## 2 MONITOR USE - TRACE WITH *wait* AND *signal*

Example application of Monitors and translation of condition variables, waits and signals to semaphors. Downs and ups in special case. Critical Region Shared Memory orientation

## 3 TRANSLATION MONITOR TO BINARY SEMAPHORS

**PRODUCER, CONSUMER**. Automatic exit on *signal* **FIXING PROBLEMS-DETAILS**

## 4 QUALIFYING THE GIVEN TRANSLATION MONITOR TO BINARY SEMAPHORS

[Contents]

<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>SINGLE VS MULTIPLE MONITORS</td>
</tr>
<tr>
<td>6</td>
<td>THE TWO MONITOR CASE</td>
</tr>
<tr>
<td>7</td>
<td>CLASSICAL PROBLEMS-MONITOR SOLUTIONS</td>
</tr>
<tr>
<td>8</td>
<td>DINING PHILOSOPHERS SEMAPHOR SOLUTION BY COMPILATION OF MONITOR SOLUTION</td>
</tr>
<tr>
<td>9</td>
<td>MANY READERS ONE WRITER, SLEEPING BARBER SEMAPHOR SOLUTION BY COMPILATION OF MONITOR SOLUTION</td>
</tr>
<tr>
<td>10</td>
<td>MANY READERS TWO WRITERS</td>
</tr>
<tr>
<td>11</td>
<td>COMMUNICATION: PIPES (1-Way) Like Files</td>
</tr>
<tr>
<td>12</td>
<td>PIPE Example</td>
</tr>
<tr>
<td>13</td>
<td>PIPES, MESSAGE PASSING (SEND AND RECEIVE)</td>
</tr>
<tr>
<td>14</td>
<td>MESSAGE PASSING BASICS: EXAMPLES</td>
</tr>
<tr>
<td>15</td>
<td>MESSAGE PASSING COMMUNICATION BUFFER AND MAILBOXES TRACE</td>
</tr>
<tr>
<td>16</td>
<td>MESSAGE PASSING BY RENDEZVOUS</td>
</tr>
<tr>
<td>17</td>
<td>SIGNALS</td>
</tr>
<tr>
<td>18</td>
<td>TRANSLATION MONITOR TO BINARY SEMAPHORS <strong>PRODUCER,-CONSUMER.</strong> No automatic exit on signal <strong>FIXING PROBLEMS-DETAILS</strong></td>
</tr>
<tr>
<td>19</td>
<td>MESSAGE PASSING COMMUNICATION BUFFER AND MAILBOXES TRACE</td>
</tr>
</tbody>
</table>

Alternatives Monitor-Semaphor Translation, Message passing

© M C Paull 2005
Mutual Exclusion of Procedures

If Process P is executing procedure Proc In Monitor M. then no process other than Proc can be active in a procedure defined within M.

Monitor <Monitor Name>

* condition x
  condition y
  <other declarations>
** <proc_1 name> (arguments)
  <Procedure1 Body_1>
  .
  .
  .
** <proc_n name> (arguments)
  <Procedure Body_n>
End<Monitor Name>

Monitors have been defined in different ways. The first Monitor type we describe is perhaps the simplest of these with certain advantages and disadvantages considered later with other Monitor definitions.

It implies that using a Monitor:

1. Only one Process can be active within an Monitor procedure at a time. (True of all Monitor models)
2. A Process can only be blocked at the call of M’s procedure because another Process is still active within a procedure, in M, or inside a procedure at a wait(…) of M. It can not block elsewhere.
   (Note that using semaphors a Process can be blocked anywhere, within or outside a Critical Region.)
3. With this description of the signals effect-if a Process P is waiting in a procedure in M, and it recieves a signal from another procedure in M, run by Process Q. then the next Process to be active in M will be P. (If more than 1 is waiting on that signal one of them will be the next to run in M.)

Synchronization

Use condition variables*(y here) and two commands:

wait(y) Process P blocks within procedure proc at the point just after the wait(x). proc is no longer active, so another procedure (or in some implementation in M the one with the blocked wait) can be called by another Process..

signal(x) P leaves proc. because this command must be immediately followed by a exit in proc (If another procedure was blocked on x (on a wait(x)), it will be unblocked and made to continue from the point after the wait(x).

* a condition variable is a name without contents. It Serves to relate wait and signal calls.

MONITOR USE -TRACE WITH wait AND signal
Producer-Consumer With Monitor

Producer Process
{while(TRUE)
{produce(item); Prod-Cons.enter;}
}

Consumer Process
{while(TRUE)
{Prod-Cons.remove; consume(item);}
}

Monitor Prod-Cons
int Table[N];
condition bempty;
condition bfull;
int count = 0, N = 40

procedure(enter)
if(count==N) wait(bfull);
enter_item1(Table); count=count + 1;
if (count==1) signal(bempty);

procedure(remove)
if(count==0) wait(bempty);
remove_item1(Table); count=count - 1;
if (count==N - 1) signal(bfull);

End Monitor

synchronized mutex = 1, bfull = 0, bempty = 0;
int shared count = 0, N = 40;

Consumer

PROBLEM: on count==0
no entry through
down(x) possible

down(mutex);
if(count==0) { down(bempty);}
remove_item(item); count=count - 1;
if(count==N - 1) {up(bfull);exit;}
up(mutex);

SOLUTION: up(mutex) makes entry through
down(x) is possible

down(mutex);
if(count==0) { up(mutex); down(bempty);}
remove_item(item); count=count - 1;
if(count==N - 1) {up(bfull);exit;}
up(mutex);

1 OK

Producer

PROBLEM: on count==N
no entry through
down(x) possible

down(mutex);
if(count==N) {down(bfull);}
enter_item1(); count=count + 1;
if (count==1) {up(bempty);exit;}
up(mutex);

SOLUTION: up(mutex) makes entry through
down(x) is possible

down(mutex);
if(count==N) {up(mutex); down(bfull);}
enter_item1(); count=count + 1;
if (count==1) {up(bempty);exit;}
up(mutex);

1 OK

transformation MONITOR TO BINARY SEMAPHORS

PROBLEM: on count==0
no entry through
down(x) possible

down(mutex);
if(count==0) { down(bempty);}
remove_item(item); count=count - 1;
if(count==N - 1) {up(bfull);exit;}
up(mutex);

PROBLEM: on count==1
if Producer exits through up(bempty) though Consumer
never blocked on down(bempty) then mutex remains 0
and neither Consumer nor Producer can run

down(mutex);
if(count==N) {up(mutex); down(bfull);}
enter_item1(); count=count + 1;
if (count==1) {up(bempty);exit;}
up(mutex);

SOLUTION Only exit through up(bempty) if
Consumer exited through down(bempty)-cons flag

down(mutex);
if(count==N) {up(mutex); down(bfull);}
enter_item1(); count=count + 1;
if (count==1) {up(bempty);exit;}
up(mutex);

2 OK

TRANSLATION MONITOR TO BINARY SEMAPHORS
PRODUCER,-CONSUMER.Automatic exit on up(condition) FIXING PROBLEMS-DETAILS

© M C Paull 2005
In general the compilation of the Monitor code to semaphor code will work if:

The shared memory, and semaphors that replace the condition variables declared in the Monitor are made accessible to all Processes which call procedures in the Monitor and if procedure p() in the monitor is called from process P at position x,

1. a copy of the body of that procedure with the downs and ups required in the compilation replace the call to the Monitor at position x or.
2. The procedure, with the downs and ups required in the compilations is placed in process P, and the calls to proc p() directed to the Monitor be directed to the process P version of the procedure.

In an implementation in which a single copy of procedure P could be called from two or more different Processes then, while one Process was executing P, one or more others can call P. This means code is shared by all those Processes (re-entrant code is needed) and that Processes calling P must be queued.

Furthermore, queuing Process access to procedure also implies the necessity to queue Processes waiting and also those signalling from the same Procedure in the same procedure. We say effectively because if a procedure P in the Monitor were called by more than one Process the Compiler could make copies of P uniquely available to each of the Processes. In fact one could substitute the body of a procedure for each of its calls-with the accompanying parameters instantiated.

In some of the examples that follow we will allow a Procedure in a Monitor to be called from more than one Process, ex. Dining Philosophers. The implementation for these examples can be achieved by the simple transformation from Monitor to semaphors given previously if the appropriate multiple copies of procedures are made by the Compiler. Alternatively they could be implemented with shared code this requires more than the simple transformation—there must then be ways to queue processes waiting to execute shared procedures.

QUALIFYING THE GIVEN TRANSLATION MONITOR TO BINARY SEMAPHORS
SINGLE VS MULTIPLE MONITORS

P1-Producer --> P2-Massage --> P3-Consumer

© M C Paull 2005
firstproducer()
{ Produce_item(item);
  while(TRUE)
    { Prod-Con1.enter(item); } /*item->M12*/
}

take-massage-give()
{ while(TRUE)
    { Prod-Con1.remove(T1); massage(T1,T2); /*M12->T1, 
      massage(T1) ->T2*/
      Prod-Con2.enter(T2); } /*T2->M23*/ }

finalconsumer(item)
{ while(TRUE)
    { Prod-Con2.remove(item); Consume(item); } /*M23->item*/
}

Monitor Prod-Con1
  condition bempty = 0;
  condition bfull = 0;
  int count = 0, N12 = 40, in = 0, out = 0;
  int M12 [40]
  enter(source){
    if(count==N12) wait(bfull);
    move(M12[in], source); count=count + 1;
    in = in + 1%N12;
    if (count==1) signal( bempty); }
  remove (dest) {
    if(count==0) wait( bempty);
    move(M12[out], dest); count=count - 1;
    out = out + 1 % N12;
    if(count==N12 - 1) signal( bfull);}

Monitor Prod-Con2
  condition bempty = 0;
  condition bfull = 0;
  int count = 0, N23 = 60, in = 0, out = 0;
  int M23 [60]
  enter(source) {
    if(count==N23) wait( bfull);
    move(M23[in], source); count=count + 1;
    in = in + 1%N23;
    if(count==1) signal( bempty); }
  remove(dest) {
    if(count==0) wait( bempty);
    move(source, dest); count=count - 1;
    out = out + 1 % N23
    if(count==N23 - 1) signal( bfull); }

THE TWO MONITOR CASE
DINING PHILOSOPHERS

Thinks-Hungry-GETS Chopsticks-Eats In order to eat need left and right chopsticks
i.e., Neither Philosopher on left or right is Eating

If a Philosopher (Process), P1, is Hungry and finds that at least one of its chopsticks is unavailable
(at least one of its neighbors is Eating), P1 waits. P1 can only be restarted by another Philosopher
(Process), P2. P2 must be one of those that P1 found Eating and now is finished Eating.

The Process for the in Philosophers is:

```
while(TRUE)
{
    delay
    F.getchopsticks(i); /*Hungry*possible delay*/
    delay
    other E->T
    F.putchopsticks(i); Think, Other Eat?
}
```

Notes: This solution still leaves the possibility of starvation-explain?

The procedures in this Monitor are called with parameters from >1 different Processes-they could be
replaced with one procedure/Process (page 5)

DATABASE (Many Simultaneous Readers, One Writer(No Readers))

If any reader is active the writer must wait. If Writer is waiting and the number of Readers (rc) goes to 0 the Writer writes. While the writer is writing in try-write() no reading is possible because of the Mutual Exclusion of procedures.

The Process for the in Reader is

```
while(TRUE){
    RW. pre_read(); /*enter iff writer idle or wait*/
    read_database();
    RW.post_read();
    use_data();
}
```

The Writer Process is

```
while(TRUE){
    produce_data();
    RW .try_write();
}
while(TRUE){
    Delay
    F.getchopsticks(i);
    Delay
    F.putchopsticks(i);
}
**MANY READERS, ONE WRITER**

The Process for the $i$th reader is:

```c
while(TRUE) {
    RW.pre_read();
    read_database();
    RW.post_read();
    use_data();
}
```

The writer Process:

```c
while(TRUE) {
    produce_data();
    RW.try_write();
}
```

**Monitor RW**

```c
int rc=0;  [rc=number of readers]
cond db;
pre-read() {
    rc = rc+1;
post-read() {
    rc=rc-1;
    if(rc== 0) signal(db); }
try_write1() {
    if (rc > 0) wait (db);
    write_database();
}
```

**End Monitor**

**The Process for the $i$th customer is**

```c
BC.customer;
```

**Monitor BC**

```c
int chairs=0;  [chairs = # of customers]
cond custnum;
barber_checks() {
    if (chairs==0) wait (custnum);
    chairs = chairs-1; }
customer() {
    if (chairs<N) chairs = chairs+1;
    if (chairs==1) signal(custnum); }
```

**End BC**

**The Procedures are redefined using semaphors**

```c
int rc = 0;
semaphor db=0, mutex1=1;
database
```

**The barber code is:**

```c
while(TRUE) {
    BC.barber_checks();
    cut_hair();
}
```

**The writer Process**

```c
while(TRUE){
    RW.pre_read();
    read_database();
    RW.post_read();
    use_data();
}
```

**For each customer the code is**

```c
while(TRUE){
    BC.barber_checks();
    cut_hair();
}
```

**The writer Process**

```c
while(TRUE) {
    produce_data();
    RW.try_write();
}
```

**The barber code is:**

```c
while(TRUE) {
    BC.barber_checks();
    cut_hair();
}
```

**For each customer the code is**

```c
while(TRUE) {
    BC.barber_checks();
    cut_hair();
}
```

**SHARED MEMORY**

© M C Paull 2005
The Process for the $i$th reader is
while(TRUE) {
    RW.pre_read();
    read_database();
    RW.post_read();
    use_data();
}

The writer1 Process
while(TRUE){
    produce_data();
    RW.try_write1();
}

The writer2 Process
while(TRUE){
    produce_data();
    RW.try_write2();
}

Monitor RW

```c
int rc=0; // rc=number of readers
cond db;
pre-read() {
    rc = rc+1;
}
post-read() {
    rc=rc-1;
    if(rc== 0) signal(db);
}
try_write1() {
    if (rc > 0) wait (db);
    write_database();
    signal(db);
}
try_write2() {
    if (rc > 0) wait (db);
    write_database();
    signal(db);
}
```

End Monitor
Inter Process Communication Through A Pipe

#include “ourhdr.h”
#define MAXLINE 1024
int main(void);
{
    int fd[2];
    pid_t pid;
    char line[MAXLINE];
    if ( pipe(fd) < 0 ) err_sys("pipe error"); /*open-no name*/
    if ( (pid = fork()) < 0 ) err_sys("fork error");
    else
    {
        if (pid > 0)
        { /*parent code*/
            close(fd[0]);
            write(fd[1], "hello world\n", 12);
        }
        else
        { /*child code*/
            close(fd[1]);
            n = read(fd[0], line, MAXLINE);
        }
    }
    exit(0);
}

Inter Process Communication Through Shared File

#include “ourhdr.h”
int main(void);
{
    pid_t pid;
    char line[MAXLINE];
    fd=open(/paull/inoutfile, RWONLY) /*open by name*/;
    if ( (pid = fork()) < 0 ) err_sys("fork error");
    else
    {
        if (pid > 0)
        { /*parent code*/
            write(fd, "hello world\n", 12);
        }
        else
        { /*child code*/
            n = read(fd, line, MAXLINE);
        }
    }
    exit(0);
}

# include <unistd.h>
int fd[2];
char buf[7];
pipe(fd); /*open-no name*/
write(fd[1], "test it", 7);
read(fd[0], buf, 7);

FILE IN SINGLE PROCESS

#include "ourhdr.h"
int main(void);
{
    int fd[2];
    char buf[7];
    fd=open(/paull/inoutfile, RWONLY) /*open by name*/;
    write(fd, "test it", 7);
    read(fd, buf, 7);
    exit(0);
}

PIPE In SINGLE PROCESS

#include "ourhdr.h"
#define MAXLINE 1024
int main(void);
{
    int fd[2];
    pid_t pid;
    char line[MAXLINE];
    if ( pipe(fd) < 0 ) err_sys("pipe error"); /*open-no name*/
    if ( (pid = fork()) < 0 ) err_sys("fork error");
    else
    {
        if (pid > 0)
        { /*parent code*/
            close(fd[0]);
            write(fd[1], "hello world\n", 12);
        }
        else
        { /*child code*/
            close(fd[1]);
            n = read(fd[0], line, MAXLINE);
        }
    }
    exit(0);
}

© M C Paull 2005

COMMUNICATION: PIPES (1-Way) Like Files
```c
#include <unistd.h>
#include <stdio.h>
#define BUFSIZE 1024

main(int argc, char **argv)
{
    int fd[2], n;
    char buf[BUFSIZE];
    if (pipe(fd) == -1) { perror("pipe"); exit(1); }

    switch (fork() )
        case 0:
            map(fd[0], fd[1]); exit(0); /* Child */
        default:
            source(0, fd[1]); break; /* Parent */
    case -1
        perror("fork");
        break;
}
wait((int*)0);

n=read(fd[0], buf, BUFSIZE)
if(n > 0) { buf[n]=\0; printf("received %s \n", buf); }

source(int readfd, int sendfd)
{
    char buf[BUFSIZE];
    int n;
    n=read(readfd, buf, BUFSIZE)
    if(n > 0) { if (buf[n-1] == \n) --n; write(sendfd, buf, n); }
}

struct MAPPER
{
    char *left;
    char *right;
}

struct MAPPER T[] = { {"dog", "cat"},
                      {"horse", "mule"},
                      {"rabbit", "slug"},
                      {0, \0} };

map(int readfd, int replyfd)
{
    char buf[BUFSIZE];
    int i, n;
    n=read(readfd, buf, BUFSIZE)
    if(n > 0)
        { buf[n]=\0; printf("got \ " %s\n \n", buf);
          for(i=0; T[i].left; ++i) /* while T[i].left != 0 read PIPE */
            { if( !strcmp(T[i].left, buf)){write (replyfd,T[i].right, strlen(table[i].new)); return} }
        }/*if it is in T[i] on left print corresponding right*/
    write (replyfd, "error", 6);
}
```

**(PIPE Example)**

© M C Paull 2005
We have described how send and receive commands can be implemented with the help of mailboxes in our description of the Multiple Alternator scheme. In that implementation there is a (Synchronization) test preceding the actual sending and receiving. The test results in either continuing execution or to Blocking (through Busy Waiting). This Synchronization test occurs in both the Producer (blocked if mailbox is full) and Consumer (blocked if mailbox is empty) code shown there. When, as considered now, Message Passing is implemented by the OS, it also uses mailboxes and tests which processes the send and receive calls from the user. These result in passage forward to the actual send or receive or to Blocking, by the caller. Unlike with Multiple Alternators, all pointer and message storage is handled by the OS. The two system calls available to the processes are:

\[
\begin{align*}
\text{receive}&(\text{from\_process},\ \text{local\_message\_origin}\ (\text{buffer}) ), \quad \text{and} \\
\text{send} &\ (\text{to\_process},\ \text{local\_message\_destination}\ (\text{buffer}) ),
\end{align*}
\]

These calls go to the operating system usually to remove and place messages in a Mailbox (shared and under Op Sys control), using basically similar technique as used for Multiple Alternators.

Here we are describing one of a large number of ways in which message passing can be implemented. In particular, here the send and receive operations are assumed synchronous-blocking till success (as opposed to asynchronous) continue whether successful or not, with some test in a returned value for failure or success. The mailboxes are kept in the Kernel, rather than in user space. The messages are fixed maximum size. Here the messages are read in order they arrive in the mailbox and the position to be read is advanced-so if one is looking for a message from a particular origin it may not be the next message in the mailbox. This need not be the case. It is possible to identify messages by their type, and/or by their origin and for receive to search for the type or origin. The implementation examples given next assumes the Receiver is always reading the next entry in the mailbox for its use.

Message Passing is not so important with shared memory communications, but, being the only means of communication when Processors are at a great distance from one another, is of prime importance when dealing with distributed systems.
There are different implementations based on the arrangement of Mailboxes.

1. One possibility is to have each user assigned its own mailbox for incoming mail. In this case, the identity of the sender is kept as part of the message in the mailbox.

2. An alternative is to have mailboxes assigned to each pair for each direction of communication. So the identity of the mailbox is determined by the identity of both communicators. There is an extreme version of this approach, called Rendezvous, in which, effectively, the Mailbox has room for only one message. In the case of Rendezvous, the producer and consumer alternate in sending and receiving a single message. If within one quanta it is possible produce two messages and receive two messages then it would be better to allow two messages to be sent by the producer and then both consumed by the consumer. This leads to the idea of specifying the number of messages to be handled by sending a number of message requests \((\epsilon_s)\) based on the number that can be produced in a quanta (and consumed).

Process schema and traces using this \(\epsilon_s\) scheme for each of alternatives 1 and 2, are given below.

**Circular Buffer Again (Like Multiple Alternators)**

**The send and receive have the following properties**

- **send(msg, mailbox_id)** blocks until the current slot in mailbox_id is empty. It then fills it with msg and its next send will be to the next slot in mailbox_id.
- **receive(mailbox_id, m_loc)** blocks until the current slot in mailbox_id holds a message. It then moves that message to local memory m_loc makes the slot devoid and the next receive will be from the next slot.

**Definitions**

**Message Passing Basics: Examples**

There are different implementations based on the arrangement of Mailboxes.

### PRODUCER (Sender)

```c
while(TRUE) {
    produce(m_p);
    receive(m_c, consumer);
    put(m_p, msg_p);
    send(consumer, msg_p);
}
```

### CONSUMER (Receiver)

```c
int m_\epsilon = \epsilon; // N messages will be accommodated.
for(i=0;i<N;i++) {send(producer, msg_c)}
while(TRUE) {
    receive(msg_p, producer);
    get(msg_p, m_p);
    put(m_\epsilon, msg_c);
    send(producer, msg_c);
    consume(m_p);
}
```

**Used for Producer Consumer Problem**

\(\epsilon\) requests can be thought of as empty message containers to be filled and sent by the producer.

---

**Message Passing Basics: Examples**

© M C Paull 2005 14
while(TRUE)
{
    produce(m_p);
    receive( m_c, consumer);
    put(m_p, msg_p)
    send(consumer, msg_p);
}

while(TRUE)
{
    receive( msg_p, producer);
    get(msg_p, m_p);
    put(m_c, msg_p);
    send(producer, msg_c);
    consume(m_p);
}
Rendezvous involves a producer Process that sends, consume Process that receives:

```c
int s=0; r=0; /* r=1 indicates a receive has been executed but the corresponding send has not,
s=1 indicates a send has been executed but the corresponding receive has not. */
```

**send(to, sendm)** is a system call which results in the OS doing:
- if \( r == 1 \) { \( c( r_{sv} ) = sendm \)
  - \( r = 0; \) unblock;
- else \( s = 1; \) sendm=location where sent msg is; block

**receive(from, recm)** is a system call which results in the OS doing:
- if \( s == 1 \) { \( recm = c(s_{sv}); \)
  - \( s = 0; \) unblock;
- else \( r = 1; \) recm=location at which receiver gets msg; block

**MESSAGE PASSING BY RENDEZVOUS**
```c
#include <stdio.h>
#include <signal.h>

main()
{
    if (fork()>0) /*assuming fork always works, returns >0 = childs ID in parent*/
    {
        for (;;)
        {
            printf("I'm Parent");
        }
    }
    else
    {
        sleep(5);
        kill(getppid(), SIGKILL);
    }
}

# include <stdio.h>
# include <signal.h>
main()
{
    if (fork()>0) /*assuming fork always works, returns >0 in parent*/
    {
        /*Parent Code*/
        signal(SIGUSR1, catchme); /*When signal SIGUSR1, is received the procedure catchme is executed if it doesn’t cause an exit*/
        for(;;)
        {
            printf("I'm Parent");
        }
    }
    else
    {
        sleep(5);
        kill(getppid(), SIGUSR1);
    }
}

catchme() /*signal handler*/
{
    printf("Signal caught");
    exit(0);
}
```

- `kill(pid, signal_id)`
  - `<pid>` indicates destination of signal
  - 0 indicates all processes in group
  - -1 indicates all processes with the same user ID

- `signal_id`
  - Two examples
    1. `signal_id = SIGKILL = :kill destination`
    2. `signal_id= SIGUSR1 = causes the destination to run a function it (the destination) specifies in a signal command at the destination which looks like:

- `signal(SIGUSR1,<name of procedure called in destination Process >*)`
Monitor Prod-Cons

int Table[N];
condition bempty;
condition bfull;
int count = 0, N = 40

procedure(enter)
if(count==N) wait(bfull);
enter_item1(Table); count=count + 1;
if (count==1) signal(bempty);

procedure(remove)
if(count==0) wait(bempty);
remove_item1(Table); count=count - 1;
if (count==N-1) signal(bfull);

End Monitor

Producer-Consumer With Monitor

Producer Process
{while(TRUE)
    {produce(item); Prod-Cons.enter;}
}

Consumer Process
{while(TRUE)
    {Prod-Cons.remove; consume(item);}
}

No automatic exit on signal FIXING PROBLEMS-DETAILS
while(TRUE)
{ produce(m_p);
 receive( m_c,consumer);
 put(m_p, msg_p)
 send(consumer, msg_p);
 } PRODUCER (Sender)

while(TRUE)
{ receive( msg_p, producer);
 get(msg_p,m_p); ,put(m_ε, msg_c)
 send(producer, msg_c);
 consume(m_p)
 } CONSUMER (Receiver)

int m_ε = ε;
N messages will be
accommodated.
for(i=0,i<N,i++) {send(producer, msg_c)
while(TRUE)
{ receive( msg_p, producer);
 get(msg_p,m_p); ,put(m_ε, msg_c)
 send(producer, msg_c);
 consume(m_p)
 } CONSUMER (Receiver)