<table>
<thead>
<tr>
<th>2</th>
<th>IPC SYSTEM INVOLVED MUTUAL EXCLUSION AND SYNCHRONIZATION MECHANISMS- DISABLING INTERRUPTS, SEMAPHORS, MONITORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>SYSTEM INVOLVED MUTUAL EXCLUSION AND SYNCHRONIZATION MECHANISMS-PIPES, FILES, MESSAGE PASSING, SIGNALS</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>4</th>
<th>SEMAPHOR: DOWN AND UP SURROUNDING CRITICAL REGIONS GIVES MUTUAL EXCLUSION -- TWO FORMS OF SYNCHRONIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>SEMAPHORS: BINARY MUTUAL EXCLUSION SEMAPHOR TRACE</td>
</tr>
<tr>
<td>6</td>
<td>SYNCHRONIZATION WITH BINARY SEMAPHORS</td>
</tr>
<tr>
<td>7</td>
<td>COUNTING SEMAPHORS: SEMAPHOR TRACE</td>
</tr>
<tr>
<td>8</td>
<td>COUNTING SEMAPHORS (n) CONSTRUCTED WITH BINARY SEMAPHORS(x, y)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>9</th>
<th>BINARY SEMAPHOR-TRACE TYPE 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>BINARY SEMAPHOR-TRACE TYPE 2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>11</th>
<th>PRODUCER-CONSUMER WITH COUNTING SEMAPHORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>PRODUCER-CONSUMER WITH BINARY SEMAPHORS AND WITH TSL 1</td>
</tr>
<tr>
<td>13</td>
<td>PRODUCER-CONSUMER WITH TSL 2 WITH FEWER WRITES TO lock</td>
</tr>
</tbody>
</table>

Barriers-Synchronized starts of multiple Processes

<table>
<thead>
<tr>
<th>14</th>
<th>GENERAL BARRIER ACHIEVED WITH SEMAPHORS (Monitor Preview)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2 and 3 Process BARRIER ACHIEVED WITH SEMAPHORS</td>
</tr>
</tbody>
</table>

CONTENTS

© M C Paull 2005
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, page 2- 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 Synchronization 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

© M C Paull 2005
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.)

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 semaphors. 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.
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; (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 being > 0 will cause P to continue.

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

When P1 is in its Crit, it tests a condition (cond1) to determine if it can continue, ex. Table is not Full, If it cannot continue it must allow a companion region, in P2, to run to alleviate that condition, ex. remove entry(s) from Table. So it must allow P2 to run in P2s Crit, so up(m) allows P2 to run and down(z) blocks P1 on z. When P2 runs it will under cond2 unblock P1 with up(z). When P2 does this it also exits-leaves its Critical Region. This is necessary because P1 is Ready in its Crit and m=0. Now P1 must run next.

This is like I except exit from P2 is not forced after up(z). up(z) unblocks P1 as in I but P2 can continue to run and if before P2 exits through up(m) OS activates P1, P1 will block on down(m)

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

© M C Paull 2005
for each semaphor one could keep the list of all processes blocked on that semaphor 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 semaphor 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.
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 \(\text{if}(\text{cond1})\{\text{up}(m)\};\text{down}(z);\}\) is encountered. So, assuming \(\text{cond1}\) 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 enters its CRIT, through \(m=1\). \(m\) becomes 0. Now, assuming \(\text{cond 2}\) in \(\text{if}(\text{cond2})\{\text{up}(z)\};\}\) is true the \(\text{up}(z)\) 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 \(\text{up}(m)\).
**semaphor m = n (n>= 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 }
  else  
  {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.) }
  ```

**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.
semaphor x = 1 /*initially unblocked*/, y = 0; /*initially blocked*/ 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*/

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

© M C Paull 2005 8

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 blockon y outside the Critical Regioneliminated 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.
On up unblock the most recent to be blocked
If $I_{prog}$ is short Process I may run many times before a waiting Process J can run in fact it could starve process J.

Possible Fix:
- Do More or
- Delay

Possible Problems Of Different Mutual Exclusion Implementations:
- **TSL-Possible Problem: Hog Lock Starvation**
- **Semaphore No Starvation-Possible Alternation**
- **Peterson’s Alg No Starvation-Possible Alternation**
#define N 27
semaphor mutex = 1;
semaphor empty = N;
semaphor full = 0;
int shared item(N);

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);
        up(&mutex)
        up(&full) /*(Only after its definitely in) if any process is
BLOOKE on semaphor full UNBLOOKE 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);
        up(&mutex);
        up(&empty) /*(Only after definitely out) if any process is
BLOOKE on semaphor empty UNBLOOKE 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?
```c
#define N 27
int shared n = N; /*the integer variable n serves to count entries in buffer*/
int shared lock0 = 0, lock1=1, lock2=1 /*initially unlocked*/,
int shared buffer;
while(TRUE)
{
    produce_item(&item);
    down(x);
    if (n == N) {up(x);down(y);} /*if full block on yt*/
    enter_item(&item,buffer);
    n = n + 1;
    if (n == N-1) up(y);continue; /*if n= N -Prod blocked on y, unblock y*/
    up(x);
}
while(TRUE)
{
    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 empty block on y*/
    up(x);
    consume_item(&item);
}
}
```

**PRODUCER-CONSUMER WITH BINARY SEMAPHORS AND WITH TSL 1**

© M C Paull 2005
PRODUCER

while(TRUE)
{
    produce_item(&item);
    //...
    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;
    //...
}

CONSUMER

while(TRUE)
{
    //...
    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);
    //...
}

* 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 semaphor. Instead of the OS **Remembering** the Processes waiting in the blocked state on the binary semaphor and moving one from the **blocked** on the binary semaphor to the **Ready state** when an **up** occurs on that semaphor, 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.
A sequence of 8 states s0-s7, each 0 or 1, called S0+ is transformed into a sequence S1, a small change gives S1+ then again S1+ is transformed by the same procedure to sequence S2, etc. The transformation is given by the function: $s_i = s_{i+1} \oplus s_{i-1}$

The transformations are carried out by a sequence of 7 Processes. Each process reads its two neighbor states in S0 and puts its new state into S1. S1 is not to be read until all processes have put their new entry into it. (rendevous). Only 7 states result so S1 is then shifted with 0 added at the front as s0, giving S1+. will be read and new entries placed in Copy 0, etc.

| States | Process
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 1 1</td>
<td>P1 P2 P3 P4 P5 P6 P7</td>
</tr>
<tr>
<td>1 0 1 0 1 0 0 0</td>
<td>S1</td>
</tr>
<tr>
<td>0 1 0 1 0 1 0 0</td>
<td>P1 P2 P3 P4 P5 P6 P7</td>
</tr>
<tr>
<td>1 1 1 1 1 1 0 0</td>
<td>S2</td>
</tr>
</tbody>
</table>

#define N 10
int n=0;
semaphor mutex = 1;
semaphor barrier = 0;
while(TRUE)
{
  down(mutex);
  if (n == N-1)
  { while(n>0) {up(barrier); n=n-1; } up(mutex); }
  else {n=n+1; up(mutex); down(barrier);}
}

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);
    }
  }

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.
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_I);
        shared_I = shared_I + 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 */
        down(&mutex)
        remove_item(&item[@shared_k]);
        shared_k = shared_k + 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. (#empty) 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 increment (#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-1) or 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