بسم الله الرحمن الرحیم

«سیستم عامل»

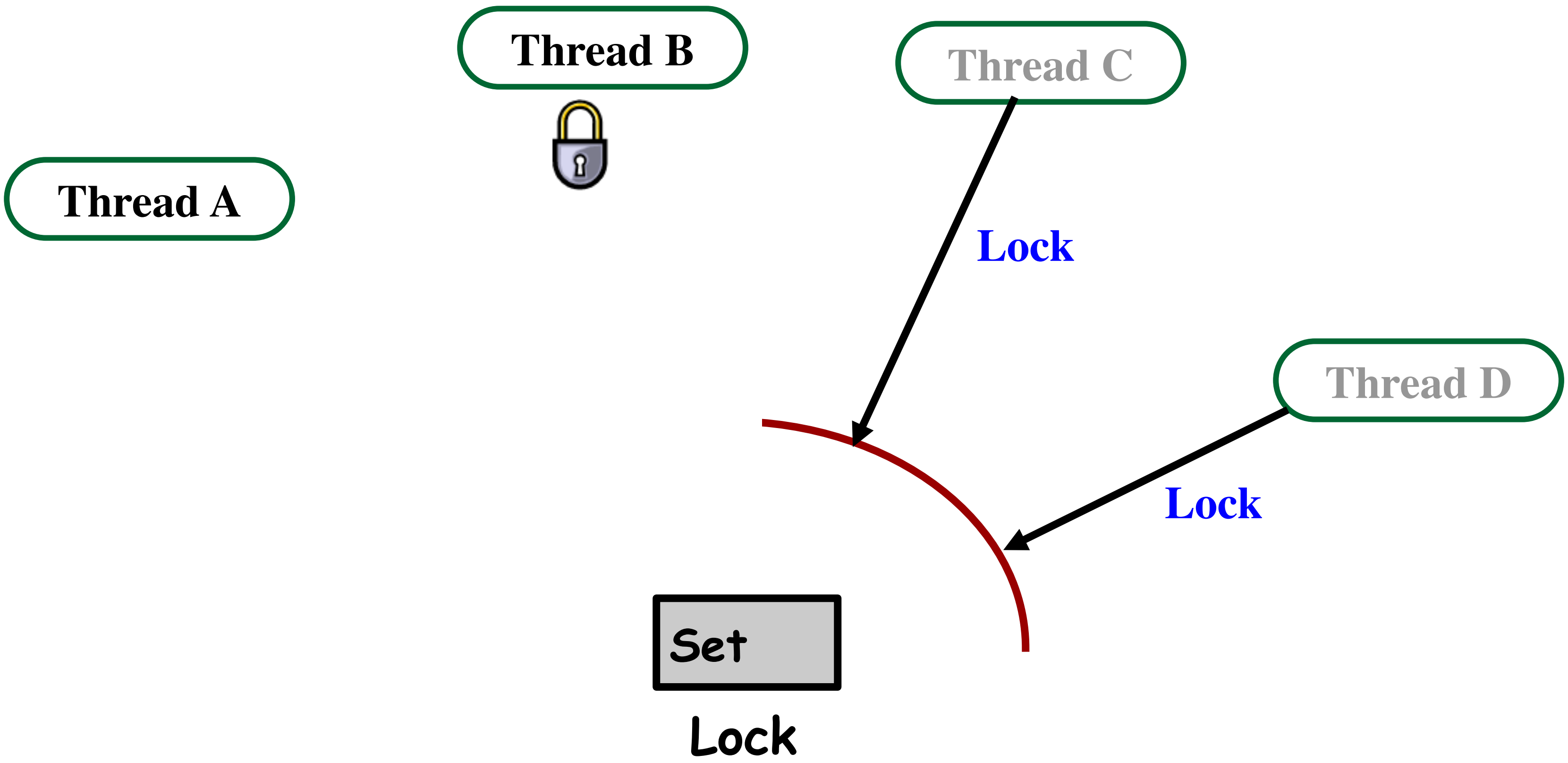
۱

جلسه ۷: کنترل هم‌روندی

یادآوری

- وضعیت رقابتی:
  - حافظه مشترک، اجرای موازی ریسمان‌ها،
  - نتایج متفاوت به ازای ترتیب متفاوت اجرا

Mutex



How to use a mutex?

<u>Shared data:</u>

```
Mutex myLock;
```

```
repeat
   Lock(myLock);
   critical section
   Unlock(myLock);
   remainder section
until FALSE
```

```
repeat
   Lock(myLock);
   critical section
   Unlock(myLock);
   remainder section
until FALSE
```

ابزار Test-and-set

- اول مقداردهی و برگرداندن مقدار قبلی
- به صورت (atomic)
- ==> می‌توانیم mutex را پیاده‌سازی کنیم.

کنترل هم‌روندی در هسته

- خاموش کردن interrupt
- صبر تا وقتی mutex آزاد شود
  - برای پردازنده‌ها و coreهای دیگر
- ضرط: وقتی توسط یک پردازنده mutex گرفته شده، وقفه روی همان پردازنده خاموش است.
  - پردازنده‌ای که mutex را دارد، به زودی آزادش می‌کند.

مثالی از کنترل هم‌روندی:

تولیدکننده و مصرف‌کننده

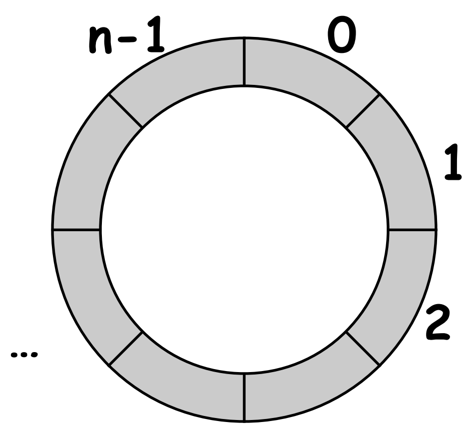
The Producer-Consumer Problem

- **An example of the pipelined model**
  - One thread produces data items
  - Another thread consumes them
- **Use a bounded buffer between the threads**
- **The buffer is a shared resource**
  - Code that manipulates it is a critical section
- **Must suspend the producer thread if the buffer is full**
- **Must suspend the consumer thread if the buffer is empty**

Is this busy-waiting solution correct?

```
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
```

This code is incorrect!

- **The "count" variable can be corrupted:**
  - Increments or decrements may be lost!
  - Possible Consequences:
    - **Both threads may spin forever**
    - **Buffer contents may be over-written**

- **What is this problem called?**

This code is 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 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!

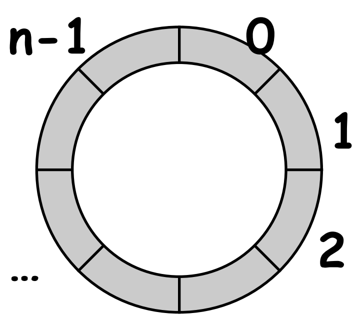
Some more problems with this code

- **What if buffer is full?**
  - Producer will busy-wait
  - On a single CPU system the consumer will not be able to empty the buffer
- **What if buffer is empty?**
  - Consumer will busy-wait
  - On a single CPU system the producer will not be able to fill the buffer
- **We need a solution based on blocking!**

Producer/Consumer with Blocking – 1st attempt

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
  }
}
```

```
thread consumer {
  while(1) {
    if (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```

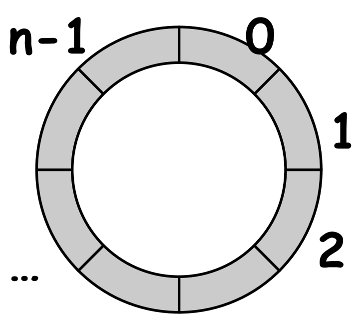



```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

Use a mutex to fix the race condition in this code

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
  }
}
```

```
thread consumer {
  while(1) {
    if (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```




```
Global variables:
    char buf[n]
    int InP = 0   // place to add
    int OutP = 0  // place to get
    int count
```

Problems

- Sleeping while holding the mutex causes deadlock !
- Releasing the mutex then sleeping opens up a window during which a context switch might occur … again risking deadlock
- How can we release the mutex and sleep in a single atomic operation?
- We need a more powerful synchronization primitive

Semaphores

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

  What is the difference between mutual exclusion and condition synchronization?

Semaphores

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

- **Condition synchronization**
  - wait until condition holds before proceeding
  - signal when condition holds so others may proceed

- **Mutual exclusion**
  - only one at a time in a critical section

Semaphores

- **An abstract data type**
  - containing an integer variable (S)
  - Two operations: Wait (S) and Signal (S)

- **Alternative names for the two operations**
  - Wait(S) = Down(S) = P(S)
  - Signal(S) = Up(S) = V(S)

- **Blitz names its semaphore operations Down and Up**

Classical Definition of Wait and Signal

```
Wait(S)
   {
         while S <= 0 do noop;  /* busy wait! */
         S = S - 1;             /* S >= 0 */
   }
Signal (S)
   {
         S = S + 1;
   }
```

Problems with classical definition

- **Waiting threads hold the CPU**
  - Waste of time in single CPU systems
  - Required preemption to avoid deadlock

Blocking implementation of semaphores

Semaphore S has a value, S.val, and a thread list, S.list.

Wait (S)

```
S.val = S.val - 1
If S.val < 0                                /* negative value of S.val */
    {   add calling thread to S.list;       /* is # waiting threads */
            block;                          /* sleep */
    }
```

Signal (S)

```
S.val = S.val + 1
If S.val <= 0
    {   remove a thread T from S.list;
            wakeup (T);
    }
```

Implementing semaphores

- Wait () and Signal () are assumed to be atomic

  How can we ensure that they are atomic?

Implementing semaphores

- **Wait () and Signal () are assumed to be atomic**

  How can we ensure that they are atomic?

- **Implement Wait() and Signal() as system calls?**
  - how can the kernel ensure Wait() and Signal() are completed atomically?
  - Same solutions as before
    - **Disable interrupts, or**
    - **Use TSL-based mutex**

Semaphores with interrupt disabling

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

```
Wait(semaphore sem)
  DISABLE_INTS
    sem.val--
    if (sem.val < 0){
      add thread to sem.L
      sleep(thread)
    }
  ENABLE_INTS
```

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

Semaphores with interrupt disabling

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

```
Wait(semaphore sem)
  DISABLE_INTS
    sem.val--
    if (sem.val < 0){
      add thread to sem.L
      sleep(thread)
    }
  ENABLE_INTS
```

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

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait`
   operation")
   EndIf
   count = count - 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait`
   operation")
   EndIf
   count = count – 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait`
   operation")
   EndIf
   count = count - 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

Blitz code for Semaphore.wait

```
method Wait ()
   var oldIntStat: int
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait`
   operation")
   EndIf
   count = count – 1
   if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
```

But what is `currentThread.Sleep ()`?

- **If sleep stops a thread from executing, how, where, and when does it return?**
  - which thread enables interrupts following sleep?
  - the thread that called sleep shouldn't return until another thread has called signal !
  - … but how does that other thread get to run?
  - … where exactly does the thread switch occur?
- **Trace down through the Blitz code until you find a call to switch()**
  - Switch is called in one thread but returns in another!
  - See where registers are saved and restored

Look at the following Blitz source code

- **Thread.c**
  - Thread.Sleep ()
  - Run (nextThread)

- **Switch.s**
  - Switch (prevThread, nextThread)

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)

endMethod
```

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)

endMethod
```

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)

endMethod
```

Blitz code for Semaphore.signal

```
method Signal ()
   var oldIntStat: int
   t: ptr to Thread
   oldIntStat = SetInterruptsTo (DISABLED)
   if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during
        'Signal' operation")
   endIf
   count = count + 1
   if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
   endIf
   oldIntStat = SetInterruptsTo (oldIntStat)

endMethod
```

Semaphores using atomic instructions

- **Implementing semaphores with interrupt disabling only works on uni-processors**
  - What should we do on a multiprocessor?

- **As we saw earlier, hardware provides special atomic instructions for synchronization**
  - test and set lock (TSL)
  - compare and swap (CAS)
  - etc

- **Semaphore can be built using atomic instructions**
  1. build mutex locks from atomic instructions
  2. build semaphores from mutex locks

Building spinning mutex locks using TSL

```
Mutex_lock:
            TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
            CMP REGISTER,#0         | was mutex zero?
            JZE ok                  | if it was zero, mutex is unlocked, so return
            JMP mutex_lock          | try again
 Ok: RET                      | return to caller; enter critical section

 Mutex_unlock:
            MOVE MUTEX,#0           | store a 0 in mutex
            RET                     | return to caller
```

Using Mutex Locks to Build Semaphores

- How would you modify the Blitz code to do this?

What if you had a blocking mutex lock?

Problem: Implement a counting semaphore

Up ()

Down ()

...using just Mutex locks

- Goal: Make use of the mutex lock's blocking behavior rather than reimplementing it for the semaphore operations

How about this solution?

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

Down ():

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

Up():

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

How about this solution?

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

Down ():

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

Up():

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

Contains a Deadlock!

How about this solution then?

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

Down ():

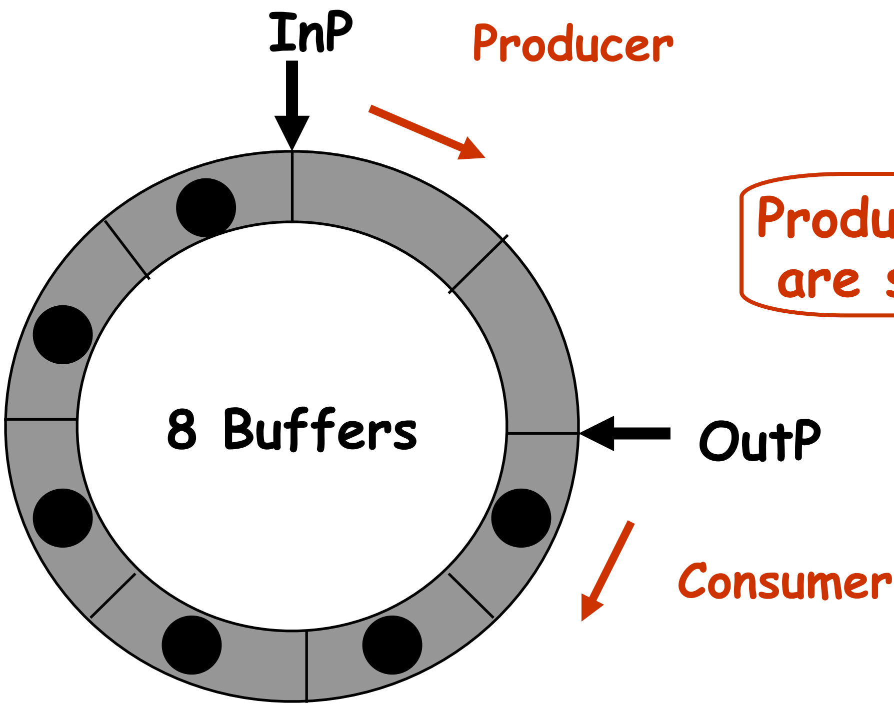
```
Lock(m1)
cnt = cnt - 1
if cnt<0
  Unlock(m1)
  Lock(m2)
else
  Unlock(m1)
endIf
```

Up():

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

Producer consumer problem

- Also known as the bounded buffer problem

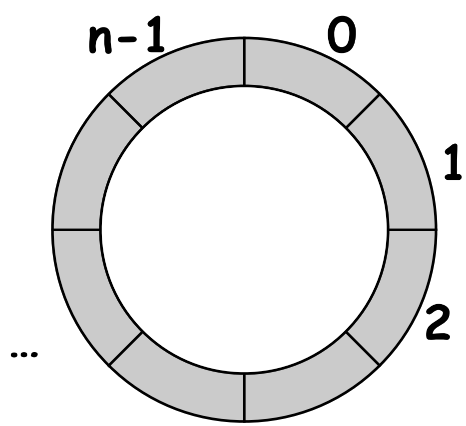


Producer and consumer are separate threads

Is this a valid solution?

```
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
```

Does this solution work?

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

```
thread producer {
  while(1){
    // Produce char c...
    down(empty_buffs)
    buf[InP] = c
    InP = InP + 1 mod n
    up(full_buffs)
  }
}
```

```
thread consumer {
  while(1){
    down(full_buffs)
    c = buf[OutP]
    OutP = OutP + 1 mod n
    up(empty_buffs)
    // Consume char...
  }
}
```

درست است؟

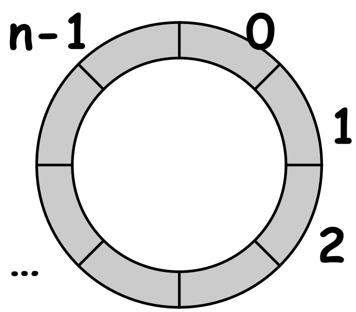
Use a mutex to fix the race condition in this code

```
thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
  }
}
```

wait(m)
signal(m)
wait(m)
signal(m)

```
thread consumer {
  while(1) {
    if (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}
```

wait(m)
signal(m)
wait(m)
signal(m)




```
Global variables:
    char buf[n]
    int InP = 0    // place to add
    int OutP = 0   // place to get
    int count
```

How about this solution then?

```
var cnt: int = n            -- Signal count
var m1: Mutex = unlocked -- 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)
```

How about this solution then?

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

۰- ابتدا n=۰

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)
```

How about this solution then?

```
var cnt: int = n             -- Signal count
var m1: Mutex = unlocked -- 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
```

۰- ابتدا n= ۰

۱- چند
ریسمان اینجا

Up():

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

How about this solution then?

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

Down ():

۰- ابتدا n=۰

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

۱- چند ریسمان اینجا

Up():

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

۲- چندین بار این صدا شود

How about this solution then?

```
var cnt: int = n            -- Signal count
var m1: Mutex = unlocked -- 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
```

۰- ابتدا n=۰

۱- چند ریسمان اینجا

Up ():

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

۲- چندین بار این صدا شود

چون هنوز Lock(m2) صدا نشده، نتیجه چندین بار باز کردن قفل باز نامعلوم است

پیاده‌سازی Semaphore با Mutex (روش ۱)

```
var
        mutex=1: binary-semaphore;
        delay=0: binary-semaphore;
        C={initvalue}: integer;
```

```
Procedure Wait()
        begin
        wait(mutex);
        C:=C-1;
        if C < 0 then begin
                signal(mutex);
                wait(delay);
                end
        signal(mutex);
        end
```

```
Procedure Signal()
        begin
        wait(mutex);
        C:=C+1;
        if C <= 0 then
                signal(delay)
        else
                signal(mutex)
        end
```

پیاده‌سازی Semaphore با Mutex (روش ۲)

```
var
        mutex=1: binary-semaphore;
        delay=0: binary-semaphore;
        barrier=1: binary-semaphore;
        C={initvalue}: integer;
```

```
Procedure Wait()
        begin
        wait(barrier);
        wait(mutex);
        C:=C-1;
        if C < 0 then begin
                signal(mutex);
                wait(delay);
                end
        else
                signal(mutex);
        signal(barrier);
        end
```

```
Procedure Signal()
        begin
        wait(mutex);
        C:=C+1;
        if C = 1 then
                signal(delay)
        signal(mutex)
        end
```

پیاده‌سازی Semaphore با Mutex (روش ۳)

```
var
        mutex=1: binary-semaphore;
        delay={min(1,initvalue)}: binary-semaphore;
        C={initvalue}: integer;
```

```
Procedure Wait()
        begin
        wait(delay);
        wait(mutex);
        C:=C-1;
        if C > 0 then
                signal(delay);
        signal(mutex);
        end
```

```
Procedure Signal()
        begin
        wait(mutex);
        C:=C+1;
        if C = 1 then
                signal(delay)
        signal(mutex)
        end
```