr>
<tr>
<td>if any threads are waiting, wake one of them up</td>
</tr>
</tbody>
</table>

The integer will always be >= 0.
Both **Up()** and **Down()** are assumed to be *atomic***!!*
A kernel implementation will ensure atomicity
Semaphores

There are multiple names for the two operations

\[ \text{Down}(S) \quad \text{Wait}(S) \quad \text{P}(S) \]
\[ \text{Up}(S) \quad \text{Signal}(S) \quad \text{V}(S) \]

Each semaphore contains an integer...

\textbf{Up} (sometimes called \textit{Signal})
Increment integer
(May wake up another thread)

\textbf{Down} (sometimes called \textit{Wait})
Decrement integer, but never go negative.
(May cause the thread to sleep)
Semaphores

There are multiple names for the two operations

- `Down(S)`  `Wait(S)`  `P(S)`
- `Up(S)`    `Signal(S)`  `V(S)`

Each semaphore contains an integer...

- **Up** (sometimes called **Signal**)
  - Increment integer
  - (May wake up another thread)
- **Down** (sometimes called **Wait**)
  - Decrement integer, but never go negative.
  - (May cause the thread to sleep)

*But you must NEVER access the integer directly!!!*

*Why?*
Variation: Binary Semaphores

Semaphore (normal)
(Sometimes called “counting semaphore”)

Binary Semaphore
A specialized use of semaphores
The semaphore is used to implement a Mutex Lock
Variation: Binary Semaphores

**Semaphore (normal)**
(Sometimes called “counting semaphore”)

**Binary Semaphore**
A specialized use of semaphores
The semaphore is used to implement a *Mutex Lock*
The count will always be either
  0 = locked
  1 = unlocked
Variation: Binary Semaphores

Semaphore (normal)
(Sometimes called “counting semaphore”)

Binary Semaphore
A specialized use of semaphores
The semaphore is used to implement a Mutex Lock
The count will always be either

0 = locked
1 = unlocked

Up = Unlock the mutex (may wake up another thread)
Down = Lock (may wait if already locked)
Using Semaphores for Mutex

semaphore mutex = 1

```
1 repeat
2   down(mutex);
3   critical section
4   up(mutex);
5   remainder section
6   until FALSE
```

Thread A

```
1 repeat
2   down(mutex);
3   critical section
4   up(mutex);
5   remainder section
6   until FALSE
```

Thread B
Using Semaphores for Mutex

semaphore mutex = 0

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread A

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread B
Using Semaphores for Mutex

\[ \text{semaphore mutex} = 0 \]

\[
\begin{align*}
1 & \text{ repeat} \\
2 & \quad \text{down(mutex)}; \\
3 & \quad \text{critical section} \\
4 & \quad \text{up(mutex)}; \\
5 & \quad \text{remainder section} \\
6 & \quad \text{until FALSE}
\end{align*}
\]

Thread A

\[
\begin{align*}
1 & \text{ repeat} \\
2 & \quad \text{down(mutex)}; \\
3 & \quad \text{critical section} \\
4 & \quad \text{up(mutex)}; \\
5 & \quad \text{remainder section} \\
6 & \quad \text{until FALSE}
\end{align*}
\]

Thread B
Using Semaphores for Mutex

\[\text{semaphore mutex} = 0\]

1 repeat
2 \text{down}(mutex); \quad \text{down}(mutex);
3 \text{critical section} \quad \text{critical section}
4 \text{up}(mutex); \quad \text{up}(mutex);
5 \text{remainder section} \quad \text{remainder section}
6 \text{until FALSE} \quad \text{until FALSE}

Thread A

Thread B
Using Semaphores for Mutex

semaphore mutex = 1

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread A

Thread B
Using Semaphores for Mutex

semaphore mutex = 0

This thread is released.
It can now proceed!

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread A

Thread B
Project 2…

Implement Producer-Consumer Solution
... in BLITZ framework
Counting semaphores in producer/consumer

Global variables
    semaphore full_buffs = 0;
    semaphore empty_buffs = n;
    char buff[n];
    int InP, OutP;

0 thread producer {
1   while(1){
2       // Produce char c...
3       down(empty_buffs)
4       buf[InP] = c
5       InP = InP + 1 mod n
6       up(full_buffs)
7   }
8 }

0 thread consumer {
1   while(1){
2       down(full_buffs)
3       c = buf[OutP]
4       OutP = OutP + 1 mod n
5       up(empty_buffs)
6       // Consume char...
7   }
8 }
Implementing Semaphores

**Hardware mechanisms to support semaphores:**
- Control over interrupts (almost all computers)
- Special atomic instructions in ISA
  - test and set lock
  - compare and swap

**Techniques**
- Spin-locks (busy waiting)
  - may waste a lot of cycles on uni-processors
- Blocking the thread
  - may waste a lot of cycles on multi-processors
Implementing Semaphores (using blocking)

```c
struct semaphore {
    int val;
    list L;
};

Down(semaphore sem)
    DISABLE_INTS
    sem.val--
    if (sem.val < 0) {
        add thread to sem.L
        block(thread)
    }
    ENABLE_INTS

Up(semaphore sem)
    DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
        th = remove next
        thread from sem.L
        wakeup(th)
    }
    ENABLE_INTS
```
Semaphores in UNIX

See the POSIX specification for details.
Two varieties...

**Named Semaphores**
- Have an associated “name”
- Can be used from different processes

**Unnamed Semaphores**
- Local to a single address space.
POSIX Semaphore Example

```
#include <semaphore.h>
sem_t * my_sem_ptr;

main(){
    sem_init(my_sem_ptr, shared, 1);
    ...
    sem_wait (my_sem_ptr);

    [CRITICAL SECTION]

    sem_post (my_sem_ptr);
    ...
    sem_destroy (my_sem_ptr);
}
```
Managing your UNIX semaphores

Listing currently allocated ipc resources

   ipcs

Removing semaphores

   ipcrm -s <sem number>
Implementation Possibilities

• Implement Mutex Locks
  ... Using Semaphores

• Implement Counting Semaphores
  ... Using Binary Semaphores
  ... Using Mutex Locks

• Implement Binary Semaphores
  ... etc

Can also implement using
Test-And-Set
Calls to Sleep, Wake-Up
Dining Philosophers Problem

Five philosophers sit at a table
Between each philosopher there is one fork

Philosophers:

```
while(TRUE) {
    Think();
    Grab first fork;
    Grab second fork;
    Eat();
    Put down first fork;
    Put down second fork;
}
```

Why do they need to synchronize?
How should they do it?

Each philosopher is modelled with a thread
Dining philosopher’s solution???

Why doesn’t this work?

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_fork(i);
        take_fork((i+1)% N);
        Eat();
        put_fork(i);
        put_fork((i+1)% N);
    }
}
```
Dining philosopher’s solution (part 1)

```c
#define N 5  /* number of philosophers */
#define LEFT (i+N-1)%N  /* number of i’s left neighbor */
#define RIGHT (i+1)%N  /* number of i’s right neighbor */
#define THINKING 0  /* philosopher is thinking */
#define HUNGRY 1  /* philosopher is trying to get forks */
#define EATING 2  /* philosopher is eating */

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)  /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {
        think();  /* repeat forever */
        take_forks(i);  /* philosopher is thinking */
        eat();  /* acquire two forks or block */
        put_forks(i);  /* yum-yum, spaghetti */
        put_forks(i);  /* put both forks back on table */
    }
}
```
Dining philosopher’s solution (part 2)

```c
void take_forks(int i) /* i: philosopher number, from 0 to N–1 */
{
    down(&mutext);     /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i);          /* try to acquire 2 forks */
    up(&mutext);      /* exit critical region */
    down(&s[i]);     /* block if forks were not acquired */
}

void put_forks(i) /* i: philosopher number, from 0 to N–1 */
{
    down(&mutext);     /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT);        /* see if left neighbor can now eat */
    test(RIGHT);       /* see if right neighbor can now eat */
    up(&mutext);       /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```
Dining Philosophers

Is this correct?
What does it mean for it to be correct?
Is there an easier way?
The Sleeping Barber Problem
The Sleeping Barber Problem

**Barber:**
While there are people waiting for a hair cut, put one in the barber chair, and give him a haircut. When done, move to the next customer. Else go to sleep, until someone comes in.

**Customer:**
If the barber is sleeping, wake him up and get a haircut. If someone is getting a haircut...
wait for the barber to free up by sitting in a chair. If the waiting chairs are all full, leave the barbershop.
Solution to the sleeping barber problem

```plaintext
const CHAIRS = 5
var customerReady: Semaphore
   barberReady: Semaphore
   lock: Mutex
   numWaiting: int = 0

Barber_Thread:
   while true
      Down(customerReady)
      Lock(lock)
      numWaiting = numWaiting-1
      Up(barberReady)
      Unlock(lock)
   endWhile

Customer_Thread:
   Lock(lock)
   if numWaiting < CHAIRS
      numWaiting = numWaiting+1
      Up(customerReady)
      Unlock(lock)
      Down(barberReady)
      GetHaircut()
   else -- give up & go home
      Unlock(lock)
   endIf
```
The Readers and Writers Problem

- Readers and writers want to access a database. (Each is a thread)
- Multiple readers can proceed concurrently.
- Writers must synchronize with readers and other writers.
  
  *Only one writer at a time.*
  *When someone is writing, must be no readers.*

**Goals:**
- Maximize concurrency.
- Prevent starvation.
One solution to readers and writers

**Writer Thread:**
while true
    Lock(dbLock)
    ...Write shared database...
    Unlock(dbLock)
    ...Remainder section...
endWhile

**Additional Code:**

```java
var dbLock: Mutex
nReaders: int = 0
counterAccess: Mutex
```
One solution to readers and writers

**Reader Thread:**

```java
while true
    Lock(counterAccess)
    nReaders = nReaders + 1
    if nReaders == 1
        Lock(dbLock)
    endIf
    Unlock(counterAccess)
    ... Read shared database...
    Lock(counterAccess)
    nReaders = nReaders - 1
    if nReaders == 0
        Unlock(dbLock)
    endIf
    Unlock(counterAccess)
    ... Remainder section...
endWhile
```

```java
var dbLock: Mutex
nReaders: int = 0
counterAccess: Mutex
```
Readers and Writers – No Starvation

**Concurrency Control:**

```plaintext
var
dbLock: Mutex  // Protects the shared database
nreaders: int = 0  // How many readers are in the database
counterAccess: Mutex  // Protects the nreaders variable
waitListLock: Mutex
```

**Goal:**

Many readers can get in to the database
... unless a writer is waiting.
Then, any new readers must wait until after the writer leaves the database.
Readers and Writers – No Starvation

**Writer Thread:**

```java
while true
    Lock(waitListLock)
    Lock(dbLock)
    Unlock(waitListLock)
    ...Write shared database...
    Unlock(dbLock)
    ...Remainder section...
endWhile
```

If a writer is waiting to get into the database, then `waitListLock` will be locked. This will prevent any new readers from getting in. Assumption: Threads waiting on `waitListLock` will be unlocked in FIFO order.
Readers and Writers – No Starvation

**Reader Thread:**

```plaintext
while true
    Lock(waitListLock)
    Lock(counterAccess)
    nreaders = nreaders + 1
    if nreaders == 1 then
        Lock(dbLock)
    endIf
    Unlock(counterAccess)
    Unlock(waitListLock)
    ...Read shared database...
    Lock(counterAccess)
    nreaders = nreaders - 1
    if nreaders == 0 then
        Unlock(dbLock)
    endIf
    Unlock(counterAccess)
    ...Remainder section...
endWhile
```
Implementing Counting Semaphores

**Problem:** Implement a counting semaphore

Up ()
Down ()
...using just Mutex locks.
Possible Solution

var cnt: int = 0        -- Signal count
var m1: Mutex           -- Protects access to "cnt"
    m2: Mutex = locked   -- Locked when waiting

**Down ():**

Lock(m1)
cnt = cnt - 1
if cnt<0
    Unlock(m1)
    Lock(m2)
else
    Unlock(m1)
endIf

**Up ():**

Lock(m1)
cnt = cnt + 1
if cnt<=0
    Unlock(m2)
endIf
Unlock(m1)
Possible Solution

```plaintext
var cnt: int = 0          -- Signal count
var m1: Mutex            -- Protects access to “cnt”
    m2: Mutex = locked  -- Locked when waiting

Down ():
    Lock(m1)
    cnt = cnt - 1
    if cnt<0
        Unlock(m1)
        Lock(m2)
    else
        Unlock(m1)
    endif

Up ():
    Lock(m1)
    cnt = cnt + 1
    if cnt<=0
        Unlock(m2)
        Unlock(m1)
    endif
```

Contains a Race Condition!
STILL INCORRECT

```plaintext
var cnt: int = 0        -- Signal count
var m1: Mutex           -- Protects access to “cnt”
    m2: Mutex = locked  -- Locked when waiting
    m3: Mutex

Down ():
    Lock(m3)
    Lock(m1)
    cnt = cnt – 1
    if cnt<0
        Unlock(m1)
        Lock(m2)
    else
        Unlock(m1)
    endIf
Unlock(m3)

Up ():
    Lock(m1)
    cnt = cnt + 1
    if cnt<=0
        Unlock(m2)
    endIf
    Unlock(m1)
```