

2 IPC SYSTEM INVOLVED MUTUAL EXCLUSION AND SYNCHRONIZATION MECHANISMS- DISABLING INTERRUPTS, SEMAPHORS, MONITORS

3 SYSTEM INVOLVED MUTUAL EXCLUSION AND SYNCHRONIZATION MECHANISMS- PIPES, FILES, MESSAGE PASSING, SIGNALS

System Implementation of Semaphors, Binary and Counting and their use for Mutual Exclusion and Synchronization

4 SEMAPHOR: DOWN AND UP SURROUNDING CRITICAL REGIONS GIVES MUTUAL EXCLUSION -- TWO FORMS OF SYNCHRONIZATION

5 SEMAPHORS: BINARY MUTUAL EXCLUSION SEMAPHOR TRACE

6 SYNCHRONIZATION WITH BINARY SEMAPHORS

7 COUNTING SEMAPHORS: SEMAPHOR TRACE

8 COUNTING SEMAPHORS (n) CONSTRUCTED WITH BINARY SEMAPHORS(x, y)

Tracing the interaction of Processes and System with use of down and up system calls amongst other system calls

9 BINARY SEMAPHOR-TRACE TYPE 1

10 BINARY SEMAPHOR-TRACE TYPE 2

Alternative implementations of Producer-Consumer problem, SEMAPHORS-COUNTING & BINARY, BINARY and TSL

11 PRODUCER-CONSUMER WITH COUNTING SEMAPHORS

12 PRODUCER-CONSUMER WITH BINARY SEMAPHORS AND WITH TSL 1

13 PRODUCER-CONSUMER WITH TSL 2 WITH FEWER WRITES TO lock

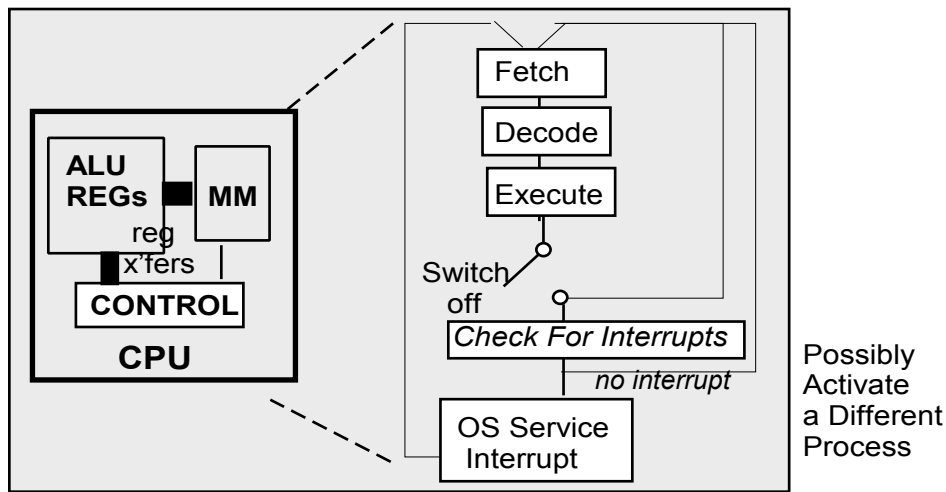
Barrier-ss-Synchronized starts of multiple Processes

14 GENERAL BARRIER ACHIEVED WITH SEMAPHORS (Monitor Preview)

15 2 and 3 Process BARRIER ACHIEVED WITH SEMAPHORS

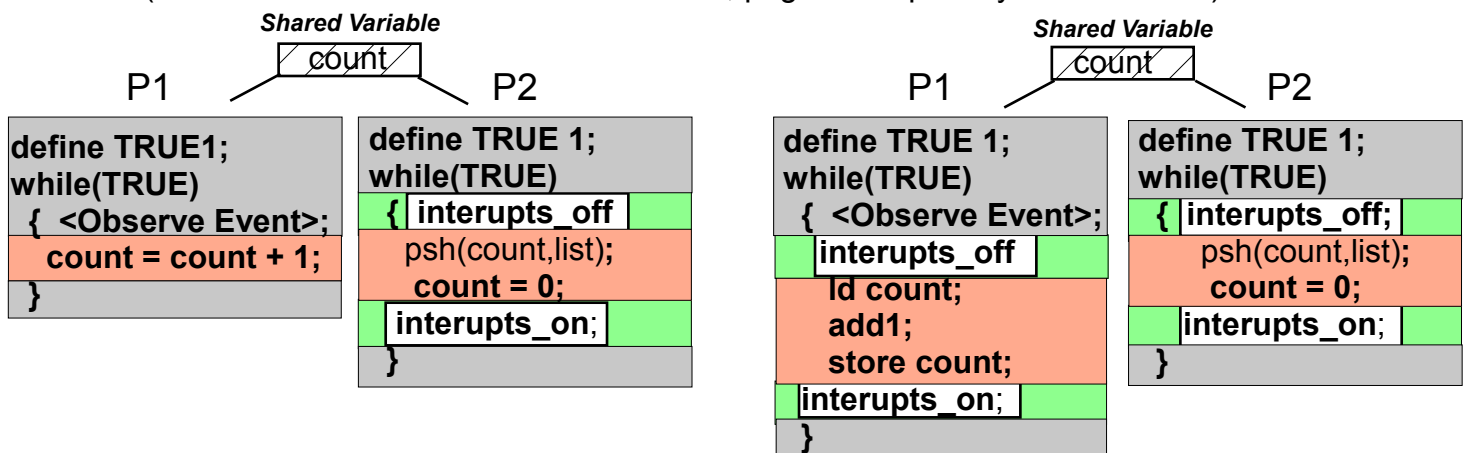
CONTENTS

Disabling Interrupts



A Process running in the active state may leave that state for many reasons. Those reasons are often initiated by an interrupt, that doesn't have to be handled immediately-non-urgent events, ex. Quanta or I/O completed, etc. There are also reasons for leaving the active state, such as page faults, read and/or write call which require more immediate attention-urgent events. If interrupts are disabled by the Kernel for an active Process it will continue to run in all cases. If we can be assured that no urgent events caused the interrupt, then the Process can remain active until the interrupts are turned on again. So a simple way to ensure that a running Process, will run through a (Critical) region without interference, i.e., atomically, is to **turn off the interrupt mechanism**. As long as read and or write, page fault or similar commands are not in the Critical Region mutual exclusion is insured. This method is not generally available to applications-but is used within OS. works if interrupts are turned off on all of the involved Processors.

EXAMPLE: (Refer to Non-OS Race Avoidance Notes, page2- Simple Asymmetric Case)



SEMAPHORS

Two system calls are involved, here named **down(m)** and **up(m)** and. In a Process the **down(m)** is placed before, and **up(m)** after a Critical Region whose execution is to be mutually exclusive to all Regions in other Processes similarly surrounded by **down(m)** and **up(m)**. So access to code sections delineated by **down(m)** and **up(m)**. in different Processes is atomic with respect to sections in other Processes also delineated by **down(m)** and **up(m)**.

Also the **down(m)** can be used to block a Process anywhere. That Process will only be unblocked when another Process does a **up(m)**. It can thus Synchronize Processes. (see Page 4)

MONITOR

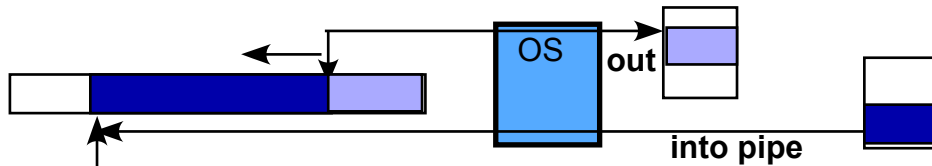
A Monitor is a higher level construct which allows Mutual Exclusion amongst a group of procedures as well as Synchronizattion using **wait and signal** commands. This higher level code then is translated into code with appropriate system calls to implement this code: Semaphor calls, for example,

IPC-SYSTEM INVOLVED MUTUAL EXCLUSION AND SYNCHRONIZATION MECHANISMS-DISABLING INTERRUPTS, SEMAPHORS, MONITORS

PIPES and FILES

Mutual exclusive Access To Shared Memory Achieved by Passing All Requests (read writes) from Involved Processes to be Executed by a ANOTHER (THIRD PARTY) Single Process (usually the kernel):

In general pipes like mailboxes are circular buffers through which communication between two Processes is established. When a pipe is created it is given a fixed size, (~4k bytes). A write of any length, or read request from any Process is executed atomically, i.e., **mutually exclusively**. If a pipe write will overflow the pipe, or a read finds an empty pipe the Process will block or optionally returns a value which indicates whether it worked or not. **It behaves as one might expect a File** with reads and writes behaves of the same form, but unlike a File the data is stored in MM in the kernel-making access to it faster and the restricted point of access for reading in and writing out. Like reading and writing Files the required mutual exclusion of reads and writes is achieved by having one Process, in this case the Kernel (OS), getting all such requests from different Processes and implement one at a time. (This is not generally a safe assumption for Files in UNIX.)



When IPC is done using messages or pipes the information is read out in the order it was written. Messages are read out and written in fixed size units. Pipes are read out and written in any size consistent with the maximum contents of the pipe.

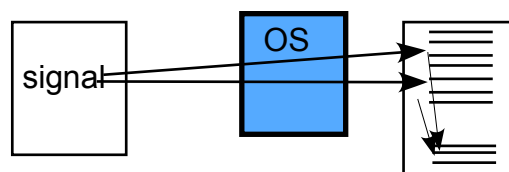
In these cases access to specified memory is atomic with respect to other accesses to the same specified memory

MESSAGE PASSING (SYNCHRONOUS)

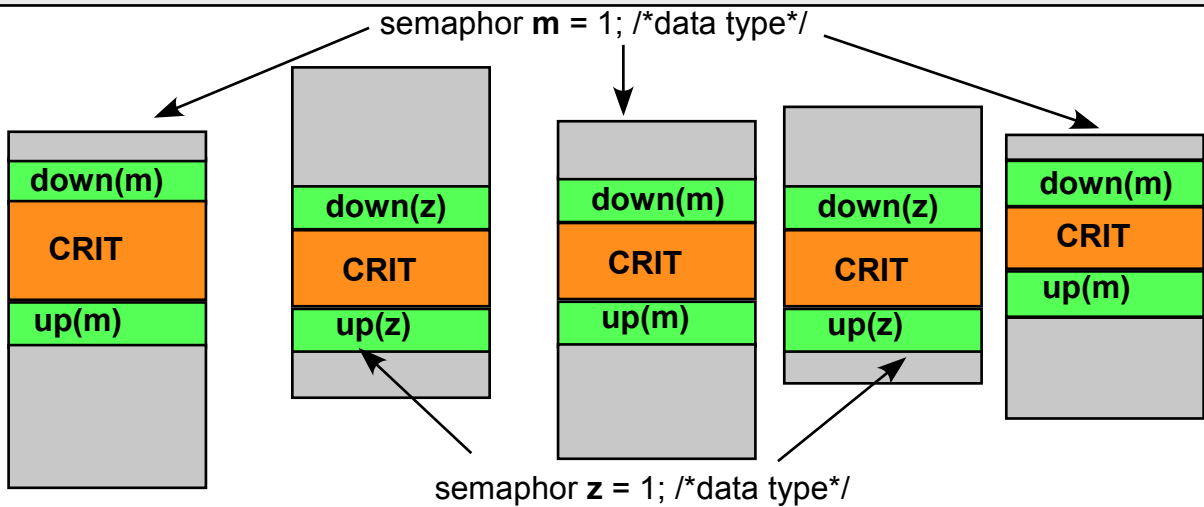
The system message passing facility incorporates a blocking procedure. It can be used to pass information between processes and thereby accomplish any protected communication between processes that can be accomplished with explicit shared memory combined with the mutual exclusion and synchronization mechanism provided by semaphores. This becomes the main mode of communication between remote distributed Processors or from and to a Micro-Kernel. It is designated **synchronous** the actual sending and reception occur at the sites of send and receive commands in communicating Processes-fixed places within the Processes.

SIGNALS (ASYNCHRONOUS)

Signals provide a number of specific messages, ex. *exit*, which can cause preset action by the receiving Process. The receiving Process has the option of ignoring any of these messages or executing a self defined function on receiving a signal. Since a signal may interrupt a Process at any point in its execution it is an **asynchronous** form of signalling, acting like Exceptions in programming languages. All the other forms of communications we have outlined, being read and written by specific commands located at specific position in the Process code are synchronous.



SYSTEM INVOLVED MUTUAL EXCLUSION AND SYNCHRONIZATION MECHANISMS- PIPES, FILES, MESSAGE PASSING, SIGNALS



In the text (and here):

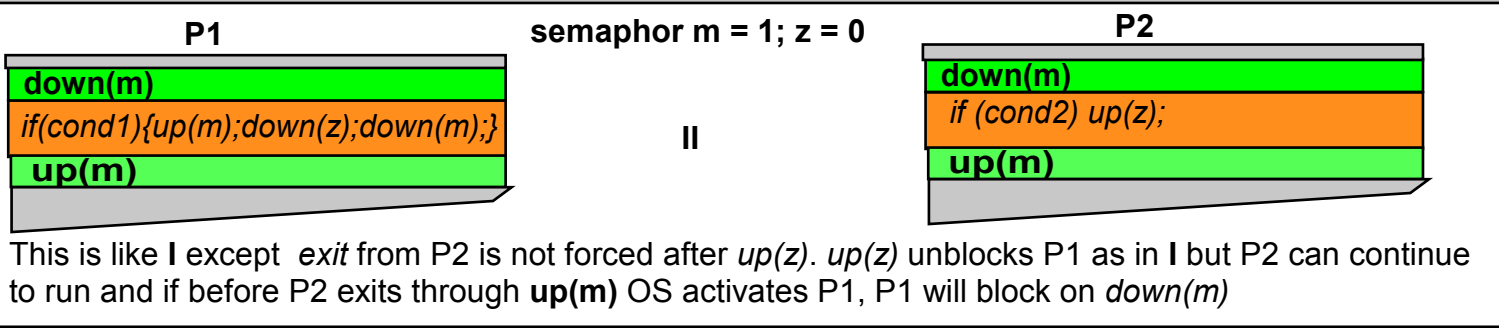
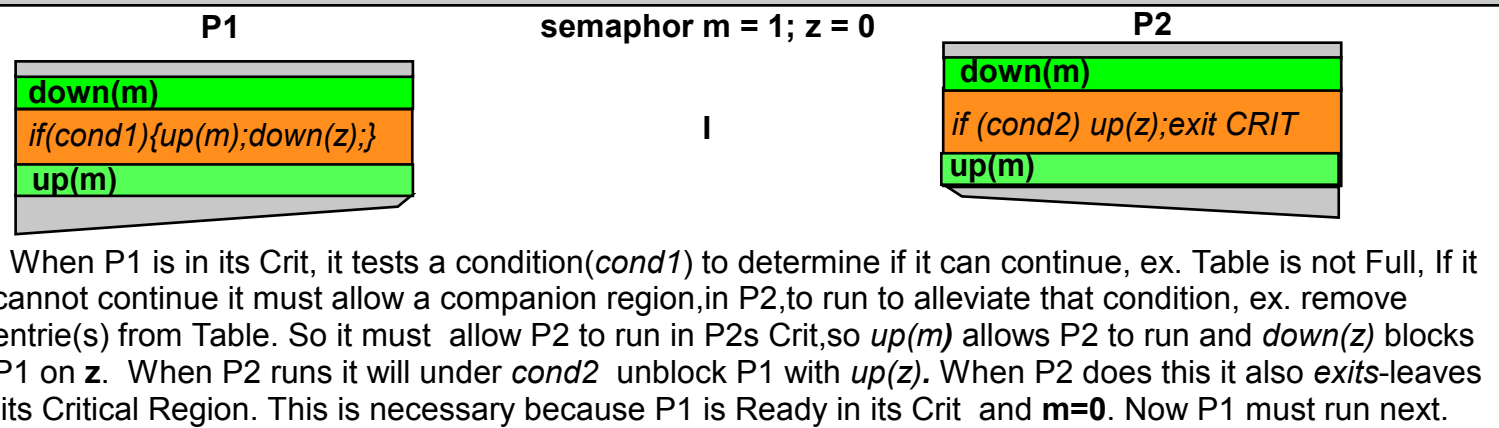
A **semaphor** is a special data type which can only be used as the argument of the system calls: **down** and **up**. It can be initialized. It cannot be used in any arithmetic operations. In these notes a variable, (**m**, in the example above), is declared to be a semaphor for example, as follows:
`semaphor m = 1; /*data type*/` (In UNIX much more is required to establish a semaphor)

For mutual exclusion typically the initial value of **m** is ≥ 0 , and is assigned by the user- changes in value are assigned by the OS in response to the downs and ups used by the programmer.

When the same semaphor, say **m**, appears in two processes it implies that they are in communication. By using the same semaphor they are allowing the **down(m)** and **up(m)** calls to determine blocking and unblocking in each others Processes.

A **down(m)** with $m = 0$ in **P** will cause **P** to block, while $m > 0$ will cause **P** to continue.

MUTUAL EXCLUSION: EX. 3 CRITs are Protected with semaphor m and 2 with semaphor z



TWO KINDS OF SYNCHRONIZATION

SEMAPHOR: DOWN AND UP SURROUNDING CRITICAL REGIONS GIVES MUTUAL EXCLUSION -- TWO FORMS OF SYNCHRONIZATION

semaphor $m = n$ ($n = 0$ or 1); /*data type*/

down(m): [In Process P] OS Action

```

if ( m==1)
{ m=0; continue to L [in P] or (P -> Ready
  at L [1st location after down];)}
else [m must == 0]
{ P is Blocked, Waiting on m; its registers
  and return address L are saved; someone
  else is made Active) }
  
```



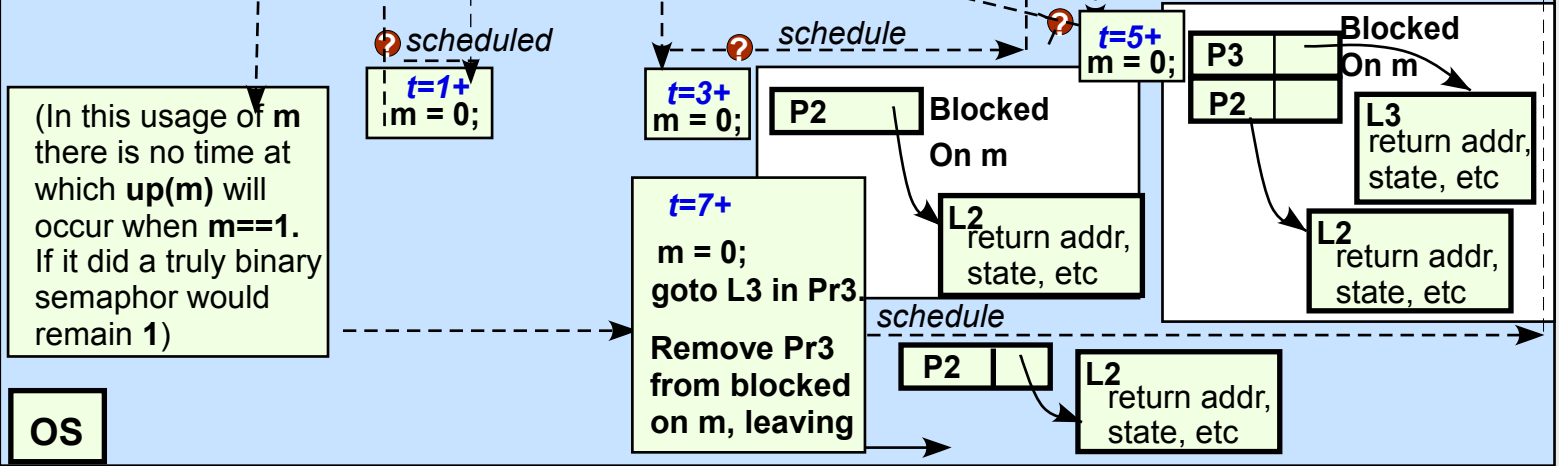
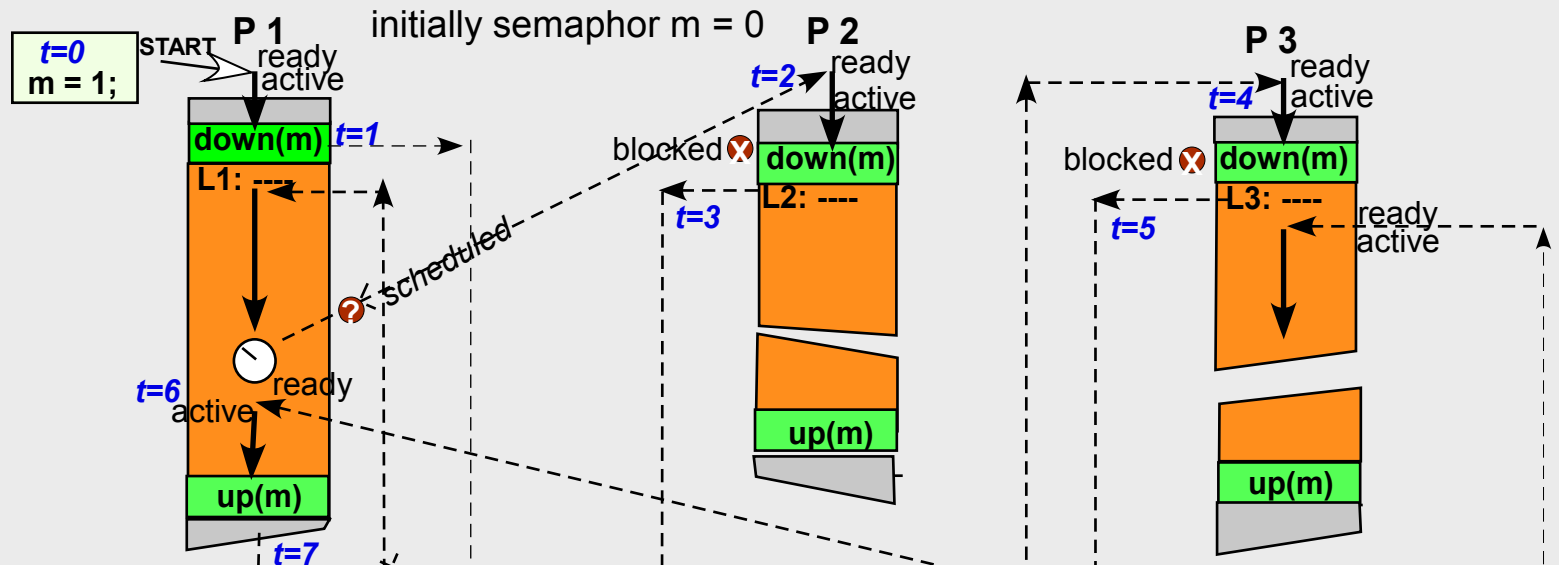
up(m): [In Process P] OS Action

```

if ( m==1) continue in P
if ( m==0 && No process is Waiting on m)
{ m=1; continue }
else
{ switch a Process from Blocked Waiting on
  m; to Ready. (Make a Ready Process
  Active.) } [m remains 0]
  
```

MUTUAL EXCLUSION WITH BINARY SEMAPHORS

System actions on receipt of system calls $down(m)$ and $up(m)$



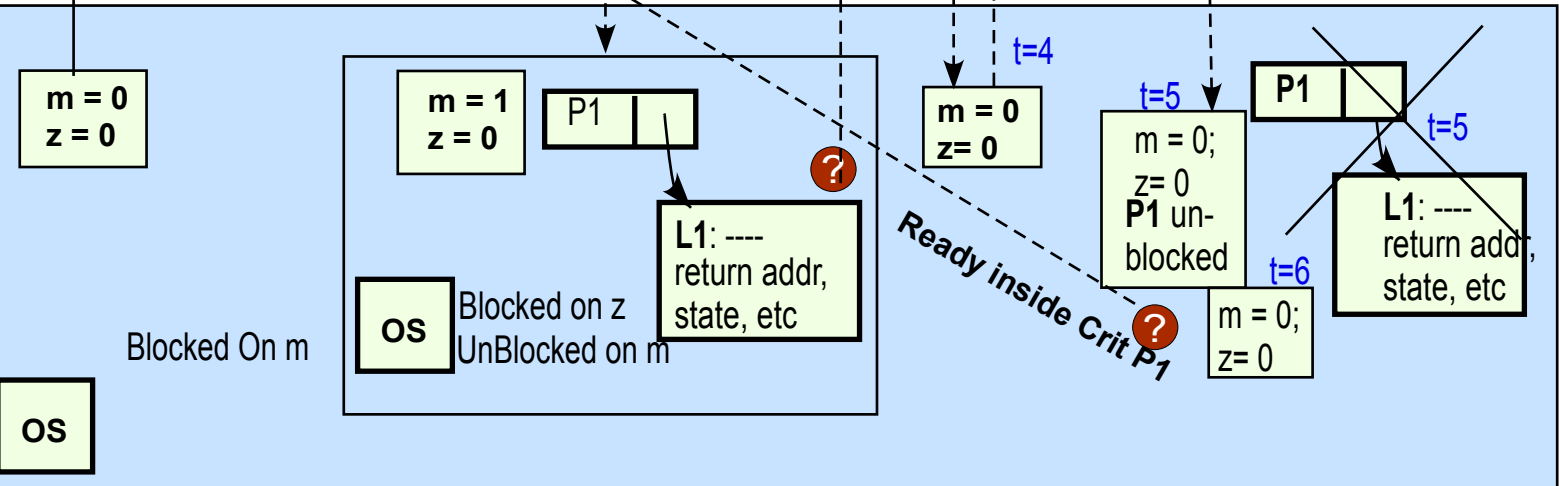
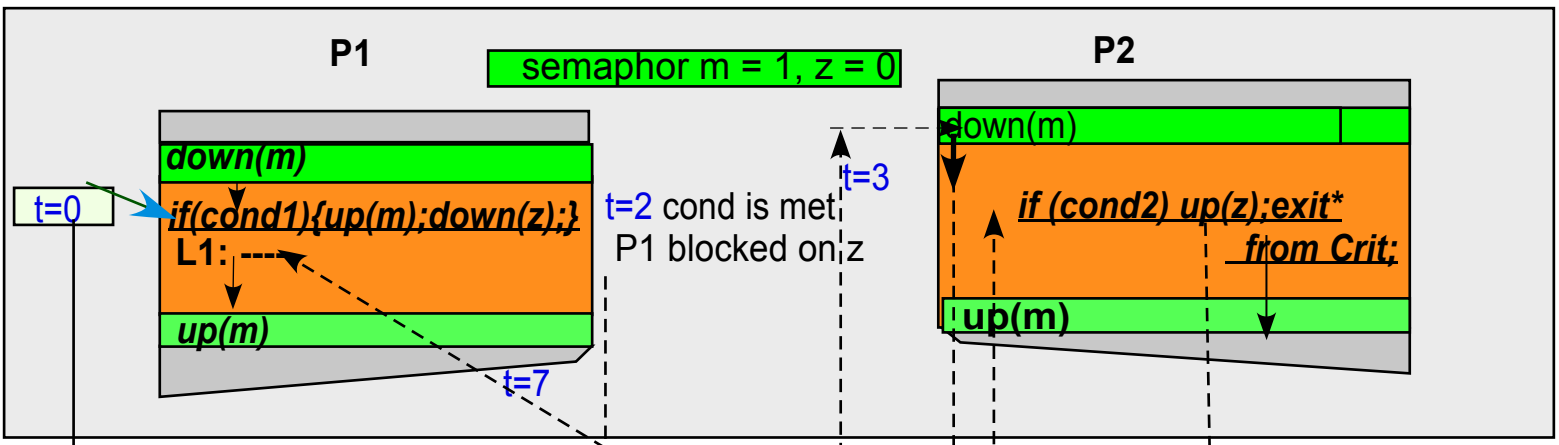
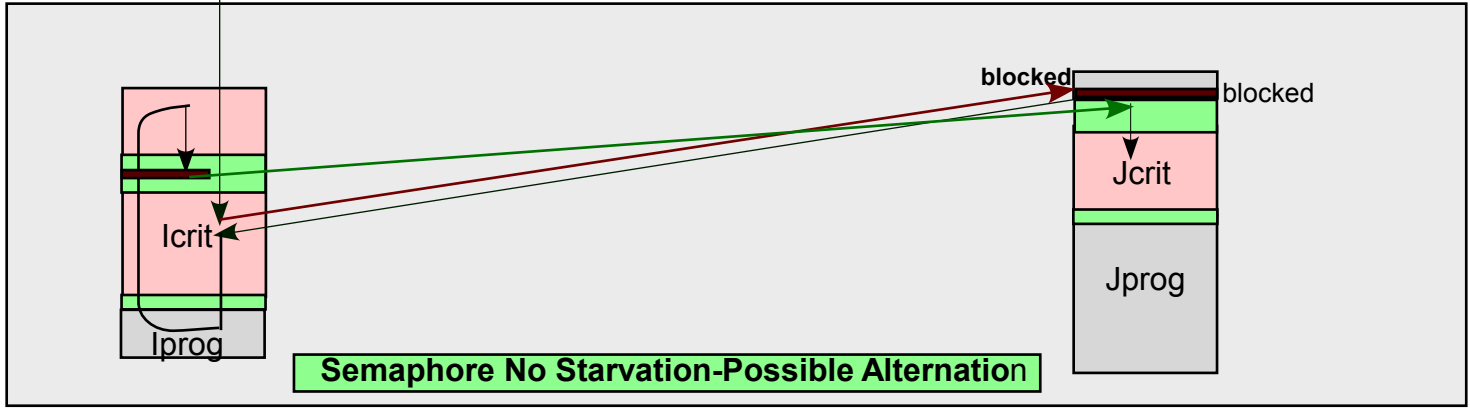
(In this usage of m there is no time at which $up(m)$ will occur when $m=1$. If it did a truly binary semaphore would remain 1)

OS

For each semaphore one could keep the list of all Processes blocked on that semaphore so as to easily determine when an up occurs which Process if any should be made ready next. if no Process is blocked on a given semaphore no list is required. This is similar to the Ready list of all Processes which are ready to run so that the scheduler can choose one when necessary.

$up(m)$ and, If $m > 0$, $down(m)$, does not necessarily require a full context switch.

SEMAPHORS: BINARY MUTUAL EXCLUSION SEMAPHOR TRACE



Synchronization involves blocking in the Critical Region of one Process, Pa on a condition (ex. Table Full) which can be relieved by another Process, Pb, when it affects the disappearance of that condition (ex. Removes an entry from Table).

Assume $m=1$ initially. P1 is entered- $m=0$ at $t=0$. Continuing with $m=0$, the expression $if(cond1)\{up(m);down(z);\}$ is encountered. So, assuming $cond1$ is true, m goes to 1, and P1 blocks on z (since it is initially 0). The OS keeps the state of P1 and that it is blocked on z . Later P2 can enter its CRIT, through $m=1$. m becomes 0. Now, assuming $cond2$ in $if(cond2)\{up(z);\}$ is true the $up(z)$ unblocks P1 (now in ready state) but unblocked z is still 0 because P1 is recorded as blocked on z). P2 exits and P1 is Ready again in its Crit with $m=0$ so no one else can enter a Crit protected by m until P1 exits through $up(m)$.

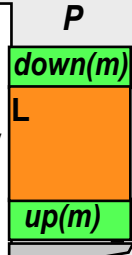
SYNCHRONIZATION WITH BINARY SEMAPHORS

semaphor $m = n$ ($n \geq 0$); /*data type*/

down(m): [In Process P] OS Action

```

if ( m >= 1)
{ m = m - 1; continue to L [in P] (or P -> Ready
  at L [1st location after down];)}
else
{if( m == 0)
{ P is Blocked. Waiting on m; its registers
  and return address L are saved; someone
  else is made Active }
}
[m remains 0]
  
```

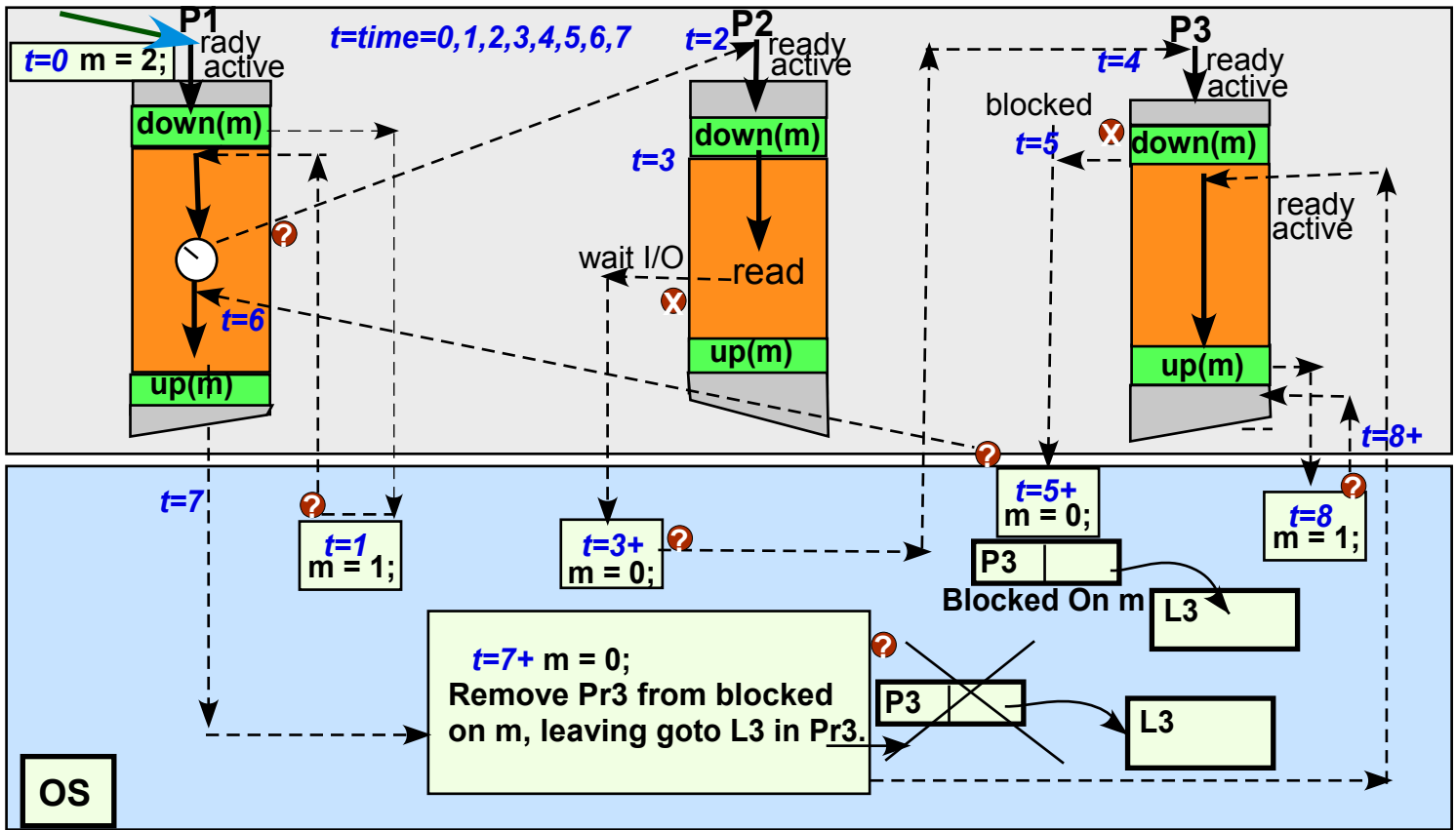


up(m): [In Process P] OS Action

```

if ( m > 0) m = m + 1;
else
{if ( m == 0 && no Process is Waiting on m)
{ m = 1; continue }
else
{ switch a Process from Blocked Waiting on
  m;
  to Ready. (Make a Ready Process Active.) }
}
[m remains 0]
  
```

MUTUAL EXCLUSION WITH COUNTING SEMAPHORS
System actions on receipt of system calls *down(m)* and *up(m)*



With Counting Semaphors initialized to m and used as above, $m-1$ can be in their CRITs together

COUNTING SEMAPHORS: SEMAPHOR TRACE

semaphor $x = 1$ /*initially unblocked*/, $y = 0$; /*initially blocked it will serve to block Process if $n=0$ */
 int shared $n = N$; blocked = 0; /*the integer variable n serves as the count for the **counting** semaphor*/

DOWN (n,x,y)

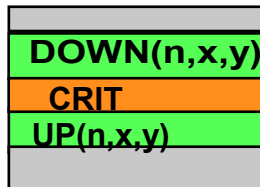
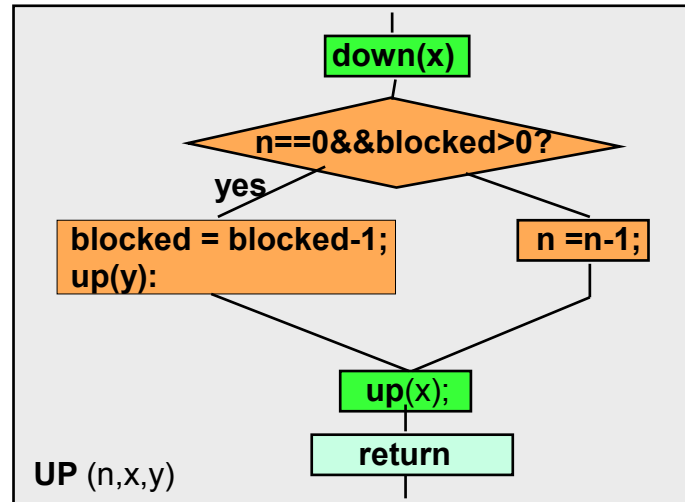
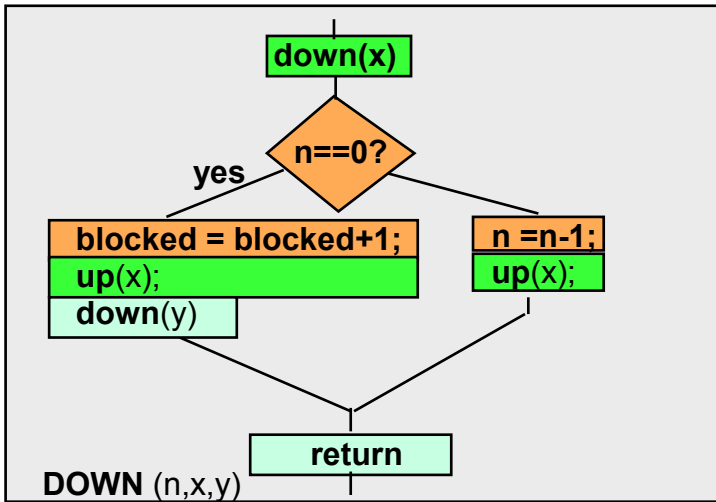
```

{
  down(x); (1)
  if (n==0) {blocked=blocked+1; up(x); down(y); } /*if n == 0 block on y (n is still ==0) */ (3a)
  else {n = n - 1; up(x); } /*if n != 0 */ (2)
}
    
```

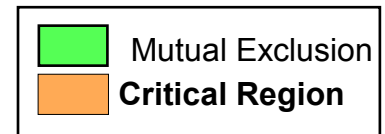
UP (n,x,y)

```

{
  down(x); (1)
  if (n==0 && blocked > 0) {blocked = blocked-1; up(y);} /*if n == 0 && >= 1 proc blocked on (3b)
  else {n = n + 1;} y then unblock 1 of them proc */ (2)
  up(x); /*if n != 0 */
}
    
```



USE

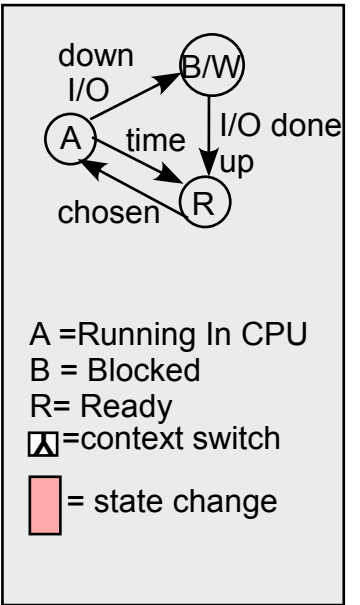
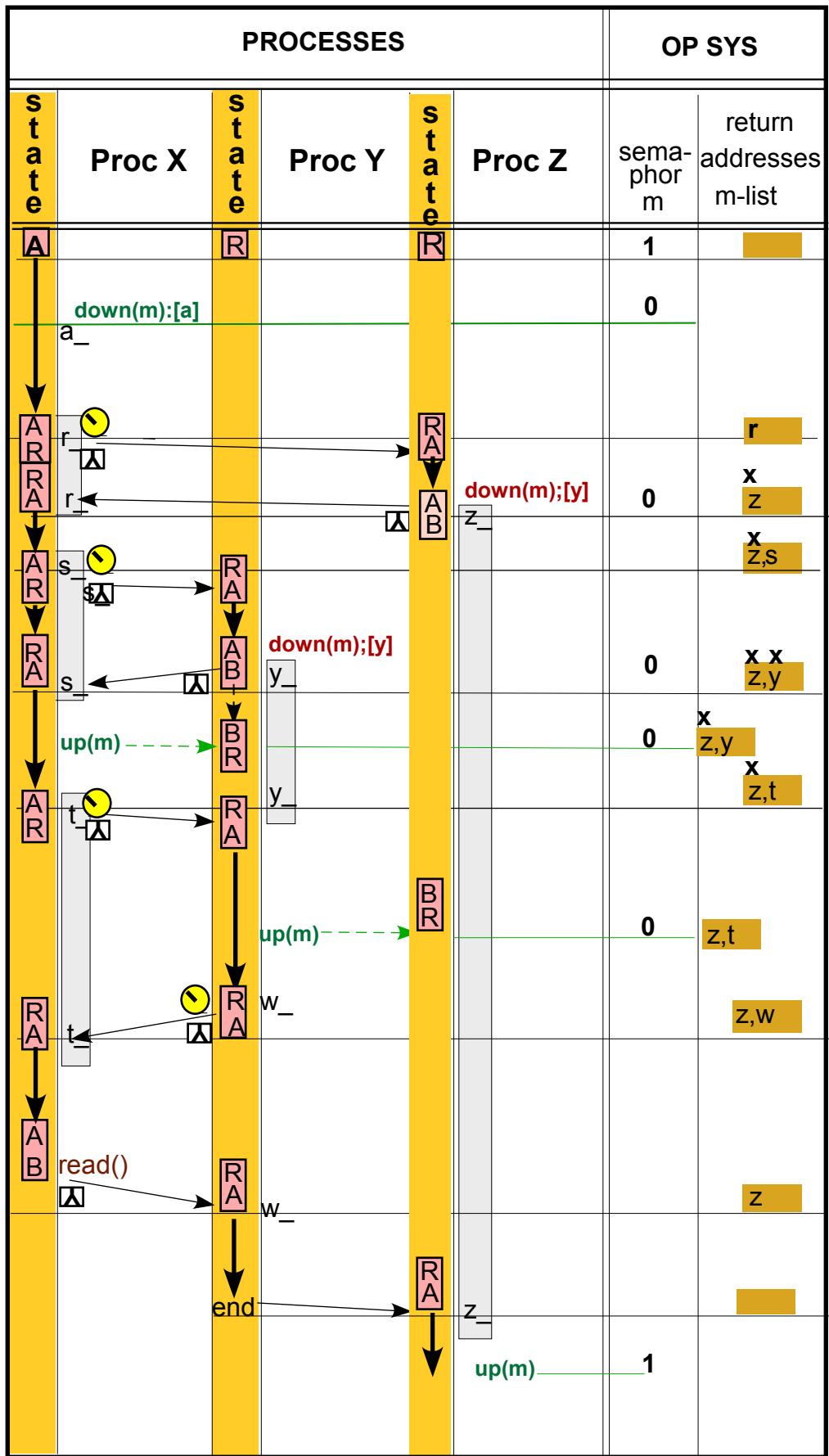


For each counting semaphor to be simulated a different variable n , and semaphors x , and y are necessary.

- (1) The procedures **DOWN** and **UP** both refer to shared variables “blocked” and “ n ” in such a way that mutually exclusive is needed. This is accomplished by surrounding each with **down(x) and up(x)**.
- (2) If test “ $n=0$ ” is false, that is, if $n > 0$ then: **DOWN** just subtracts 1 from n , and **UP** just adds 1 to n .
- (3) If the test “ $n=0$ ” is true then
 - (3a) **DOWN** must block. It blocks on y (Initially = 0) by calling down(y). In doing so it must remove protection by semaphor x by generating **up(x)**. Furthermore it is necessary to keep track of how many processes are blocked on semaphor y this is accomplished adding 1 to shared variable “blocked”. Note for each successive **down(y)** blocked with $n=0$, the OS records that the process blocked and its state, return address, etc. for use when an **up(y)** occurs with $y=0$. Also the processes block on y outside the Critical Region delimited by semaphor x
 - (3b) **UP** must determine if anyone is blocked on y . If no: then it simply subtracts 1 from n . If yes: it subtracts 1 from blocked and **calls up(y)**. As a result **OS will find a process, say Q, blocked on y and put it in the Ready state just beyond DOWN (n,x,y)**.

Assume that **P1** blocks on **DOWN**(n,x,y), then **P2** calls **UP**(n,x,y). This arrangement allows **P2** to run **UP**(n,x,y) again even after run it and released **DOWN**(n,x,y) from being blocked on **down(y)** by issuing an **up(y)** and thus allowing **P1** also to run.

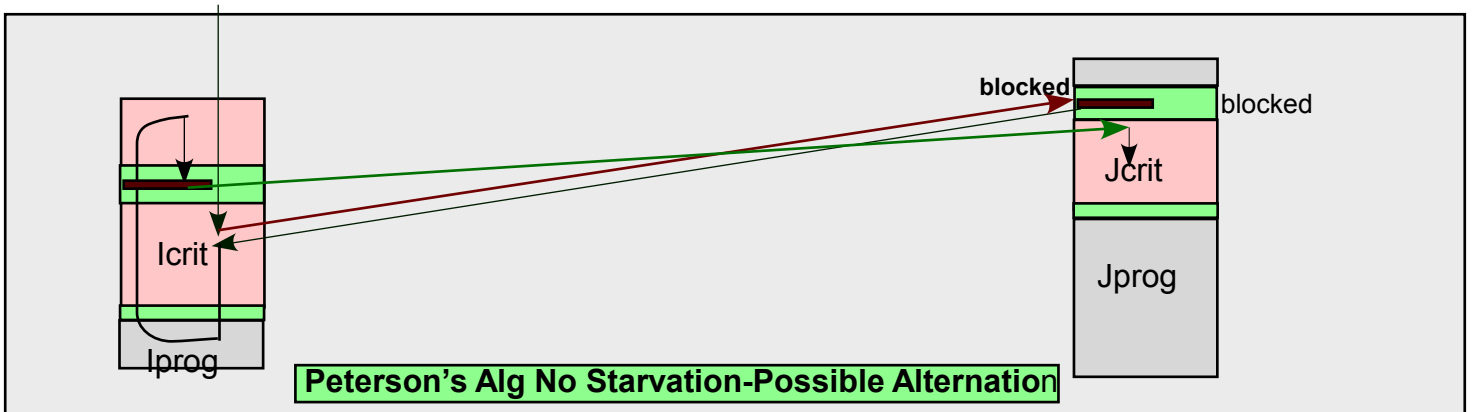
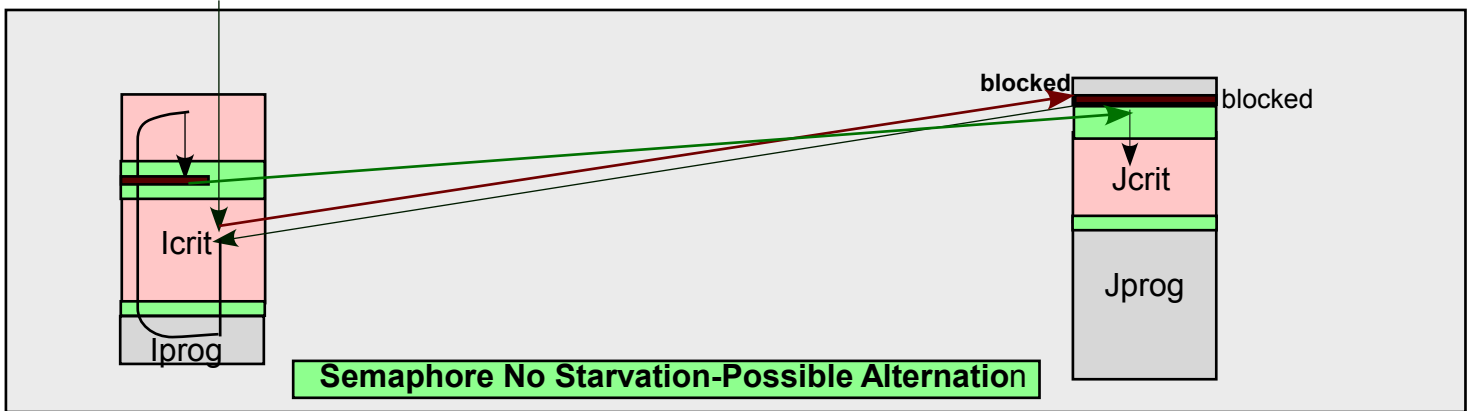
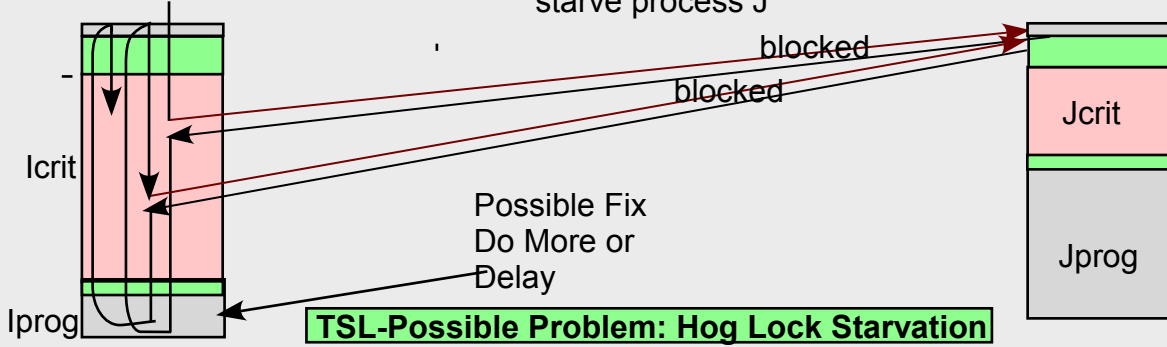
COUNTING SEMAPHORS (n) CONSTRUCTED WITH BINARY SEMAPHORS(x, y)



On up unblock the most recent to be blocked

BINARY SEMAPHOR-TRACE 1

If Iprog is short Process I may run many times before a waiting Process J can run in fact it could starve process J



Possible Problems Of Different Mutual Exclusion Implementations

```
#define N 27
semaphor mutex = 1;
semaphor empty = N;
semaphor full = 0;
int shared item(N);
```

```
void producer(void)
```

```
{ int item;
  while(TRUE) PRODUCER
```

```
{ produce_item(&item);
  (down(&empty) /* if (empty > 0) empty = empty - 1, else BLOCK */
  down(&mutex);
  enter_item(item); [pointer:shared_i?]
  up(&mutex)
  up(&full) /*(Only after its definitely in) if any process is
            BLOCKED on semaphor full UNBLOCK one,
            otherwise full = full+1 */
}
```

```
void consumer(void)
```

```
{ int item;
  while(TRUE) CONSUMER
```

```
{ down(&full) /* if(full > 0 ) full = full - 1, else block */
  C: down(&mutex)
  remove_item(&item);
  up(& mutex);
  up(&empty) /*(Only after definitely out) if any process is
            BLOCKED on semaphor empty UNBLOCK one,
            otherwise empty = empty+1 */
  consume_item(item);
}
```

“(#)empty” contains a count of the number of buffer entries that are empty (available) ((#)empty). Initially N, Generally (#)empty, is checked by PRODUCER (down(empty) PRODUCER blocks only if 0 Only after the CONSUMER has removed an entry does it increment ((#)empty), up(&empty), making another space in table available to PRODUCER.

“(#)full” contains a count of the number of buffer entries that are occupied (#full) Initially 0 (#)full, it is checked by CONSUMER (down(full) CONSUMER blocks only if 0. Only after the PRODUCER has completed an entry does it increments ((#)full),up(&full) so CONSUMER has another entry to consume.

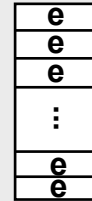
“mutex” is a binary semaphore. It makes access to the buffer by PRODUCER and CONSUMER mutually exclusive.

As long as neither Producer or Consumer is blocked nor Active(#)empty + (#)full = N. But if Producer is blocked on down((#)empty) (#)empty + #full = N-1or if Consumer is blocked on down((#)full) then also (#)empty + #full = N-1 is possible.

What happens with multiple Producers and Consumers?

PRODUCER-CONSUMER WITH COUNTING SEMAPHORS

time0
CONSUMER Goes

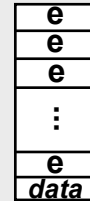


(#)empty = N
(#)full = 0
CONSUMER
Blocked on #full = 0

time1

PRODUCER Goes
on empty > 0

OS: (#)empty = N - 1

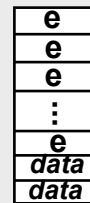


(#)empty = N-1
(#)full = 0 but
unblocked CONSUMER@ C:

time2

PRODUCER Goes
on (#)empty > 0

OS:(#)empty=(#)empty-1==9
DONE:(#)full=(#)full + 1==1

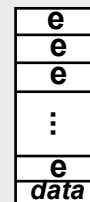


(#)empty = N-2
(#)full = 1 unblock
Consumer will go

time3

CONSUMER Goes

DONE: (#)empty=(#)empty+1



(#)empty = N-1
(#)full = 1

```
#define N 27
semaphor x = 1, y=0 /*x and y initially 1, unblocked*/,
int shared n = N; /*the integer variable n serves to count entries in buffer*/
int shared buffer;
```

```
while(TRUE)
    PRODUCER
    {
        produce_item(&item);
        down(x);
        if (n = N){up(x);down(y);} /*if full block on y*/
        enter_item(&item,buffer);
        n = n + 1;
        if (n == 1 && cons ==1) {cons = 0;up(y); continue}
        /*if n was 0 and blocked on y (cons), unblock it on y*/
        up(x);
    }
```

```
while(TRUE)
    CONSUMER
    {
        down(x);
        if (n ==0){cons =1; up(x);down(y);} /*if empty block on y*/
        remove_item(&item,buffer);
        n = n - 1;
        if (n = N-1) up(y);continue; /*if n= N -Prod blocked on y, unblock y*/
        up(x);
        consume_item(&item);
    }
```

cons is necessary for the case that n = 0 initially and PRODUCER was called and therefore an up(y) would result though there had never been a down(y).

```
#define N 27
int shared lock0 = 0, lock1=1, lock2=1 /*initially unlocked*/,
int shared n = N;
int shared buffer;
```

```
while(TRUE)
    PRODUCER
    {
        produce(item)
        Test(lock0, l1)
        if(n==N){prod=1; lock0=0; Test(lock1,l1)}
        enter_item1();
        n=n+ 1;
        if n==1&&cons==1){cons=0; lock2=0;}
        lock0=0;
    }
```

```
while(TRUE)
    CONSUMER
    {
        Test(lock0, l2)
        if(count==0){cons=1; lock0=0; Test(lock2, l2)}
        remove_item1(item);
        n=n - 1;
        if(count==N-1&&prods==1 ){prods=0;lock1=0;}
        lock0=0;
        consume(item)
    }
```

```
Test(lck,loc)
{ X tsl(lck,loc);
  cmp loc;
  jnz X;
```

Equivalent 3 lock TSL Solution

PRODUCER-CONSUMER WITH BINARY SEMAPHORS AND WITH TSL 1

```

Test(lock,loc)
{ X tsl(lock,loc); /* requires a write to cache,
  cmp loc;          perhaps a write back */
  jnz X;}

```

```

#define N 27
int shared lock = 0 /*initially unlocked*/,
int shared n = N; /*the integer variable n serves to count */
int shared buffer;

```

while(TRUE) **PRODUCER**

```

{
  produce_item(&item);
  T Test(lock, t1); /*protect n and buffer
  if (n = N ){lock = 0; jmp T;} /*if full open lock -time*/
  enter_item(&item,buffer);
  n = n + 1;
  lock = 0;
}

```

Synchronization

CONSUMER

```

while(TRUE)
{
  T Test(lock, t2); /*protect n and buffer*/
  if (n = 0 ){lock = 0; jmp T;} /*if empty open lock-time*/
  remove_item(&item,buffer);
  n = n - 1;
  lock = 0;
  consume_item(&item);
}

```

Synchronization

** This is a loop which runs for a fixed time but does nothing*

***The purpose of while(lock==1) is to decrease the number of writes by the tsl. A system with a cache may require write back to MM every time the tsl is executed in a waiting loop. In a multiprocessor system the change in one local cache by a write, requires changes in all other caches also using the written location.*

tsl behaves very much like a binary semaphore. Instead of the OS *Remembering* the Processes *waiting* in the blocked state on the binary semaphore and moving one from the *blocked* on the binary semaphore to the *Ready state* when an **up** occurs on that semaphore, all *waiting* (locked-out) processes are *BusyWaiting* and are in ready stateso when **lock = 0** occurs the next one in *busy waiting to be scheduled* will run.

**PRODUCER-CONSUMER WITH TSL 2
(DECREASING THE NUMBER OF WRITES TO LOCK)**

Barrier: All Processes Must End Before Any Restart

TOY EXAMPLE Problem Requiring Barrier

A sequence of 8 states s_0-s_7 , each 0 or 1, called S_0+ is transformed into a sequence S_1 , a small change gives S_1+ then again S_1+ is transformed by the same procedure to sequence S_2 , etc. The transformation is given by the function: $S_j = S_{j+1} \oplus S_{j-1}$

The transformations are carried out by a sequence of 7 Processes. Each process reads its two neighbor states in S_0 and puts its new state into S_1 . S_1 is not to be read until all processes have put their new entry into it. (rendevous). only 7 states result so S_1 is then shifted with 0 added at the front as s_0 , giving S_1+ . will be read and new entries placed in Copy 0, etc.

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	
S_0+	0	1	1	0	0	1	1	1	states
		P1	P2	P3	P4	P5	P6	P7	Processes
S_1		1	0	1	0	1	0	0	
S_1+	0	1	0	1	0	1	0	0	states
		P1	P2	P3	P4	P5	P6	P7	Processes
S_2		1	1	1	1	1	1	0	

```
#define N 10
int n=0;
semaphor mutex = 1;
semaphor barrier = 0;
while(TRUE)
```

```
{
    |
    CRIT i
}
```

```
down(mutex);
if (n == N-1)
{ while(n>0) {up(barrier); n=n-1; } up(mutex);}
else {n=n+1; up(mutex); down(barrier);}
}
```

**Process i for
General Semaphore Solution**

```
while(TRUE)
```

```
{
    |
    CRIT i
    procedure_k;
}
```

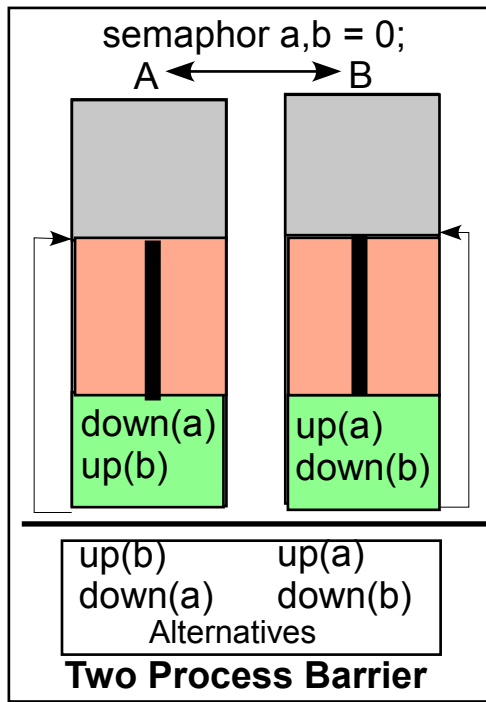
MONITOR

```
procedure(proc_k) [k=1 to N]
if (n == N-1)
{ while(n>0)
{ n=n-1;
signal(barrier)}
}
else
{ n=n+1;
wait(barrier);
}
}
```

**Process i and
Monitor for Monitor Solution**

- 1) All Processes run through their **CRITs**.
- 2) As each one completes its **CRIT** it test **n** for **N-1** after **down(mutex)**. if not it increments **n** and does **up(mutex)** thus allowing another process to make this test. (Because of the **down** and **up** on **mutex** only one process at a time makes this test). It then blocks on **down(barrier)**.
- 3) Finally, after the last CRIT is done $n = N-1$ is true, say in process P. In P a while loop generates a succession of **up(barrier)s**, unblocking the $N-1$ processes blocked on semaphor barrier, putting them in the Ready state so they can again run through their CRITs. When they complete their CRITs they are blocked on mutex as long as P is generating up(barriers).
- 4) When P has completed all its up(barriers) it does an up(mutex) thus enabling other processes after finishing their CRITs to again test for $n = N-1$, and also allowing P to enter its CRIT.

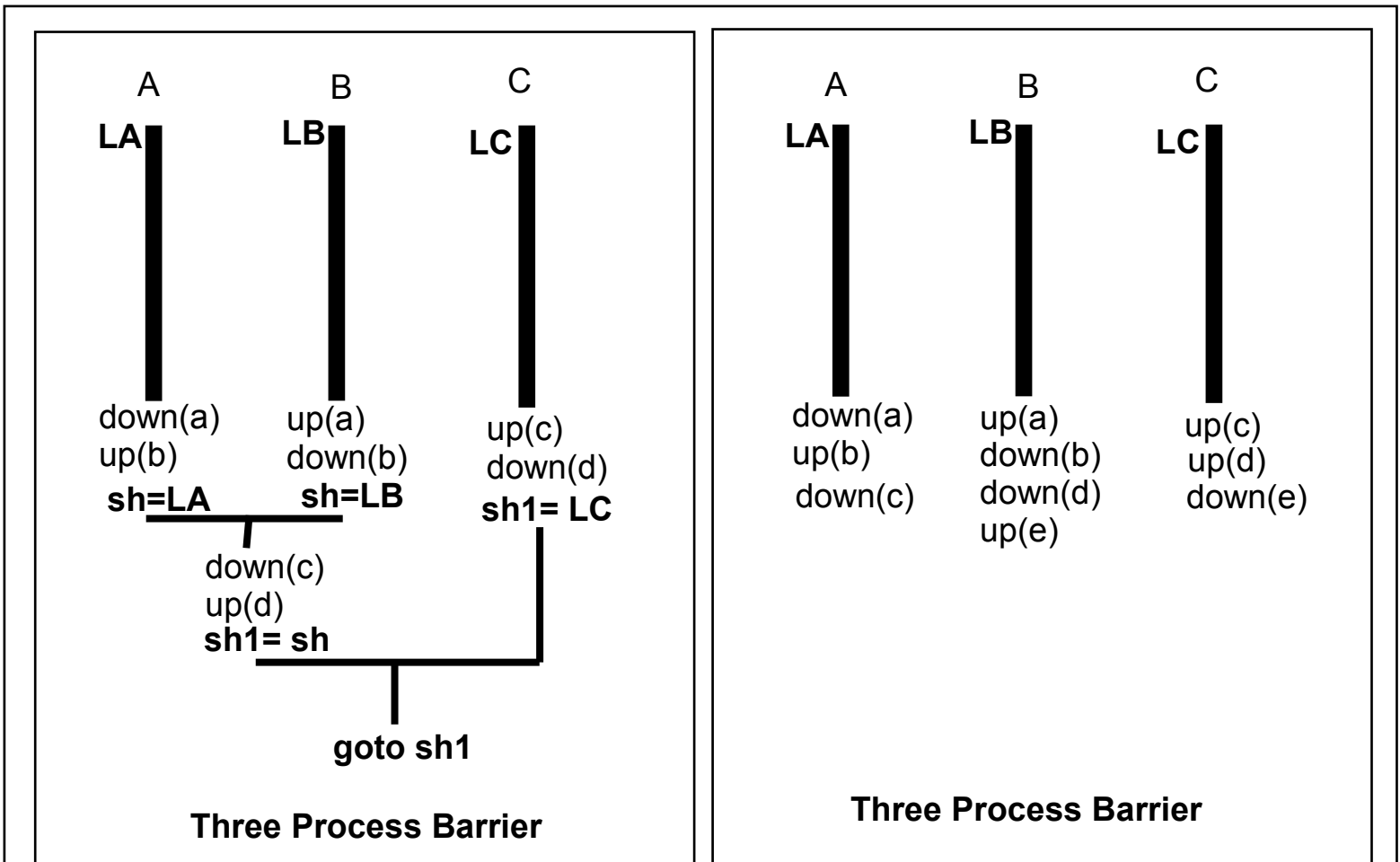
GENERAL BARRIER ACHIEVED WITH SEMAPHORS (Monitor Preview)



If A finishes 1st: down(a) and blocks
 B finishes next: up(a) unblocks A

A continues through up(b)
 B can continue through down(b)

B blocks on down(b)
 A does up(b)
 B can continue through down(b)



A goes until it blocks or restarts
 B goes until it blocks or restarts
 C goes until it blocks or restarts

2 and 3 Process BARRIER ACHIEVED WITH SEMAPHORS

```
#define N 27
semaphor mutex = 1;
semaphor empty = N;
semaphor full = 0;
int shared item(N);
int shared_i =1, shared_k=1;
```

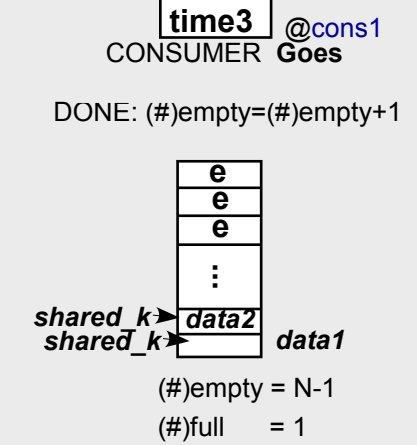
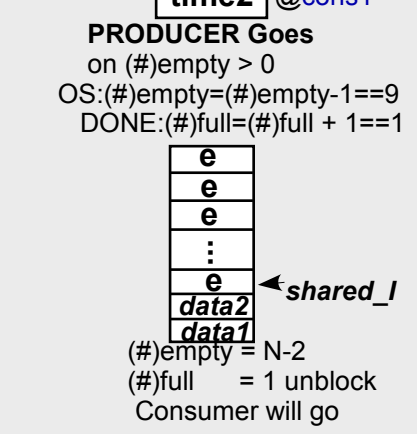
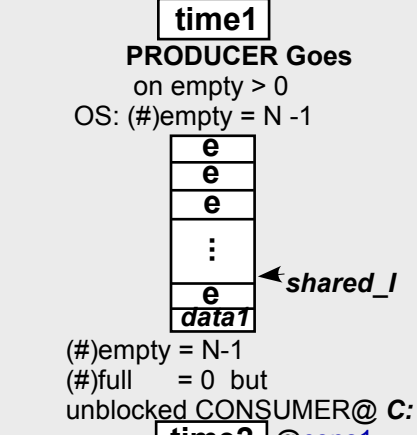
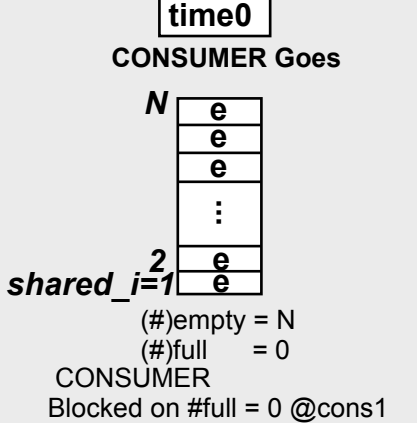
```
void producer(void)
{
  int item;
  while(TRUE)
  {
    produce_item(&item);
    (down(&empty) /* if (empty > 0) empty = empty - 1, else BLOCK */
    down(&mutex);
    enter_item(item,@shared_l); shared_l=shared_l+1
    up(&mutex)
    up(&full) /*(Only after its definitely in) if any process is
    BLOCKED on semaphor full UNBLOCK one,
    otherwise full = full+1 */
  }
}
```

```
void consumer(void)
{
  int item;
  while(TRUE)
  {
    down(&full) /* if (full > 0 ) full = full - 1, else block */
    C: down(&mutex)
    remove_item(&item @ shared kl); shared k =shared +1;
    up(& mutex);
    up(&empty) /*(Only after definitely out) if any process is
    BLOCKED on semaphor empty UNBLOCK one,
    otherwise empty = empty+1 */
    consume_item(item);
  }
}
```

“(#)empty” contains a count of the number of buffer entries that are empty (available) (#)empty). Initially N, Generally (#)empty, is checked by PRODUCER (down(empty) PRODUCER blocks only if 0 Only after the CONSUMER has removed an entry does it increment ((#)empty), up(&empty), making another space in table available to PRODUCER.

“(#)full” contains a count of the number of buffer entries that are occupied (#full) Initially 0 (#)full, it is checked by CONSUMER (down(full) CONSUMER blocks only if 0. Only after the PRODUCER has completed an entry does it increments ((#)full),up(&full) so CONSUMER has another entry to consume.

“mutex” is a binary semaphor. It makes access to the buffer by PRODUCER and CONSUMER mutually exclusive.



As long as neither Producer or Consumer is blocked nor Active(#)empty + (#)full = N. But if Producer is blocked on down((#)empty) (#)empty + #full = N-1or if Consumer is blocked on down((#)full) then also (#)empty + #full = N-1 is possible.

What happens with multiple Producers and Consumers?
PRODUCER-CONSUMER WITH COUNTING SEMAPHORS and indexed Table