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

Considerations in use of Monitor

5 SINGLE VS MULTIPLE MONITORS

6 THE TWO MONITOR CASE P1-Producer --> P2-Massage --> P3-Consume

Standard problems requiring Mutual Exclusion and Synchronization with Monitor AND Semaphor Solutions.

7 CLASSICAL PROBLEMS-MONITOR SOLUTIONS

8 DINING PHILOSOPHERS SEMAPHOR SOLUTION BY COMPILATION OF MONITOR SOLUTION

9 MANY READERS ONE WRITER , SLEEPING BARBER SEMAPHOR SOLUTION BY COMPILATION OF MONITOR SOLUTION

10 MANY READERS TWO WRITERS

Aurtomatically atomic writes and reads to Memory-Pipes and Files for Mutual Exclusion-Simple Programs

11 COMMUNICATION: PIPES (1-Way) Like Files

12 PIPE Example

13 PIPES, MESSAGE PASSING (SEND AND RECEIVE)

IPC by sending messages Talking to each other orientation

14 MESSAGE PASSING BASICS: EXAMPLES

15 MESSAGE PASSING COMMUNICATION BUFFER AND MAILBOXES TRACE

16 MESSAGE PASSING BY RENDEZVOUS

Asynchronous Communication-Interrupt orientation

17 SIGNALS

Alternatives Monitor-Semaphor Transslation, Message passing

18 TRANSLATION MONITOR TO BINARY SEMAPHORS PRODUCER,-CONSUMER.
No automatic exit on signal **FIXING PROBLEMS-DETAILS**

19 MESSAGE PASSING COMMUNICATION BUFFER AND MAILBOXES TRACE
Physical Buffer Larger Than Number Of ε s

CONTENTS

© 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>

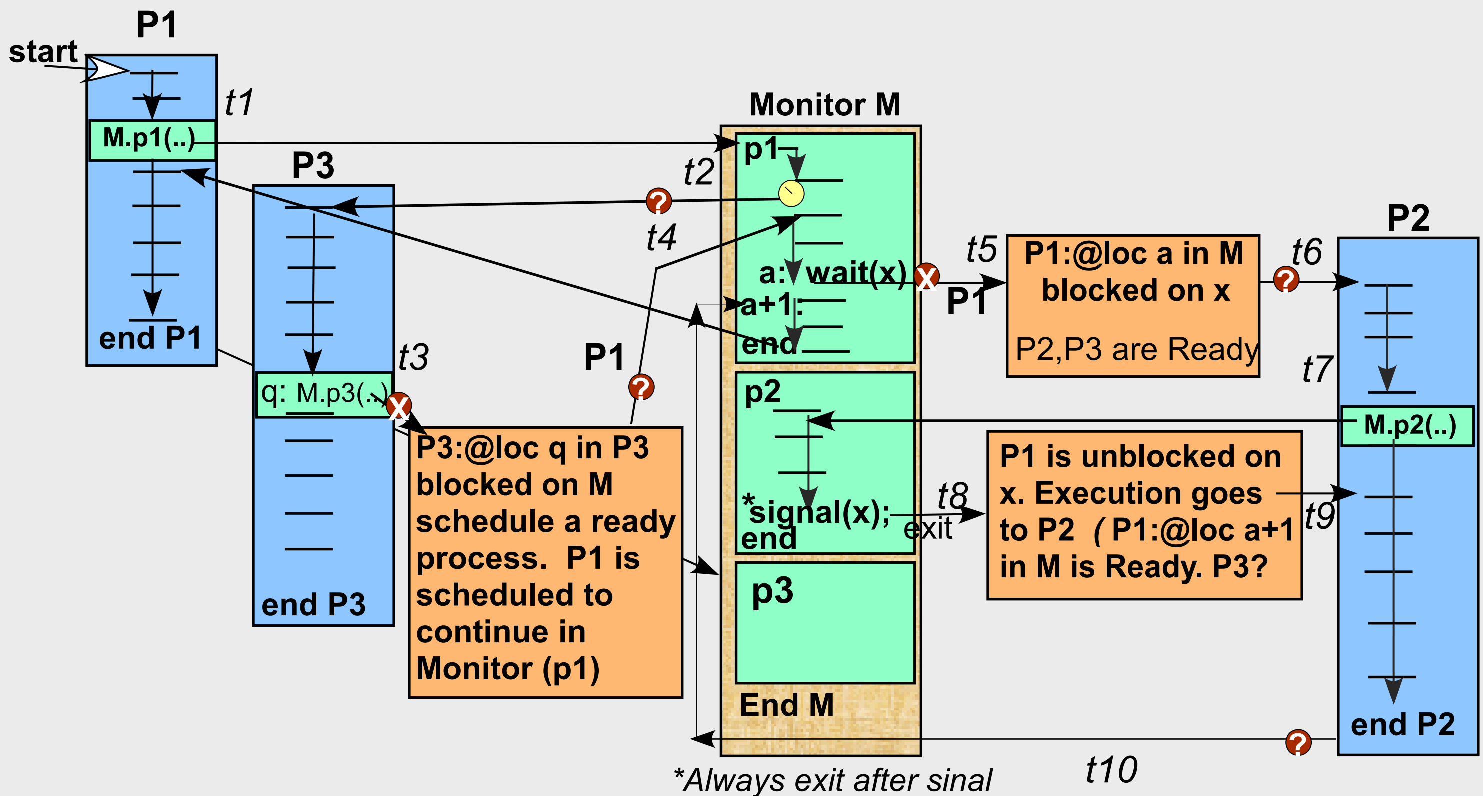
Synchronization
Use condition variables*(*y* here) and two commands:

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 **proc** is no longer active, so another procedure (or in some implementationsm the one with the blocked wait) can be called by another Process..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 Procedure Call: <Monitor Name> **.** **Proc** j (actual arguments)



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 Momitor 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.)

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

© M C Paull 2005

```
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);}
  }
```

```
semaphor 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);
```

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);
```

1 NO GOOD

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);
```

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

```
down(mutex);
if(count==0){ up(mutex); 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);
```

2 BUT NOW NO GOOD

SOLUTION:add cons flag

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

2 OK

TRANSLATION MONITOR TO BINARY SEMAPHORS

PRODUCER,-CONSUMER.Automatic exit on up(condition) **FIXING PROBLEMS-DETAILS**

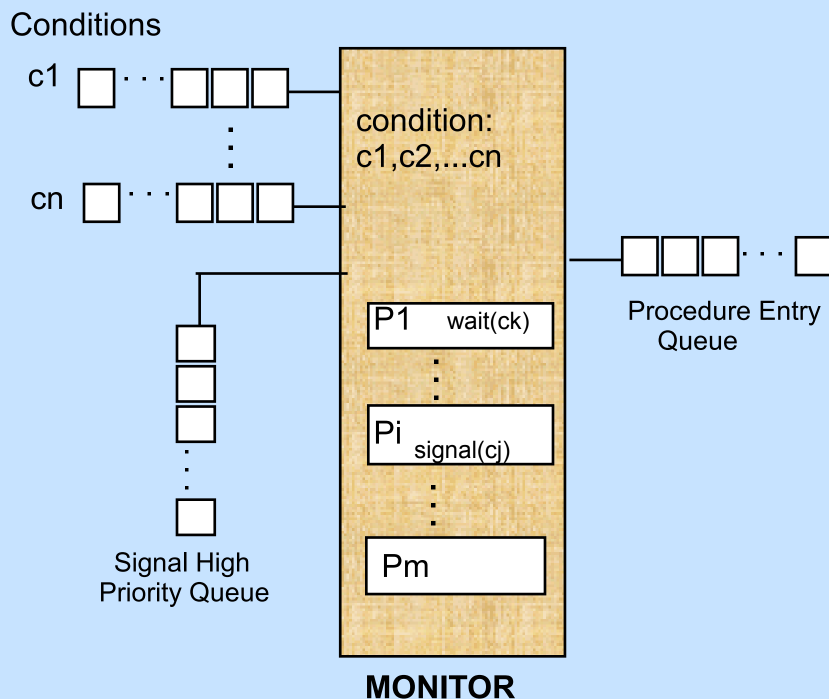
© M C Paull 2005

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 accesible 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 han the simple transformation-there must then be ways to queue processeswaiting to execute shared procedures.

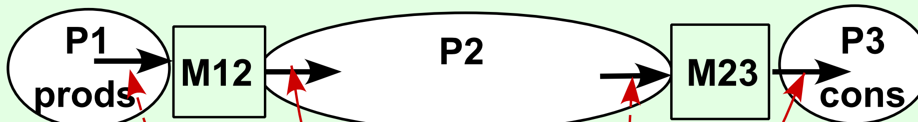


In general a number of different sources will be attempting to get access to the Monitor. When a procedure in the monitor arrives at a wait(c) it is queue up to continue when a signal on that condition, c. occurs. In case the system is one in which any signal terminates a Process that signal will immediately make a process waiting (if there is one) on that condition active.

QUALIFYING THE GIVEN TRANSLATION MONITOR TO BINARY SEMAPHORS

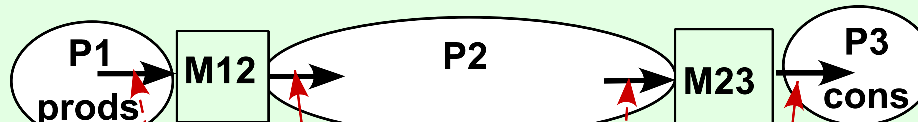
© M C Paull 2005

int M12[N12] int M23[N23]



Must Pass Arguments M12 M12 M23 M23

remove M12
enter M12



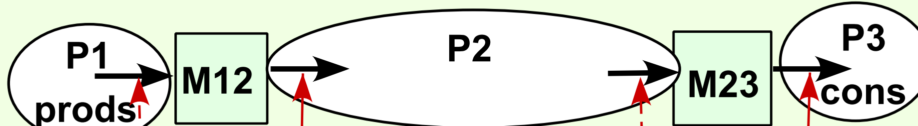
remove M12
enter M12
enter M23
remove M23

Single Monitor

Only One Procedure At A Time Runs In Monitor

SEE NEXT PAGE

firstproducer(term) **take-massage-give()** **finalconsumer(item)**



enter M12
remove M12

enter M23
remove M23

Two Monitors

A Procedure From Each Monitor Can Run In Parallel

SINGLE VS MULTIPLE MONITORS

P1-Producer --> P2-Massage --> P3-Consume

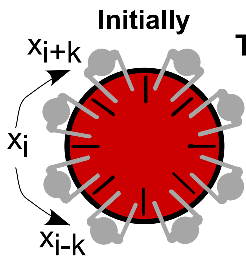
© M C Paull 2005

```
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);}
  End Monitor

 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); }
 End Monitor
```

THE TWO MONITOR CASE

© M C Paull 2005



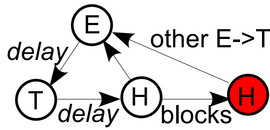
DINING PHILOSOPHERS

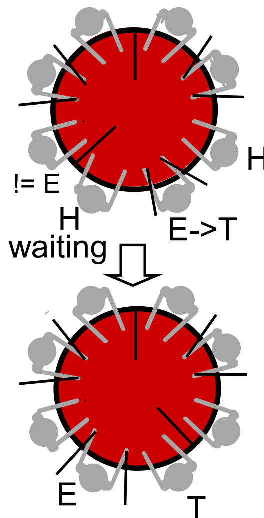
Thinks-Hungry-Gets Chopsticks-Eats In order to eat need left <u>and</u> right chopsticks
i.e., Neither Phlosopher on left or right is Eating

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 i*th* Philosopher is:

```
while(TRUE)
 {  delay
   F.getchopsticks(i);  /*Hungry*possible
   delay*                          wait/
   F.putchopsticks(i);  Think, Other Eat?
  }
```






```
Monitor F
   int i;
   int q(1,...,N) =T; /*state for each initialized to T*/
   cond s(N);         /*condition var for each*/
   int T=1,   /*Thinking*/
   int H=2,   /*Hungry*/
   int E=3;   /*Eating*/
   getchopsticks(i) {
     q(i) = H;    /*q(i) = State of ith Philosopher*/
     if (q(i-1)==E || q(i+1)==E)wait(s(i));
                          /*if either side eating? wait*/
     q(i) = E; }  /*must have chopstick both sides*/
   putchopsticks(i) {
     q(i) = T;
     if (q(i+1)== H && q(i+2 )!= E) signal(s(i+1));
     if (q(i-1) == H && q(i-2 ) != E) signal(s(i-1)); }
 End F
```

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))

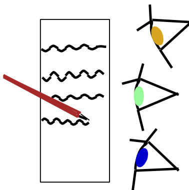
If any reader is active the writer must wait. If Writer is waiting and the <u>number of Readers (rc)</u> goes to 0 the Writer writes. While the writer is writing in try-write() no reading is possible because of the Mutual Exclusion of procedures.

The Process for the i*th* 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();
}
```



```
Monitor RW
    int rc=0; [rc = the number of readers]
    int state=0;
    cond db;

  pre-read()
   { rc = rc+1;}

  post-read()
    { rc = rc-1;
     if (rc==0) signal (db); }

  try_write()
      { if (rc > 0)wait(db);  /*wait if there are any readers */
      write_database();}   /*no one can read*/
  End RW
```

SLEEPING BARBER

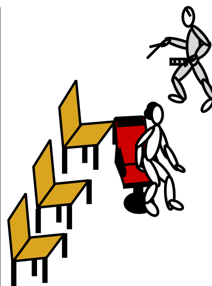
Barber sleeps if no customers otherwise cuts hair of next customers.

The Process for the Barber is:

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

The Process for the i*th*:Customer is

```
       BC.customer;
```



Similar to the Prod_Cons

```
Monitor BC
 int chairs=0;/*chairs = # of customers awaiting haircuts*/
 cond custnum;
 barber_checks()
  { if (chairs==0) wait (custnum)  ;/*wait if no customers */
    chairs = chairs-1; }
 customer()    /*When customer arrives*/
  { if (chairs < N) chairs = chairs +1;      /*if empty chairs */
    if (chairs==1) signal(custnum); }      /*if first chair*/
 End BC
```

CLASSICAL PROBLEMS-MONITOR SOLUTIONS

© M C Paull 2005

```
int q(N)=T;          /*state for each-initially T*/
cond s(N);      /*condition var for each*/
```
SHARED

Initially

The process for the ith philosopher is:
```
while(TRUE){
  Delay
  F.getchopsticks(i);
  Delay
  F.putchopsticks(i);
  }
```
q(i) = State of ith Philosopher

delay, delay, blocks, other E->T (T, H, E, H)

```
while(TRUE){
   Delay
  .getchopsticks(i);
   Delay
  putchopsticks(i);
  }
 getchopsticks(i)
 { down(mutex)
    q(i) = H;
    if (q(i-1)==E || q(i+1)==E)
          {up(mutex) down(s(i); }
    q(i) = E;
   up(mutex) }
 putchopsticks(i)
  { down(mutex)
     q(i) = T;
     if (q(i+1)== H && q(i+2 )!= E) up(s(i+1));
     if (q(i-1) == H && q(i-2 ) != E) up(s(i-1));
    up(mutex) }
```
The Process for the ith Philosopher :

```
Monitor F
  int i;
  int q(N)=T;      /*state for each-initially T*/
  semaphor  s(N);  /*condition var for each*/
  int T=1,       /*Thinking*/
  int H=2,       /*Hungry*/
  int E=3;      /*Eating*/
  getchopsticks(i) {
    q(i) = H;
    if (q(i-1)==E || q(i+1)==E)wait(s(i));
                    /*if either side eating? wait*/
    q(i) = E; }    /*must have chopstick both sides*/
  putchopsticks(i) {
    q(i) = T;
    if (q(i+1)== H && q(i+2 )!= E) signal(s(i+1));
    if (q(i-1) == H && q(i-2 ) != E) signal(s(i-1)); }
End F
```

```
int q(N)=T;      /*state for each-initially T*/
semaphor  s(N);  /*condition var for each*/
int T=1,       /*Thinking*/
int H=2,       /*Hungry*/
int E=3;      /*Eating*/
```

```
while(TRUE){
  Delay
  F.getchopsticks(0);
  Delay
  F.putchopsticks(0);
  }
```

```
while(TRUE){
  Delay
  F.getchopsticks(1);
  Delay
  F.putchopsticks(1);
  }
```

```
while(TRUE){
  Delay
  F.getchopsticks(2);
  Delay
  F.putchopsticks(2);
  }
```

```
while(TRUE){
  Delay
  F.getchopsticks(3);
  Delay
  F.putchopsticks(3);
  }
```

```
while(TRUE){
  Delay
 down(mutex)
  q(0) = H;
  if (q(3)==E || q(1)==E)up(mutex); down(s(0));
  q(0) = E; }
 up(mutex)
 Delay
 down(mutex)
  q(1) = T;
  if (q(1)==H && q(2)!= E) up(s(1));
  if (q(3)==H && q(2)!= E) up(s(3)); };
 up(mutex)
 }
```

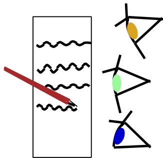
```
while(TRUE){
   Delay
 down(mutex)
   q(2) = H;
    if (q(1)==E || q(3)==E) up(mutex); down(s(2));
    q(2) = E; }
 up(mutex)
   Delay
  down(mutex)
    q(2) = T;
    if (q(3)==H && q(0)!= E) up(s3));
    if (q(1)==H && q(0)!= E) up(s1)); };
  up(mutex)
   }
```
Example: Semaphor Compilation For 4 Philosophers

DINING PHILOSOPHERS

SEMAPHOR SOLUTION BY COMPILATION OF MONITOR SOLUTION

© M C Paull 2005

MANY READERS, ONE WRITER



The Process for the *i* th reader is

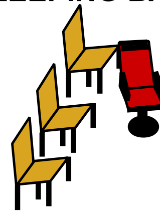
```
while(TRUE) {
   RW.pre_read();
   read_database();
   RW.post_read();
   use_data();  }
```

The writer Process

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

```
Monitor RW
  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
```

SLEEPING BARBER.



The Process for the barber is:

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

The Process for the *i* th customer is BC.customer;

```
Monitor BC
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
The bodies of the Procedures have replaced the Procedure Calls

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

For each readers the code is

```
 int wr = 0;
 while(TRUE)
 {down(mutex1)
   rc = rc+1;
 up(mutex)
  read_database();
 down(mutex1)
    rc = rc-1;
    if (rc==0 && wr ==1) {wr = 0; up(db); goto X;}  }
 up{mutex1}
X:use_data();
}
```

The writers code is:

```
while(TRUE)
 { produce_data();
   down{mutex1}
   if (rc > 0) { wr = 1; up(mutex1); down(db);}
   write_database();
   up(mutex1);
 }
```

```
int chairs=0;
semaphor mutex2=1, custnum=0;
```

The Barber code is:

```
while(TRUE)
  { down(mutex2)
     if (chairs==0) { barb=1;up(mutex2);down (custnum); }
     chairs = chairs-1;
    up(mutex2);
    cut_hair();
  }
```

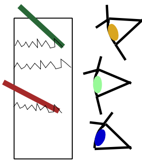
For each customer the code is

```
while(TRUE)
  {  down(mutex2);
       if (chairs<N)chairs = chairs+1;
       if (chairs==1&&barb=1) { barb=0; up(custnum);
     up(mutex2);
  }
```

SHARED MEMORY

MANY READERS ONE WRITER , SLEEPING BARBER
SEMAPHOR SOLUTION BY COMPILATION OF MONITOR SOLUTION

© 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

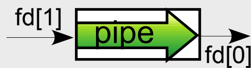
```
while(TRUE){
   produce_data();
   RW.try_write2(); }
```

```
Monitor RW
   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
```

MANY READERS TWO WRITERS

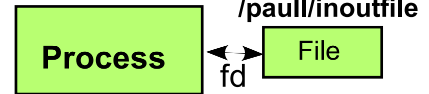
© M C Paull 2005

```
#include <unlistd.h>

int fd[2];
char buf[7];

pipe(fd); /*open-no name*/
write(fd[1], “test it”, 7);
read(fd[0], buf, 7);
```



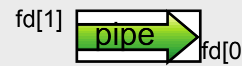
PIPE In SINGLE PROCESS

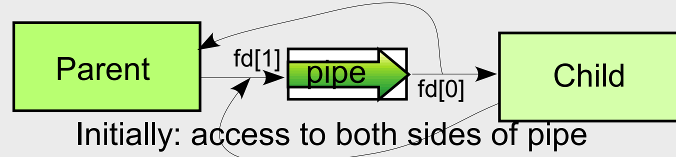
```
int fd;
char buf[7];

fd=open(/paull/inoutfile,RWONLY)  /*open by name*/
write(fd, “test it”, 7);
read(fd, buf, 7);
```



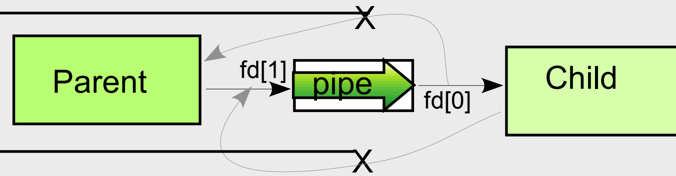
FILE IN SINGLE PROCESS

```
#include “ourhdr.h”
#define MAXLINE 1024
int main(void);
{ int fd[2];
  pid_t pid;                              [Before Fork]
  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);
}
```



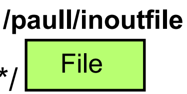


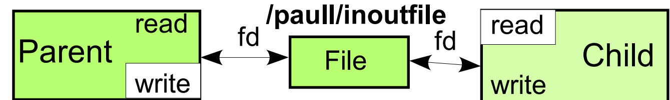
Initially: access to both sides of pipe



Inter Process Communication Through A Pipe

```
#include “ourhdr.h”
int main(void);
{ pid_t pid;
  char line[MAXLINE};              [Before Fork]
  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);
}
```



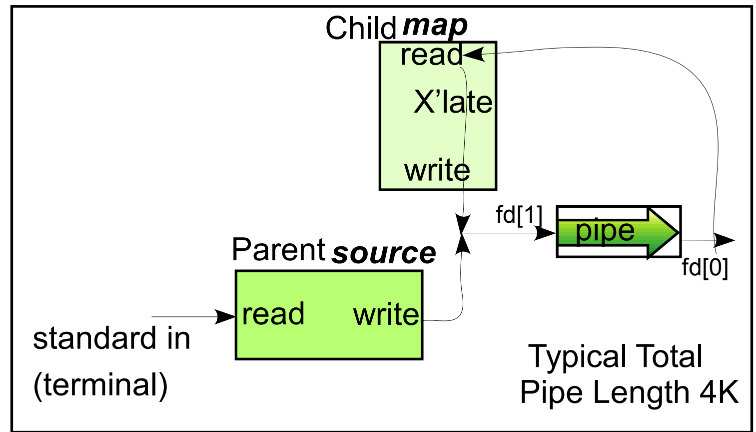


Inter Process Communication Through Shared File

COMMUNICATION: PIPES (1-Way) Like Files

© M C Paull 2005

```
#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
                                   0 =standard input*/
   case -1
     perror("fork");
  }
 wait((int*)0);
 n=read( fd[0], buf, BUFSIZE)
 if(n > 0) { buf[n]='\0'; printf("recieved %s \n", buf); }
}
```




```
source(int readdfd, 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(readdfd, buf, BUFSIZE)
    if(n  > 0)
       {  buf[n]=0; printf("got \ " %s\" \n", buf);
          for(i=0; T[i].left; ++1)                 /* 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

PIPES: (Anonnymous and Named)
Inorder to be availlable to two or more processes a pipe, as in the previous UNIX programs, must be declared before a fork. Then it is available in the parent and child resulting from the fork. Unlike a file, the pipe, in this case, does not have a name and its lifetime is limited to the runtime of those forked processes. Such a pipe is called an **anonymous pipe**. There are also **named pipes** which can be defined with a name like a file and be widely accesible to Processes, for a lifetime which lasts beyond a single run, in fact until they are explicitly removed.

The implementation of anonymous pipes with their limited lifetime can use storage MM, or they can be implemented like a file using the disk. In the latter case, since the typical pipe size is 4k which is also the size of a block in the cache for file blocks. So the pipe, when first used, will be in the MM *block cache* and can be kept there for its lifetime.

MESSAGE PASSING:
Message Passing in the shared memory environment is very similar to pipes. It deals with a bounded sizes-of data per entry and maximum number of such entries, whereas pipes deal only with a bound on the total amount of data which can accumulate without removal.

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:

receive(*from_process*, local_message_origin (buffer)), and
send (*to_process*, local_message_destination (buffer))**,**

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-b**locking till success (as opposed to **asynchronous** continue whether sucessfull or not, with some test in a returned value for failure or sucess 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 recieve 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.

© M C Paull 2005

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 s 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 **Rendevous**, in which, effectively, the Mailbox has room for only one message. In the case of Rendevous 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 ($\boldsymbol{\varepsilon}_s$) based on the number that can be produced in a quanta (and consumed).

Process schema and traces using this $\boldsymbol{\varepsilon}_s$ scheme for each of alternatives 1 and 2, are given below.

Circular Buffer Again (Like Multiple Alternators)

The send and recieve 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(maibox_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

Sender

```
put(m_s, msg_s)
send( msg_s, consumer,)
```

The message information is in m_s (local) other information is added by put to form the message msg_s before it can be sent. then sent to an OS mailbox by the send command.

Receiver

```
receive(producer,msg_r)
get(msg_r, m_r)
```

The message is moved to msg_*r* in the mailbox by the receive operation and then stripped of protocol and placed in m_r (local) by get

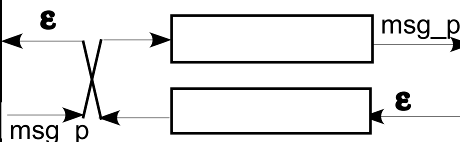
ASSUMED USAGE

PRODUCER (Sender)

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

2 Mailboxes
1 for each direction or
or 1 for each communcator

Basics



CONSUMER (Receiver

```
int m_ε = ε;            N messages will be
put(m_ε, msg_c)         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)
 }
```

USED FOR PRODUCER CONSUMER PROBLEM

*ε 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

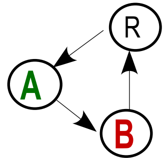
PRODUCER (Sender)

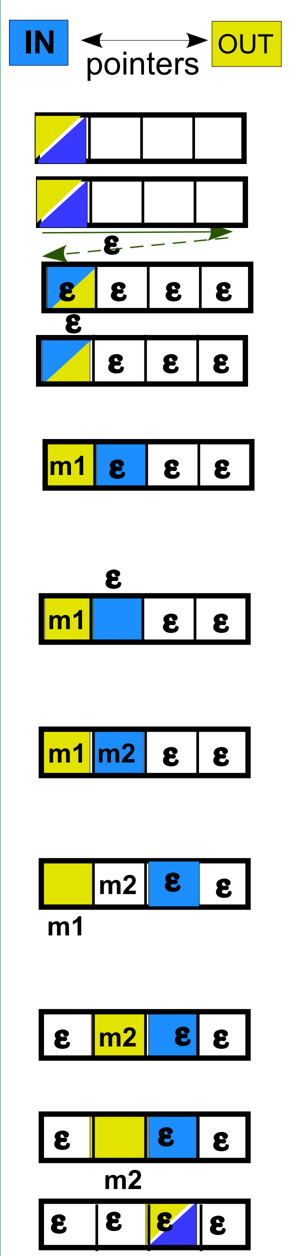
```
while(TRUE)
 { produce(m_p);
   receive( m_c,consumer);
   put(m_p, msg_p)
   send(consumer, msg_p);
 }
```

CONSUMER (Receiver

```
int m_ε = ε;
put(m_ε, msg_c)
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)
 }
```

N messages will be accommodated.





R
A
B

STATE

Producer Process

produce(m_p);

receive(m_c,consumer);

receive(m_c,consumer)

put(m_p, msg_p)

send(consumer, msg_p);

produce(m_p);

receive(m_c,consumer);

put(m_p, msg_p)

send(consumer, msg_p);

Consumer Process

int m_ε = ε;
put(m_ε, msg_c)
for(i=0,i<N,i++)
{send(producer,msg_c)
receive(msg_p, producer

receive(msg_p, producer);

get(msg_p,m_p);

put(m_ε, msg_c)

send(producer, msg_c);

consume(m_p)

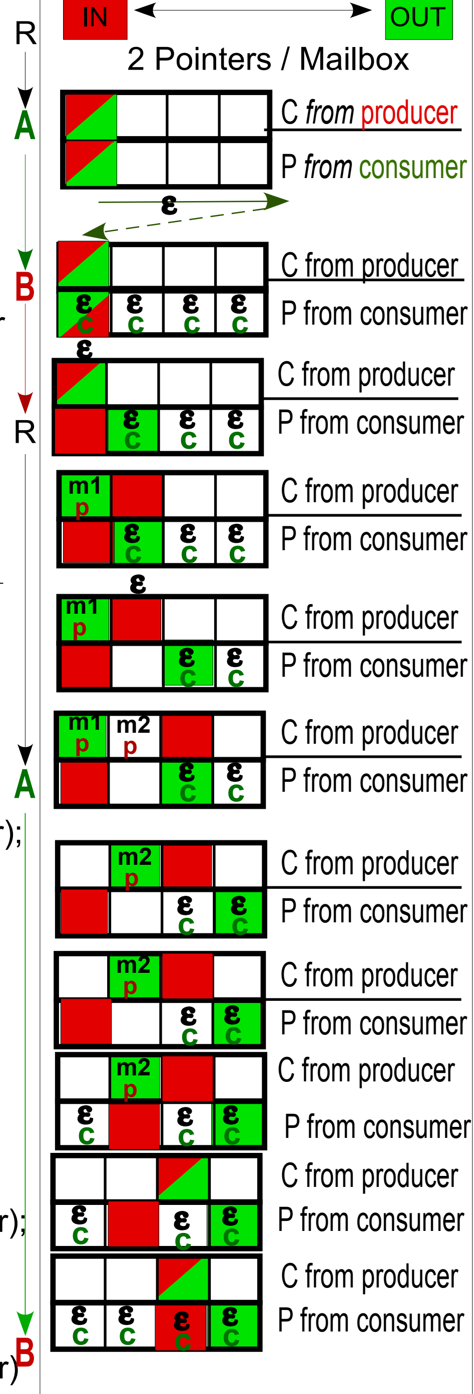
receive(msg_p, producer);

get(msg_p,m_p);

put(m_ε, msg_p,producer)

STATE

Producer may run after Consumer has sent < N εs
N is still the largest Number of messages Producer can lead Consumer by



MESSAGE PASSING COMMUNICATION BUFFER AND MAILBOXES TRACE

Physical Buffer Equal The Number Of εs

© M C Paull 2005

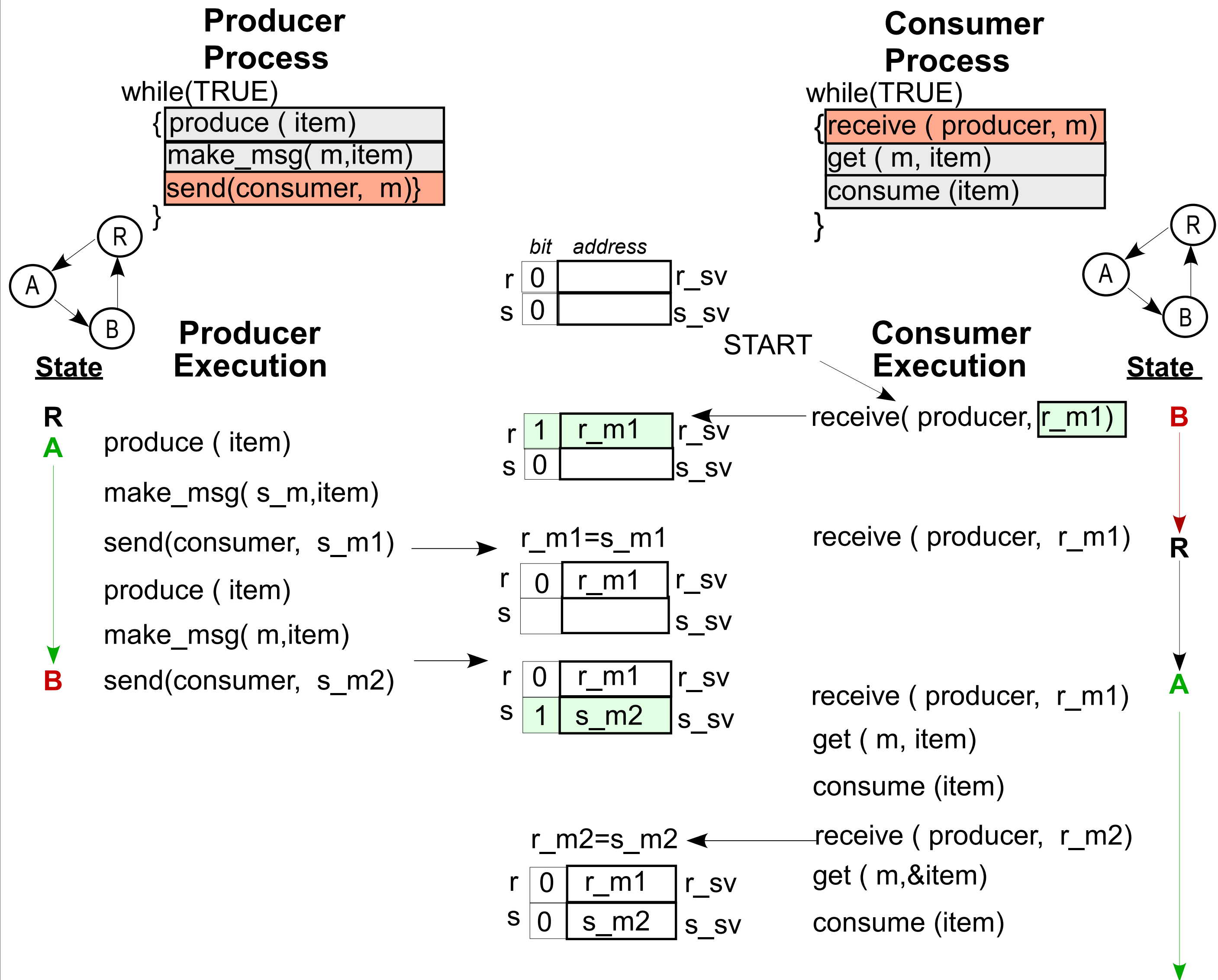
Rendevous involves A producer Process that sends, consume Process that receives;
int s=0;r=0; /*r=1 indicates a receive has been executed but the corresponding send has not,
s=1 indicates a send has been executed but the corresponding receive has not.*/

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; s_sv=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; r_sv=location at which
           receiver gets  msg; block;}
```



MESSAGE PASSING BY RENDEZVOUS

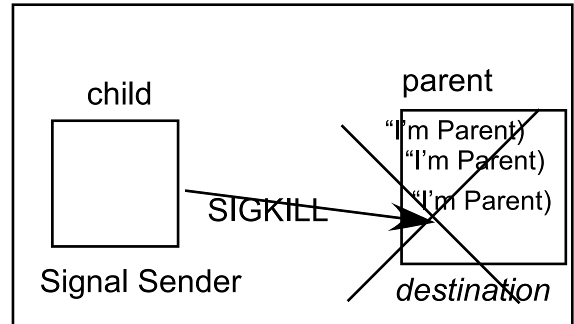
© M C Paull 2005

kill(int pid, int signal_id)

<pid> indicates destination of signal
0 indicates all processes in group
-1 indicates all processes with the same user ID

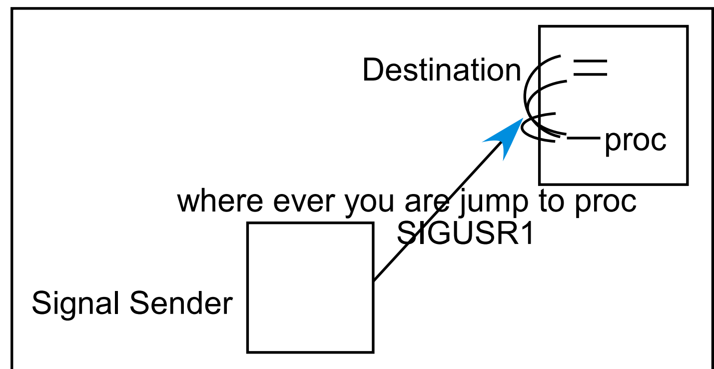
signal_id
Two examples
1 signal_id = SIGKILL = :kill destination
2 signal_id= SIGUSR! = causes the *destination* to run a function it (the *destination*) specifies in a **signal** command at the *desination* which looks like:

signal(SIGUSR1,<name of procedure called in *destination* Process >*)




```
#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                              /*getpid()   returns my process ID
     { sleep(5);                    getppid()  returns my parents ID
       kill(getppid(), SIGKILL);    getpgrp( ) returns group ID        */
     }
  }
```

```
#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);                 /*exit(n) process exits and returns a value n */
       }
```



© M C Paull 2005

```
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);}
 }
```

```
semaphor mutex = 1,  bfull = 0, bempty = 0;

int shared count = 0, N = 40;
```

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

Consumer

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

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

Producer

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

1 NO GOOD

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

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

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

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

1 OK

No Problem on count==1

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

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

2 OK

TRANSLATION MONITOR TO BINARY SEMAPHORS PRODUCER,-CONSUMER.
No automatic exit on signal **FIXING PROBLEMS-DETAILS**

© M C Paull 2007

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

```
int m_ε = ε;                          N messages will be
put(m_ε, msg_c)                       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)
```

ε msg_p msg_p ε

P-reads ε –>, writes msg

C-reads msg -->

C-writes ε -->

2 Mailboxes IN OUT

2 Pointers / Mailbox

C from producer

P from consumer

Producer Process **Consumer Process**

STATE R A B STATE R A B

Producer	Consumer
R produce(m_p);	R
A receive(m_c,consumer); B →	int m_ε = ε; put(m_ε, msg_c) for(i=0,i<N,i++) {send(producer,msg_c)
A receive(m_c,consumer) ←	receive(msg_p, producer B
put(m_p, msg_p)	
send(consumer, msg_p); →	R
produce(m_p);	
receive(m_c,consumer); ←	
put(m_p, msg_p)	
send(consumer, msg_p); → R	receive(msg_p, producer); A
	get(msg_p,m_p);
	put(m_ε, msg_c)
	send(producer, msg_c);
	consume(m_p)
	receive(msg_p, producer);
	get(msg_p,m_p);
	put(m_ε, msg_p,producer) B

Producer may run after Consumer has sent < N εs
N is still the largest Number of messages Producer can lead Consumer by

MESSAGE PASSING COMMUNICATION BUFFER AND MAILBOXES TRACE

Physical Buffer Larger Than Number Of **ε** s

© M C Paull 2005