THE PROBLEM: NEED FOR MUTUAL EXCLUSION AND OR SYNCHRONIZATION

2 INTER PROCESS COMMUNICATION THROUGH SHARED MEMORY-PROBLEM
3 RACE CONDITIONS-EXAMPLES
4 MUTUAL EXCUSION & SYNCHRONIZATION (LOCKNG MEMORY)

SOLUTION: BUSY WAITNG SPECIAL INSTRUCTION TSL FOR MUTUAL EXCLUSION

5 BUSY WAITING (BW) STRICT ALTERNATION SWITCHING MODEL:
Two Approaches
6 BW-GENERAL PROTECT CRITICAL REGION - TSL MULTIPLE PROCESSES.

CIRCULAR BUFFER FOR SYNCHRONIZATION

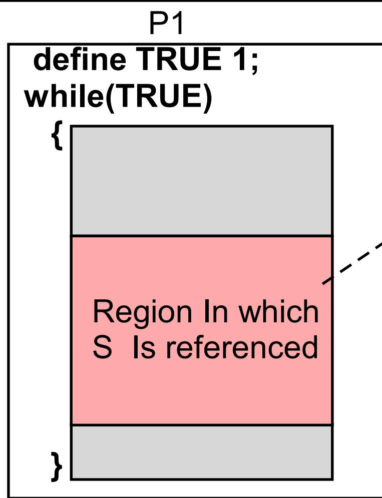
7 BW-PROTECT CRITICAL REGION MULTIPLE ALTERNATORS(CIRCULAR BUFFER)
Two Processes For Synchronization In Producer-Consumer Applications
8 BW- COUNTING INs & OUTs MULTIPLE ALTERNATORS FOR SYNCHRONIZATION
In Producer-Consumer Applications
9 MULTIPLE ALTERNATORS CAN BE A BASIS FOR MESSAGE PASSING
With one Mailbox per Sender-Reciever Pair
10 MULTIPLE ALTERNATORS-MULTIPLE USERS (>2) MARKING SHARED MEMORY
Imperfect

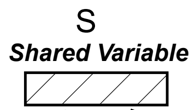
PETERSON'S ALGORITHM BUSY WAITING MUTUAL EXCLUSION

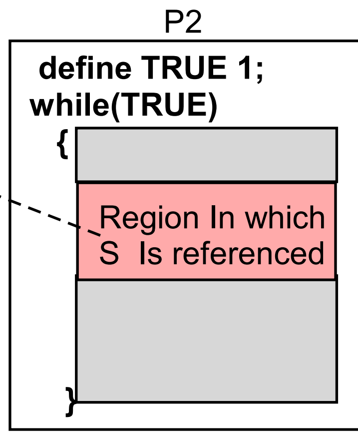
11 BW-GENERAL CRITICAL REGION PROTECTION - PETERSON'S ALGORITHM
FOR TWO PROCESSES
12 PETERSON'S ALGORITHM TWO PROCESS: SWITCHING MODEL
13 PETERSON'S ALGORITHM TWO PROCESS: SWITCHING MODEL TRACE
14 PETERSON'S ALGORITHM THREE PROCESS: SWITCHING MODEL
15 PETERSON'S ALGORITHM 4 Process SWITCHING MODEL

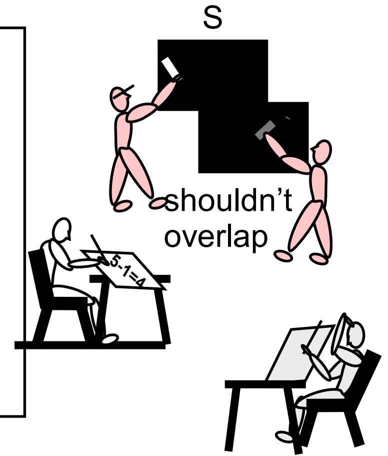
CONTENTS

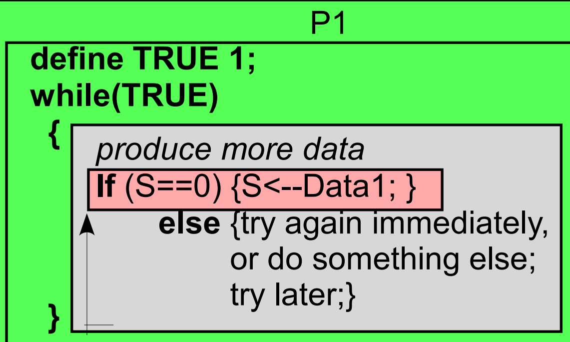
© M C Paull 2005

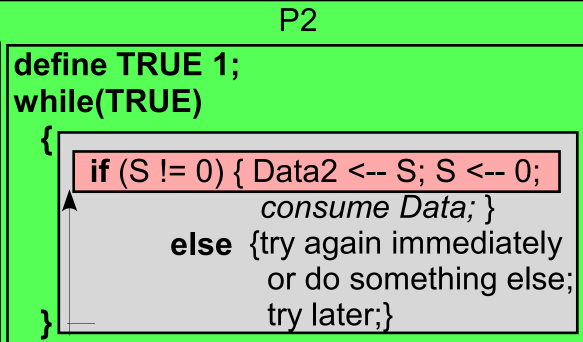


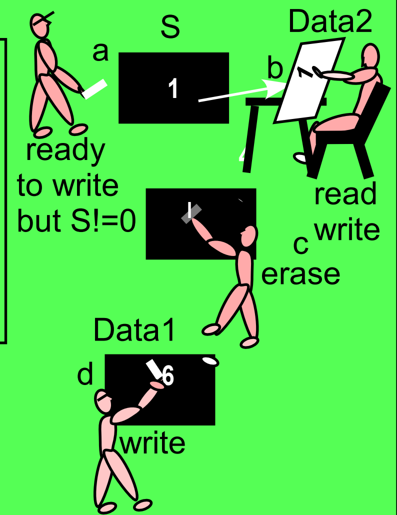






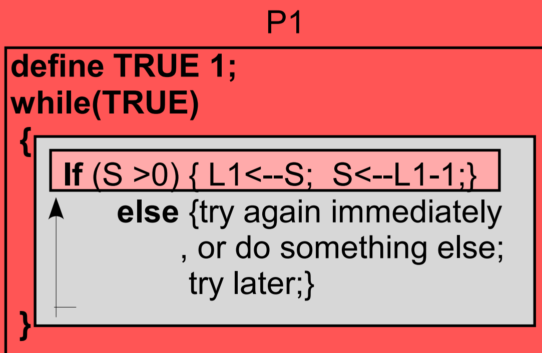


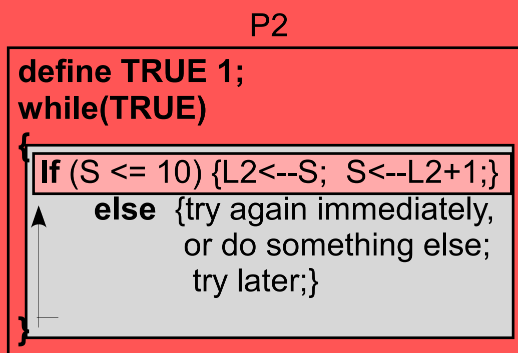


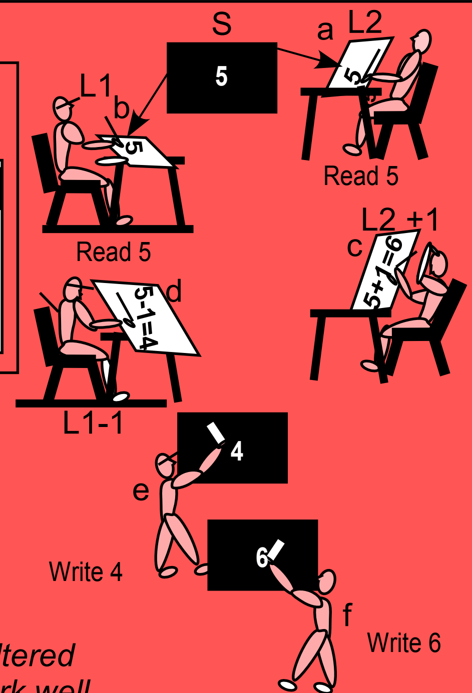


INTENT: P1 puts data in shared memory, but only if it is empty. P2 removes data from S, but only if it is not empty. This does not present a problem since, though both can read and test S in any order it will either be empty so P1 can continue or full so P2 can continue. As long as emptying/filling S can be done in one instruction or, if more than one, done completely, then no problem should result.

No problem because each signals when done and the other can proceed







INTENT: To keep track of S the difference in the number of times P2 and P1 run and assuring that difference will never exceed 10, nor be less than 0. *A problem arrises because a complete addition/subtraction requires 2 instructions moving S to a local location and adding/subtracting 1.* Since OS may assign CPU to Processes in any order for any durations,If this is done in the order illustrated on the right unintended results occur.

Both need to read S (shared) then do internal work on Sbefore returning altered value to S. If the entire set of commandsto do this were atomic it would work well, but as long as eitheris atomic problems can arrise

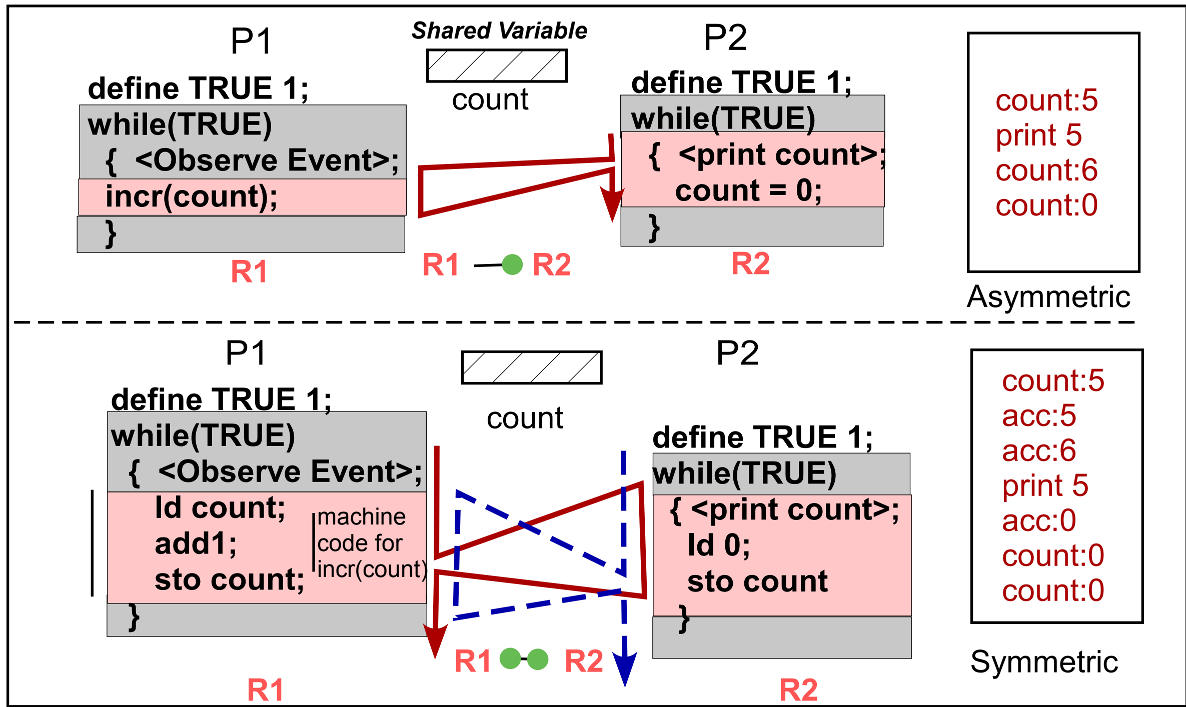
INTER PROCESS COMMUNICATION THROUGH SHARED MEMORY-PROBLEM

© M C Paull 2007

Each **line** in the critical areas is assumed to be **atomic**, i.e., once it starts it completes-no other can go

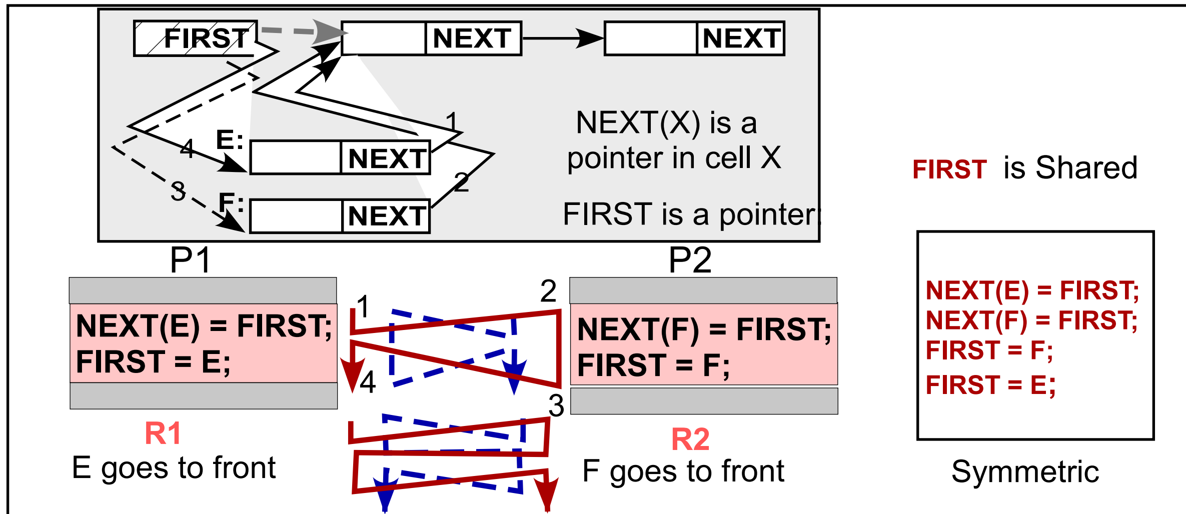
INTENTION

P1 Counts Events
P2 prints current count since its last Print of count and setting count to 0



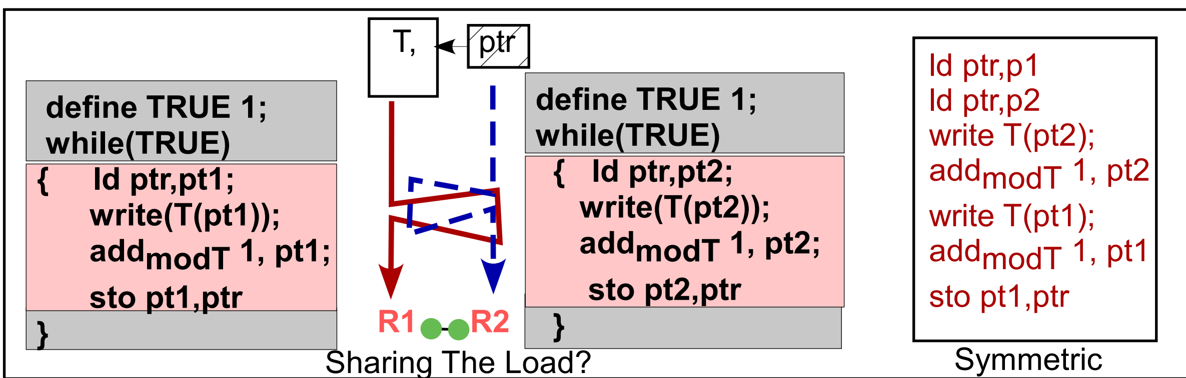
INTENTION

Each Process independently Adds a Cell to beginning of link list



INTENTION

P1 & P2 independently writes to table at shared index & increment index.



CRITICAL REGION

Regions R1 and R2, each in a different Process, are *CRITICAL REGIONS* with respect to one another. To avoid unintended results: once execution of R1 begins it must complete before R2 is entered indicated above by (R1●—R2) and/or visa-versa (R1—●R2) .Mostly "and" is true (R1●-● R2) and we say the executions must be *MUTUALLY EXCLUSIVE* execution of other regions in Processes, not CRITICAL with respect to either R1 or R2, may run whole R1 and/or R2 are in their critical regionsis allowed.)

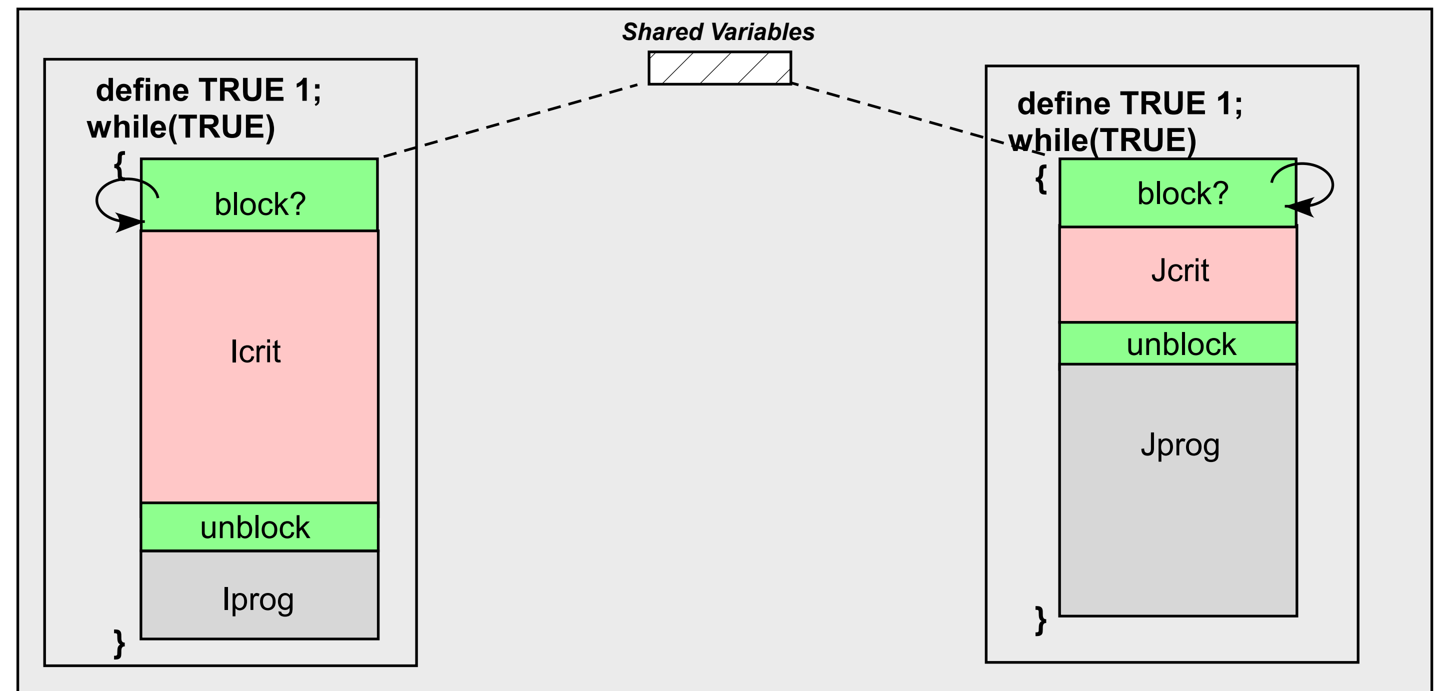
RACE CONDITIONS-EXAMPLES

© M C Paull 2005

What was needed in the previous page was **Mutual Exclusion:**
Two regions, R1 in process P1, and R2 in process P2, are mutually exclusive iff, once either R1/R2 is entered, the other, R2/R1 will not be entered (will be blocked on attempt to enter- will wait at the entry) until the R1/R2 is vacated.

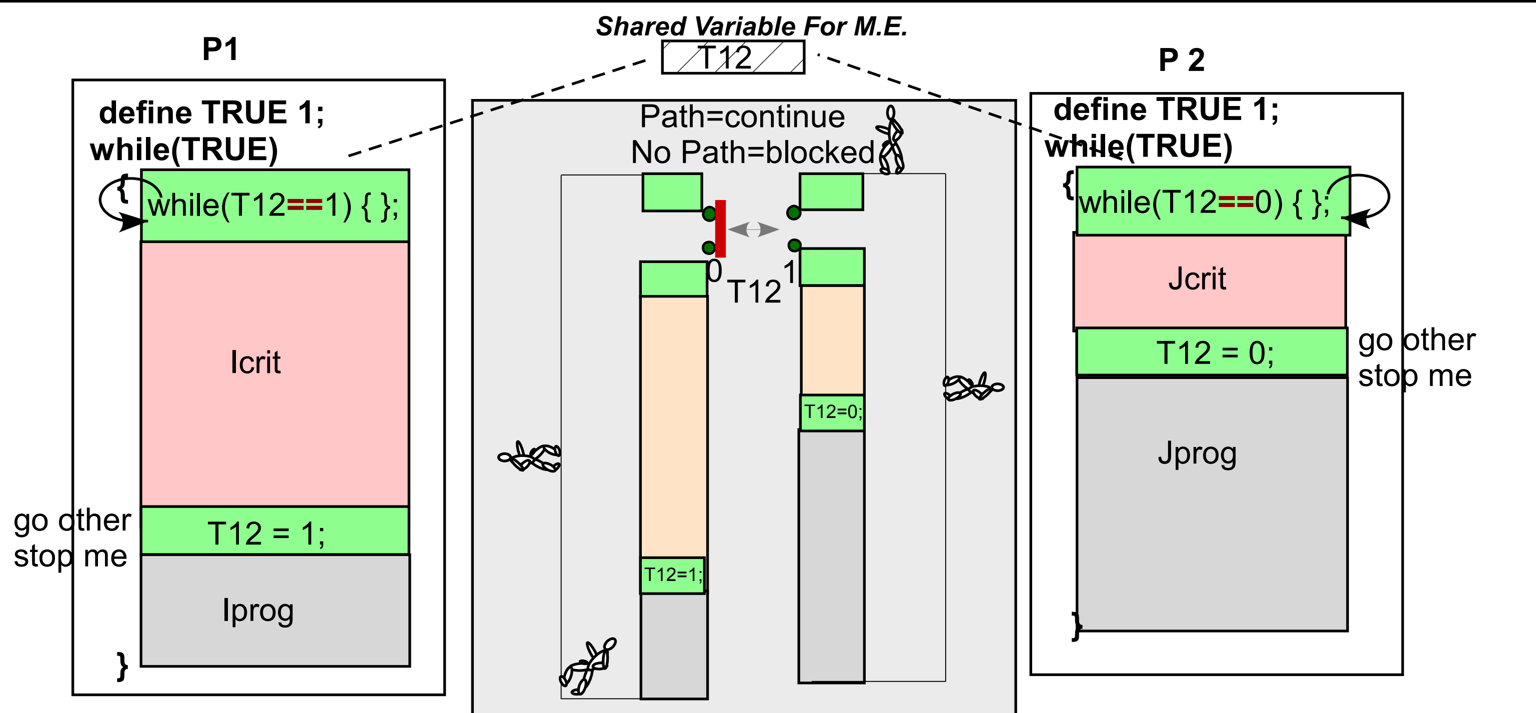
Mutual Exclusion, Sychronization, and Critical Regions

Consider two regions R1 and R2 found respectively in the two processes P1 and P2. Both regions access one or more shared variable(s) *Shared*. *Shared* may be in one of a number of states. The code in R1 and R2 serve to change the state of *Shared*. To change *Shared* it is necessary to read, alter, and write one or more of the variables. So the change of state requires passage through a number stages. Assume R1 starts such a state change. Assume R2 is entered or writes, or makes a state change in *Shared* before the final stage of the state change in R1 is complete. This will generally make the state change(s) invalid. If it does then R1 and R2 are Critical with respect to one another, and must be made **Mutually Exclusive-**R2 must be blocked from its Critical Region while R1 is in it, and visa-versa. [Note **another approach** to achieving atomic use of memory locations involves “a Process getting a **Lock on memory** locations” which then cannot be used by other Processes until that Lock is removed and gotten by another. This approach is used for exclusion with Files-for MM it requires appropriate hardware.]
There are also situations in which entry into a region R1 will cause trouble, independent of the state of any Process sharing memory with R1,. For example: P1 runs R1 to make entries in Shared Table, S, It must not enter R1 if S is full. R1 must be prevented from running, i.e it must be **blocked** until S’s contents have been reduced. To reduce the number of entries S another process R2 with access to S must remove entries from S. In fact, a number of Processes may be blocked on S and it is possible that they can be unblocked by a single process which detects the state of S which cause blocking and unblocks those processes (it may also change the contents of S). Similarly a process which is emptying a shared buffer must block if there is no entry in the buffer.
This dependence of regions in different processes accessing the same shared memory in whjch R1 conditionally blocks and depends on R2 for relieving the blocking condition also makes R1 and R2 **Critical** with respect to one another.and must therefore be **Synchronized**.
Continuous CRITICAL subregions R1 and R2 are optimally sized (not bigger nor smaller than necessary) if making them mutually exclusive/synchronized will make them safe (run as intended) and decreasing their size (either R1 or R2) will make them unsafe..

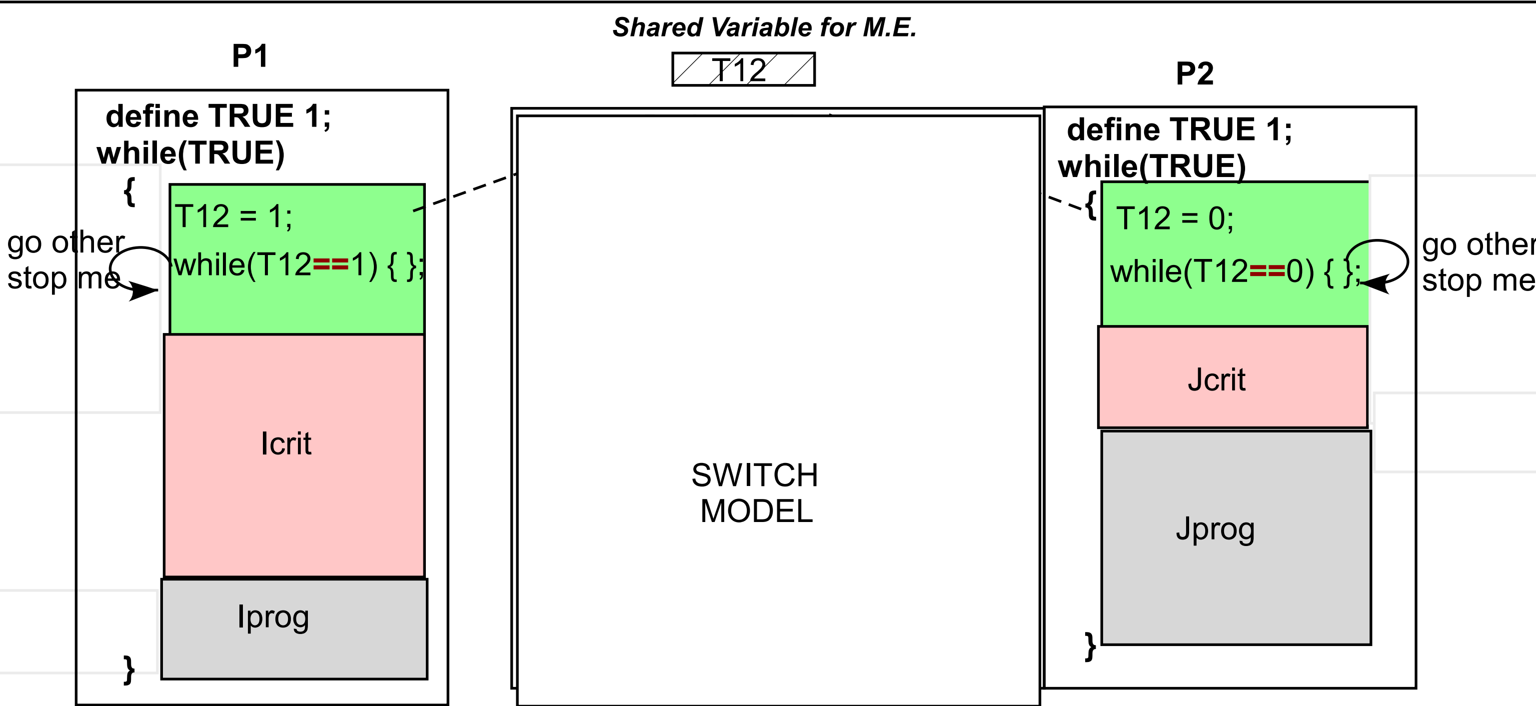


MUTUAL EXCLUSION & SYNCHRONIZATION(LOCKNG MEMORY)

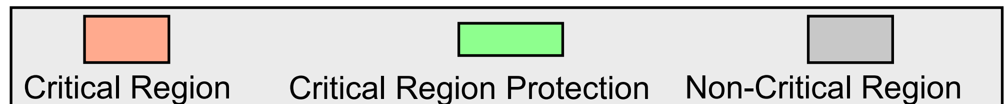
© M C Paull 2007



As soon as Process P1/2 leaves its Icrit/Jcrit Process P2/1 is free to enter its Jcrit/Icrit.
(*If either leaves or loops in its Critical Region the other will be unable to enter its Critical Region*)
(Notice Pi can block itself for Synchronization until the other runs by setting T12 to block itself.)

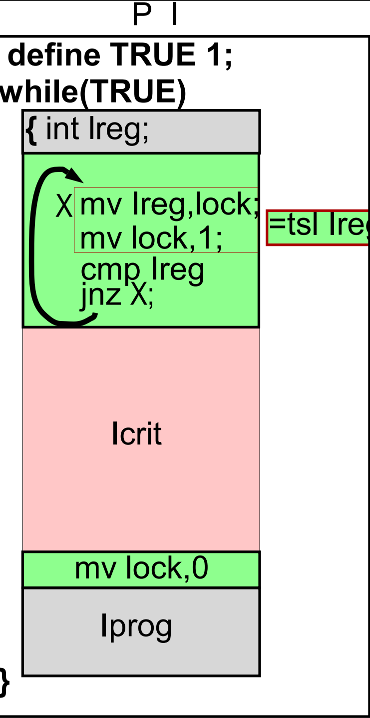


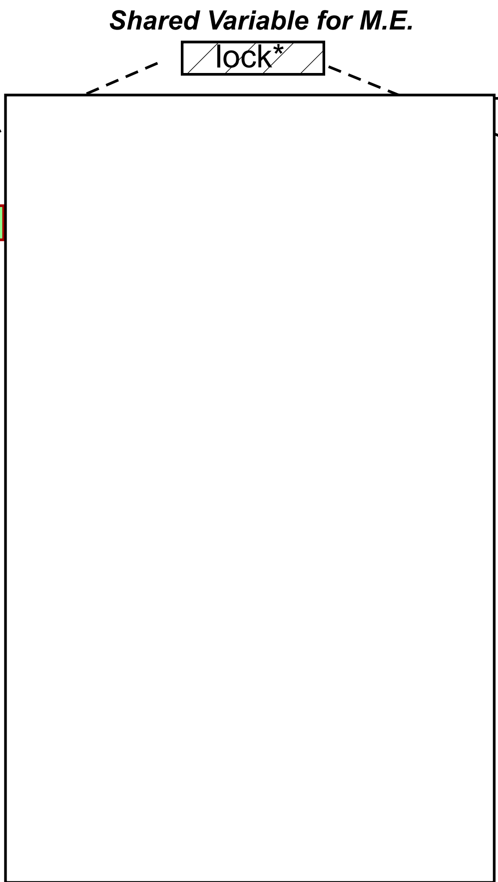
When Process 1/2 enters Green (Region of Desire) it cannot enter Icrit/Jcrit until Process 2/1 enters Green
(*If either leaves or loops in its Critical or Prog Region the other will be unable to enter its Critical Region*)
What alternates is a single pass through each of the processes
To Solve The Problem More Generally Pk should be able to keep entering as long as Pp does not wish to.

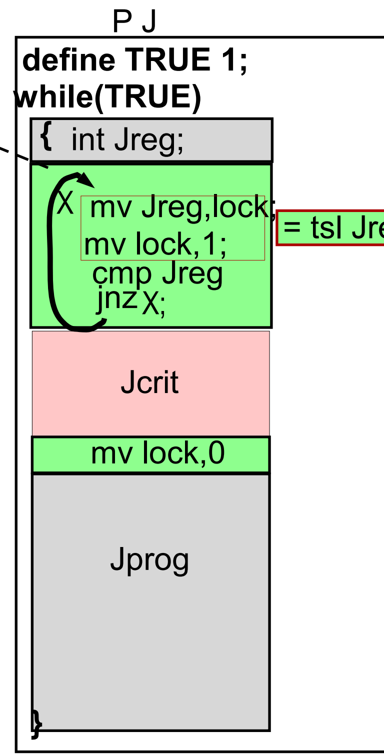


BUSY WAITING (BW) STRICT ALTERNATION SWITCHING MODEL
Two Approaches

© 2005 M C Paull





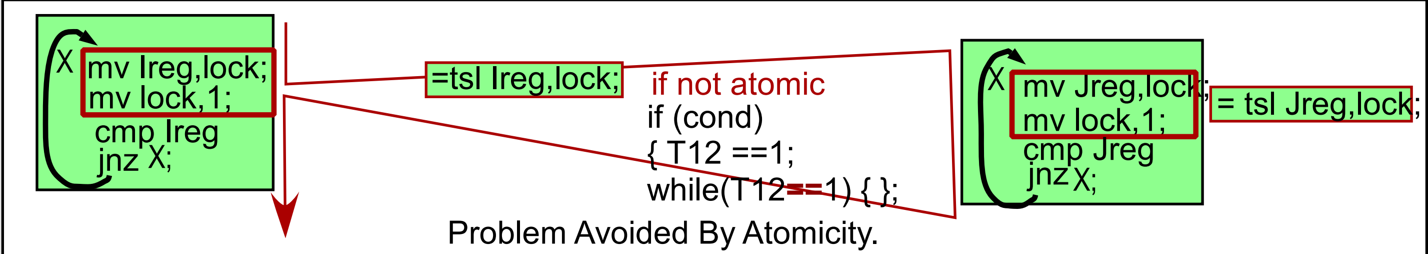


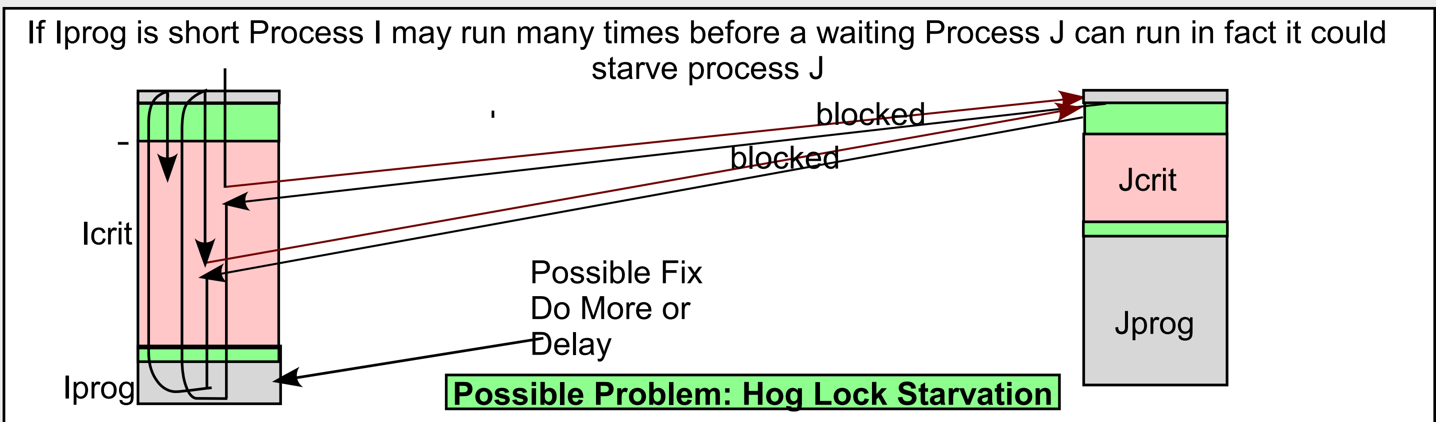
*[Locked If 1
Unlocked if 0]

mv Ireg,lock; mv lock,1; =	Ireg <--- lock lock <--- 1

TSL
Read lock to local & make lock=1 in 1atomic (indivisable) action

Immediately after a Process executes the tsl, the lock is 1, whether it was 0 or 1 before But the local register (Ireg, Jreg) has the value the lock was just before the tsl, and that is what is tested immediately after the tsl, busy waiting if it was 1, and entering its critical region if it was 0.

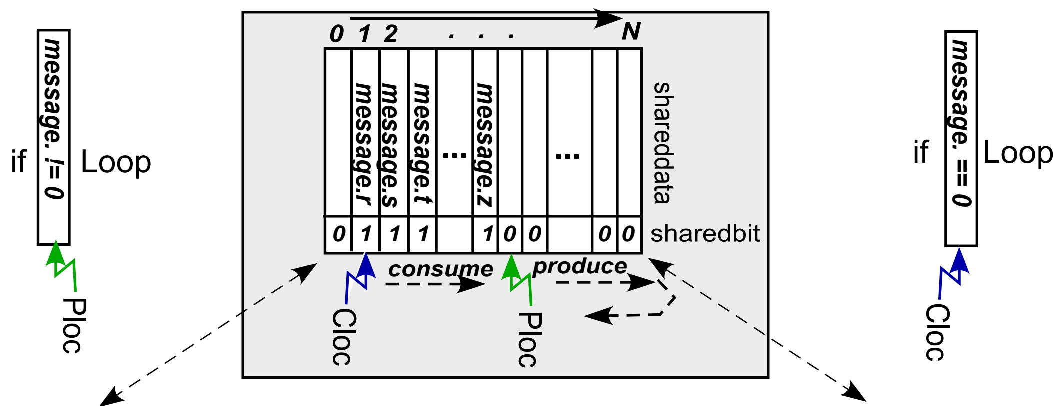




BW=Busy Waiting: During the time a Process Busy Waits the CPU is effectively idle.

BW-GENERAL CRITICAL REGION PROTECTION - TSL MULTIPLE PROCESSES.
(Handles > 2 processes easily and a process can block itself till another runs, i.e Synchronize)

© 2005 M C Paull




```
Ploc =0;    /*ptr initial position*/
while(True)
{ m = Produce(msg)
   while(sharedbit [Ploc] ==1){};
   shareddata[Ploc]=m;
   sharedbit[Ploc] = 1;
   Ploc = [Ploc+1]modN;
}
```

Producer[Send]

```
Cloc =0;    /*ptr initial position*/
while(True)
 { while(sharedbit [Cloc] ==0){};
    m = shareddata[Cloc];
    sharedbit[Cloc] = 0;
     Cloc =[Cloc+1]modN;}
     Consume(m)
  }
```

Consumer[Receive]

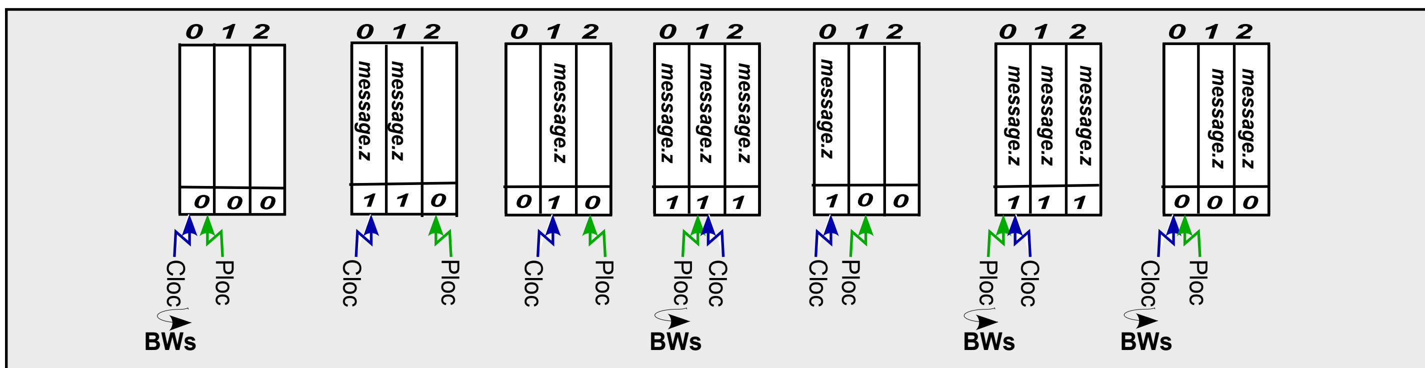
A __busy waiting__ loop is only entered when the shared table is **completely full** or **completely empty**, **Busy waiting is only needed for __SYNCHRONIZING__**. If ***N*** is large this will be infrequent. Both processes can read the same cell, but only one can write that cell as determined by whether it is marked 0 or 1.

Elaborations

1) In general this approach requires a unique mark for each Process in each cell. Any reading Process can specifiy which individual Process is to write at the location it has just read by writing its mark., and vice-versa.a

2) The messages in the cells of the buffer could be a pointer (P) together with a size (S), P being a pointer to a memory area of size S. In this way each fixed size buffer cell can identify any size memory area carrying information from *Producer* to *Consumer.*

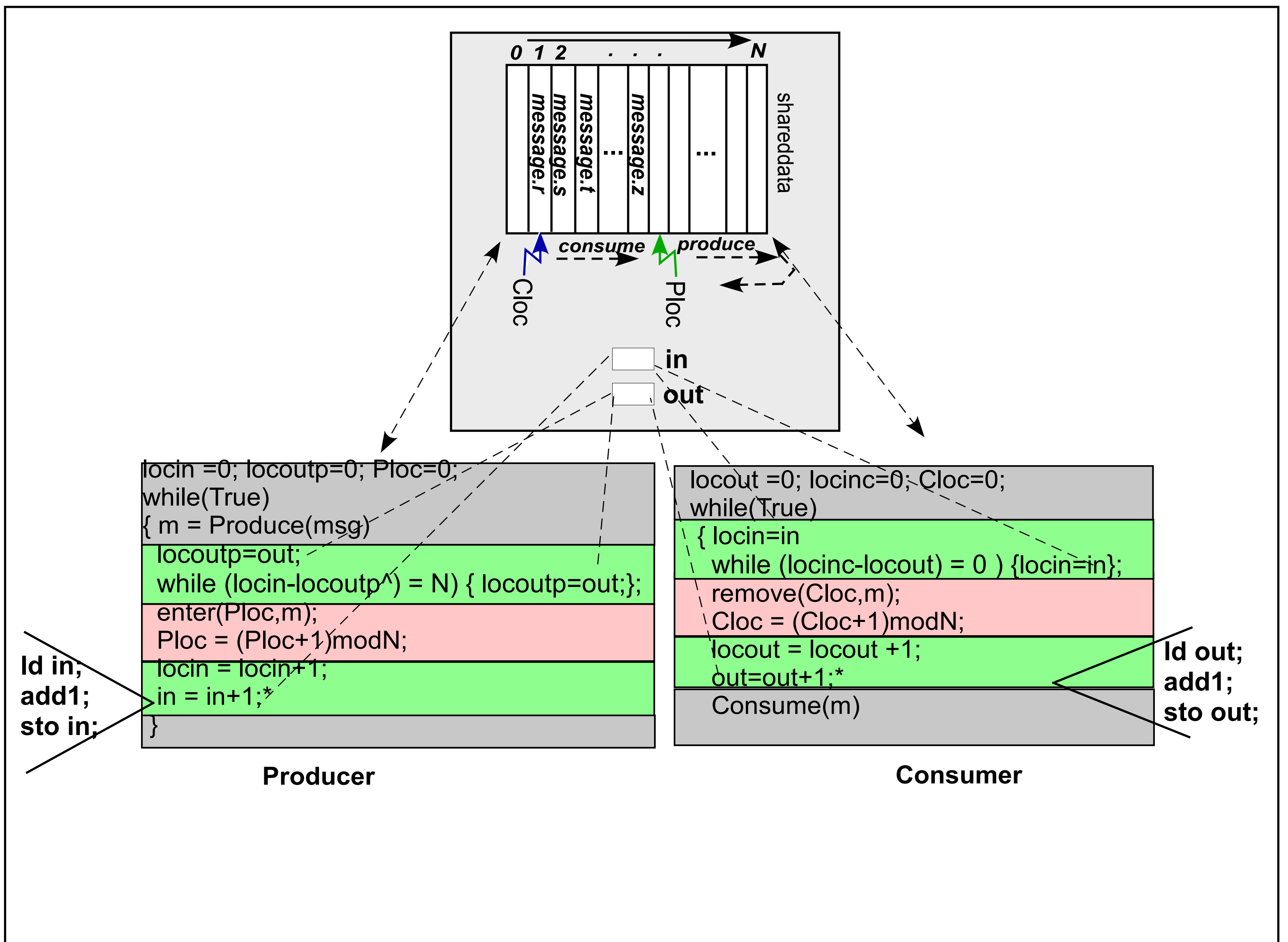
3) Since blocking is not required every time either attempts to enter their CRIT, It might be advisable to either **sleep** or **exit** or **wait, (**even perhaps consider use of **semaphors)** instead of **busy waiting. But better still would be a system call, executed when the loop condition is met say in the producer, putting it in a blocked state until consumer is given control and then producer would be put in the ready state.-and visa versa.**



T*he Producer and Consumer* ***Processes*** *necessarily alternate in controlling the CPU but, unlike strict alternation, each can do a multiple of passes (handle a number of entries in shared memory) on each alternation.On the other hand each* ***Entry*** *, <sharedbit[i],shareddata[i]> in the shared memory experiences the alternation of one Producer pass, with one Consumer pass-which is why we call this approach* ***Multiple Alternators****.*

BW-CRITICAL REGION PROTECTION MULTIPLE ALTERNATORS(CIRCULAR BUFFER)
Two Processesfor Synchronization In Producer-Consumer Applications

© M C Paull 2005



* ^ These increment commands on shared variables are not necessarily atomic. So, for example "in = in +1" could require three machine language commands "ld in; add1;store in "After "ld in ;" control may pass to Consumer. Consumer may then think there is one less produced data available than is actually available. This should not cause any problem-it will simply insure that Consumer consumes at most 1 less than is actually available.

In this implementation, like the previous one, we use local pointers Ploc, and Cloc, to fixed size units of shared memoryand advance Ploc by 1 unit after filling, and Cloc by 1 after emptying a unit.

When, in Producer, we test (locin-locout), locin is correct and locout <= out (because out could be 1 greater than locout since the OS may have removed Consumer from active after it had removed, but not yet recorded a removal. , Therefore (locin - locout) >= (locin-out), which if < N is safe for insertion by Producer. A similar argument shows that Consumers use of (locin - locout) is also safe.

BW- COUNTING INs & OUTs IN MULTIPLE ALTERNATORS FOR SYNCHRONIZATION In Producer-Consumer Applications

© M C Paull 2005

Multiple Alternators

Receive(msg)

N . . . 2 1 0

message.r ... message.z message.y ... message.s

shareddata

1 0 0 0 0 0 1 1 1 sharedbit

produce consume

send

Iloc Jloc

Send(msg)

Send(msg)

0 1 2 . . . N

sender ID

message.r message.s message.t ... message.z ...

shareddata

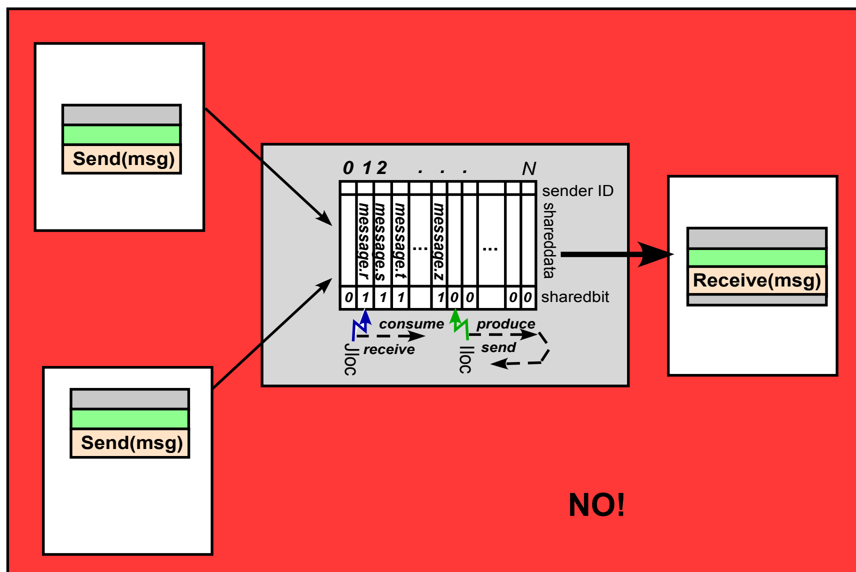
0 1 1 1 1 0 0 0 0 sharedbit

consume produce

receive send

Jloc Iloc

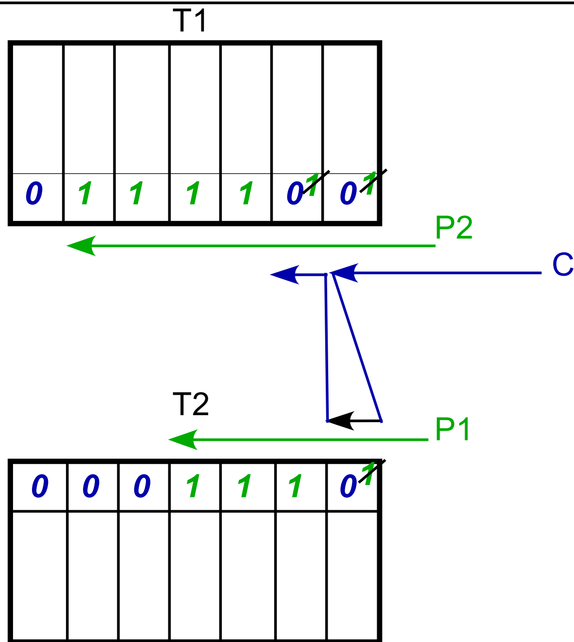
Receive(msg)

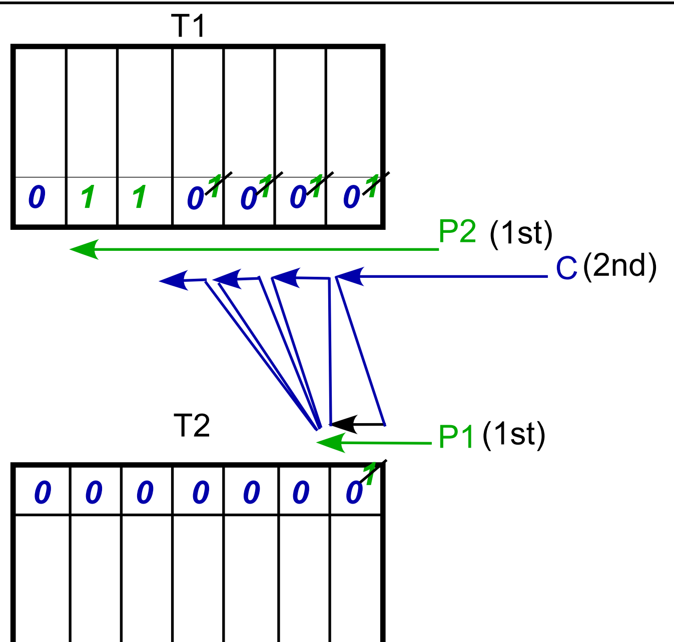


MULTIPLE ALTERNATORS CAN BE A BASIS FOR MESSAGE PASSING

With one M<u>ailbox</u> per Sender-Reciever Pair

© M C Paull 2005





Each Producer makes entries in its own circular Buffer. Consumer alternates in looking for entries between T1 and T2.

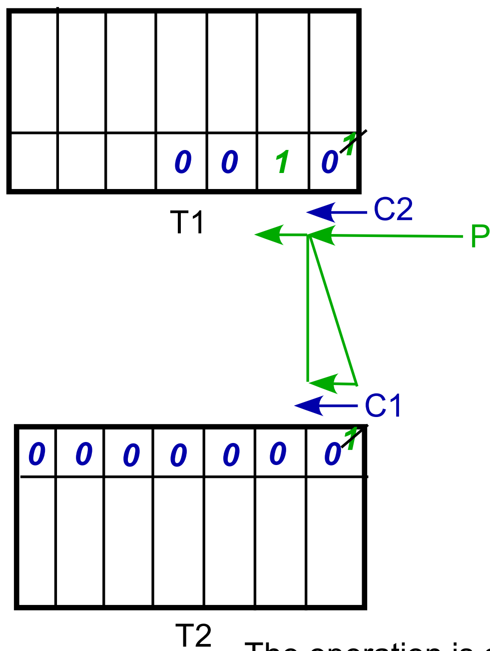
If Consumer arrives at a 0 in T2 at entry e it will then l ook at T1 and if it finds a 1 there will consume it and return to e in T2, etc.

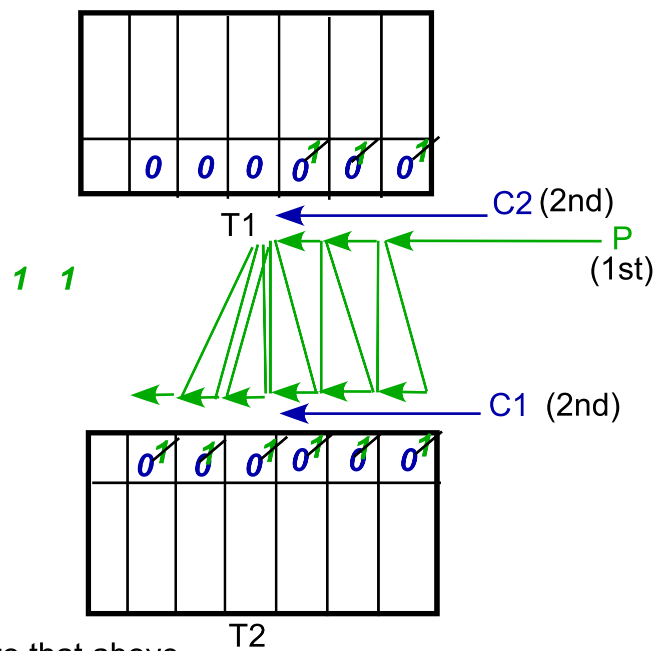
Problem: If a Producer may fill its table, say T1, and still want to produce more. However even though T2 has empty positions, P must block

Two Producers One Consumer

Each Producer Only Has Access To One Table

Consumer alternates between the two tables





The operation is analogous to that above

Two Consumers One Producer

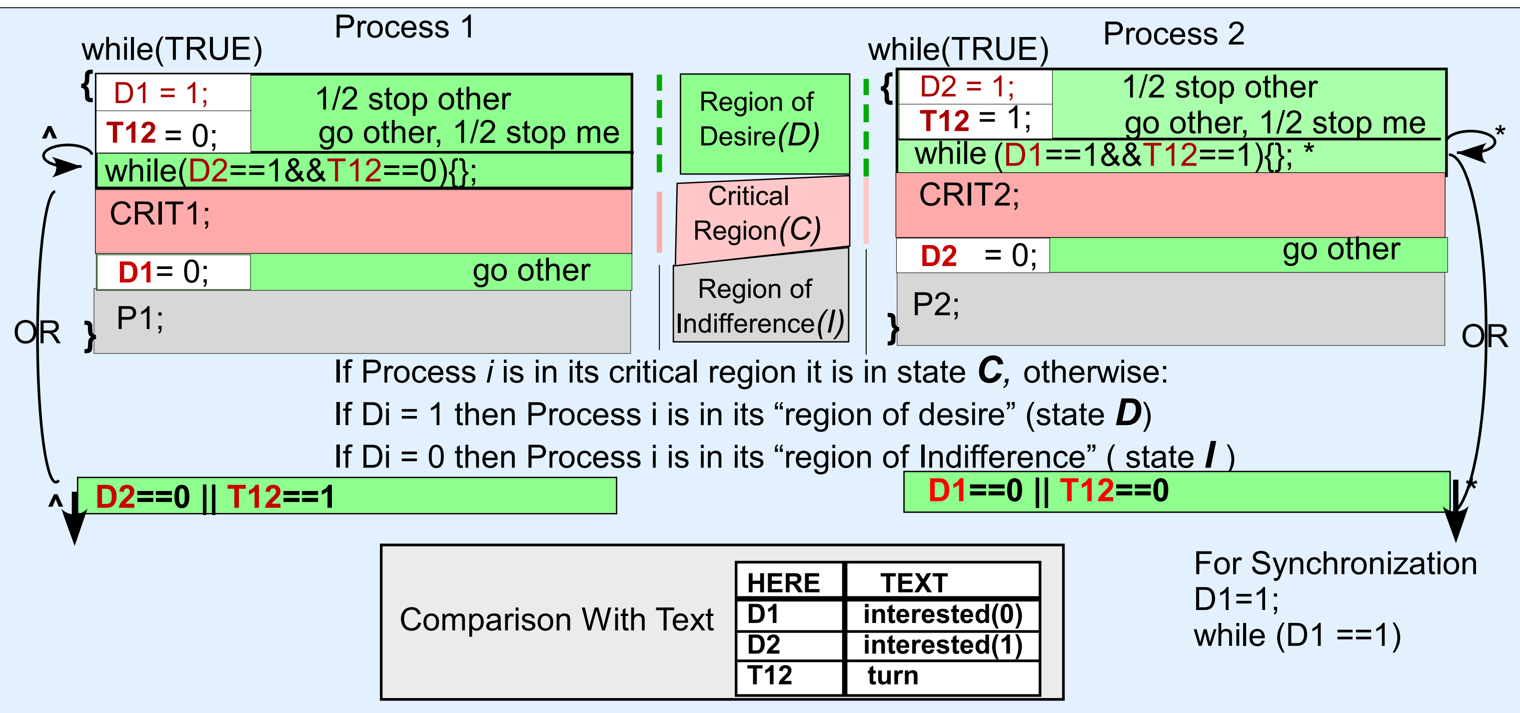
Each Consumer Has Access To Only One Table

Producerr alternates between the two tables

MULTIPLE ALTERNATORS-MULTIPLE USERS(>2) MARKING SHARED MEMORY

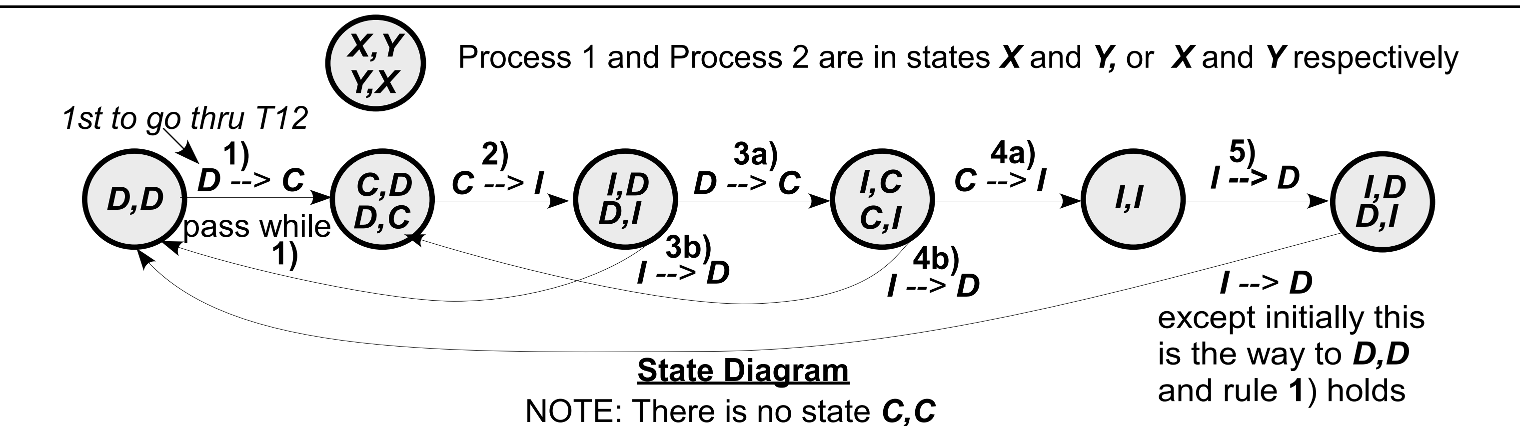
Imperfect

© M C Paull 2005



Comparison With Text

HERE	TEXT
D1	interested(0)
D2	interested(1)
T12	turn



State Diagram

NOTE: There is no state ***C,C***

Only the performance in state ***X,Y*** is spoken of, that in ***Y,X*** is analogous

1) If *both* Processes are in ***D*** the state is ***D,D*** then the first to assign T12 will pass through its **while(..)** into its CRIT, ***C***, but only after the other, the second, has assigned T12. The second will then be blocked from entering its CRIT, ***C***. Therefore, as shown in the state diagram, ***D,D*** becomes ***C,D***

2) From ***C,D*** state goes to ***I,D.*** It cannot go to ***C,C*** because in ***C,D*** T12==1 and D2==1.

3) From ***I,D***, either, (a) state goes to ***I, C*** (through its **while(..)** into its ***C)*** or (b) back to ***D,D*** The first to go through its while will be the one that enters ***C***

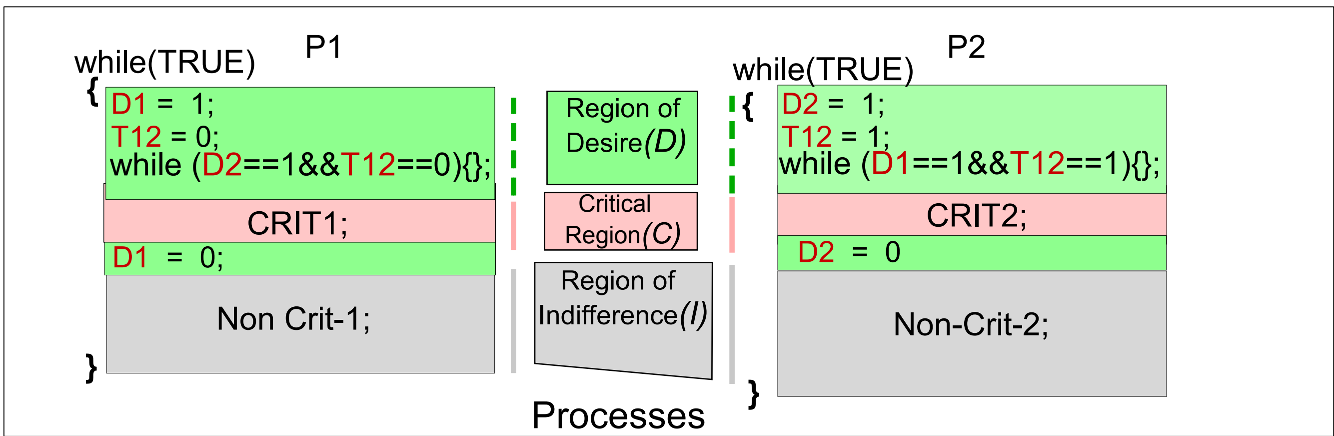
4) From ***I,C***, either (a) ***C*** changes to ***I*** and state is ***I,I*** or (b) ***I*** changes to ***D*** to ***I,I***

5) From ***I,I*** an ***I*** can change to ***D*** and the state become ***I,D***, again but this state is different than that reached by **2).** Here the Process in ***D*** is blocked, there ***D*** was not blocked but entered ***C***

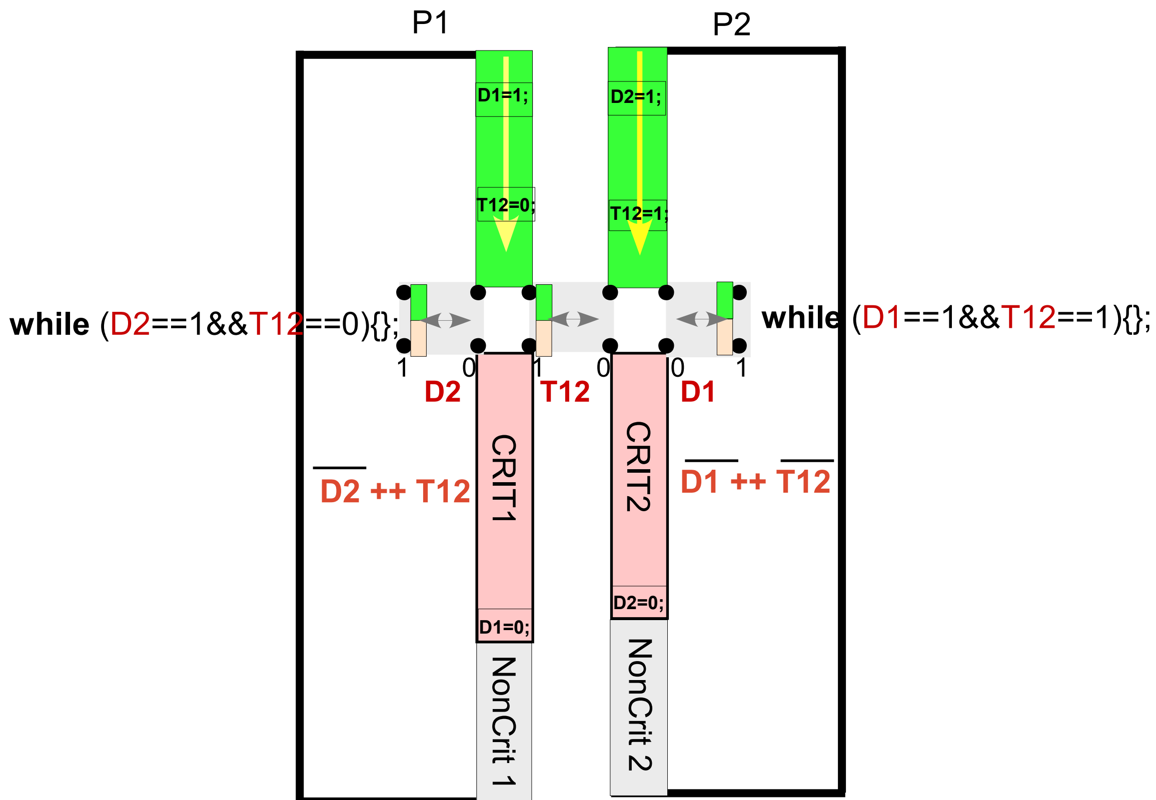
How about synchronization?

BW-GENERAL CRITICAL REGION PROTECTION - PETERSON'S ALGORITHM TWO PROCESSES

© M C Paull 2005



The path through the switches in Process P1 is available if switch D2 is in position 1 and switch T12 is in its 1 position i.e. if D2==0 && T12==1



Assume, as shown, D1 and D2 have both been executed. Both processes are in their Region of Desire. Process 2 was the first to set T12 = 1 and later Process 1 made T12 = 0 then

Process 2 was the first to reach its **while, so now with T12 = 0** Process 2 goes through T12 to CRIT2. Process 1 attempting to go past its while is stopped (Situation is symmetric so If Process 1 was first to reach its **while** then it would stop and when Process 2 reached its **while** then Process 1 goes through T12.

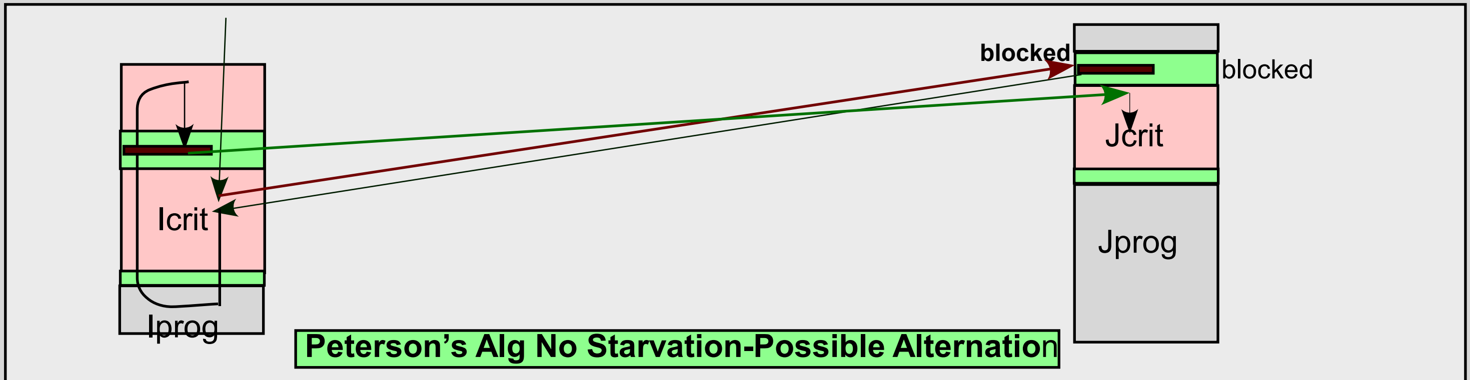
Example Of Switch Positions, Corresponding To **while** conditions, And Their Consequence:

© M C Paull 2005

.P 1 completes CRIT1 goes through D1=0.

So, if P2 is activated it can continue to D1==0 into CRIT2. Now P1.....

...continues back to its while it blocks since T12=0. So when P2 is activated it can continue to T12==0 to CRIT2.



PETERSON'S ALGORITHM TWO PROCESS: SWITCHING MODEL TRACE

© M C Paull 2005

<= 1 path active at bottom of block

Process 1

```
D1 = 1;
T12 = 0;
while(D2==1&&T12==0);
T12.3 = 0;
while(D3==1&&T12.3==0);
CRIT1;
D1=0;
Non-CRIT1
```

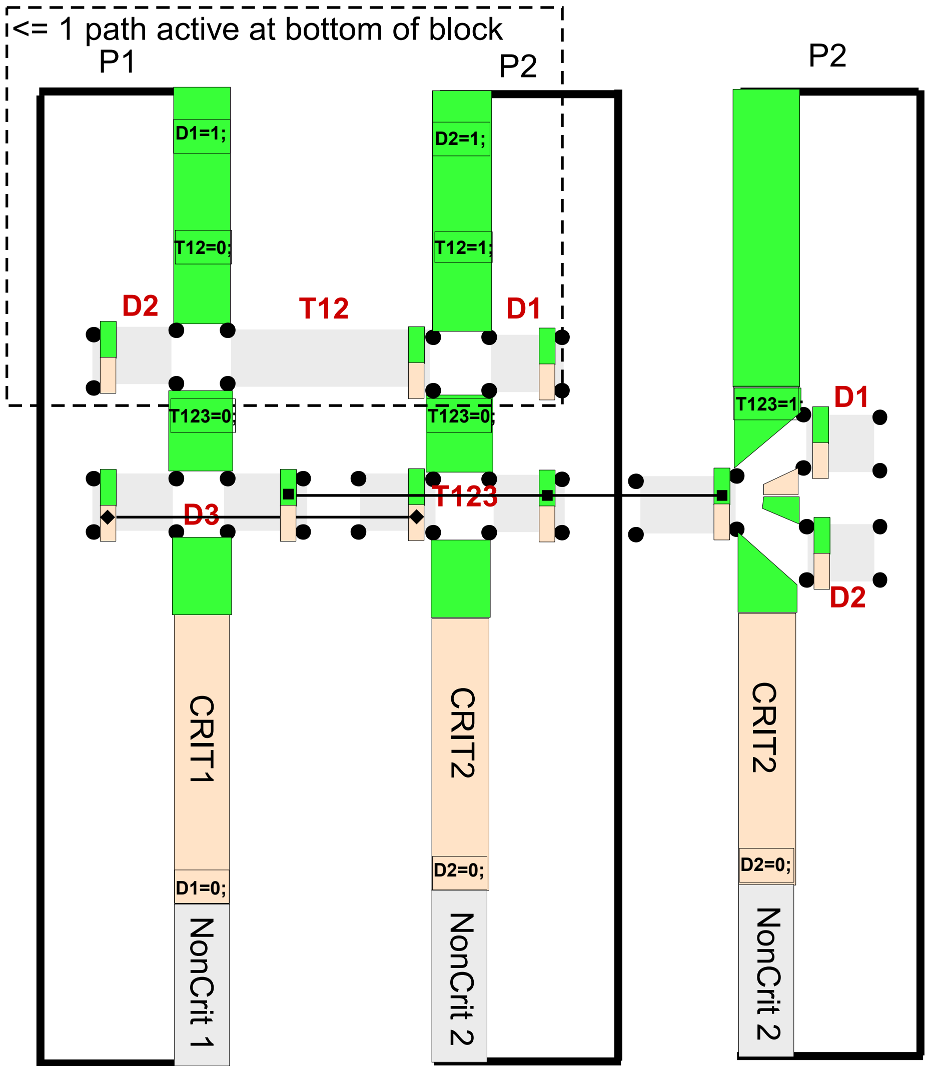
Process 2

```
D2 = 1;
T12 = 1;
while(D1==1&&T12==1);
T12.3 = 0;
while(D3==1&&T12.3==0);
CRIT1;
D2=0;
Non-CRIT1
```

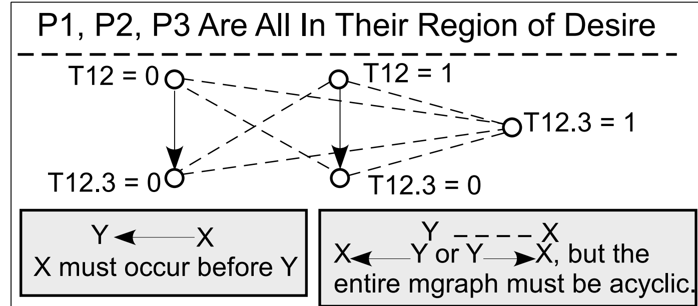
Process 3

```
D3 - 1;

T12.3 = 0;
while((D1==1++D2==1)&&T12.3==1);
CRIT3;
D3=0;
Non-CRIT3
```



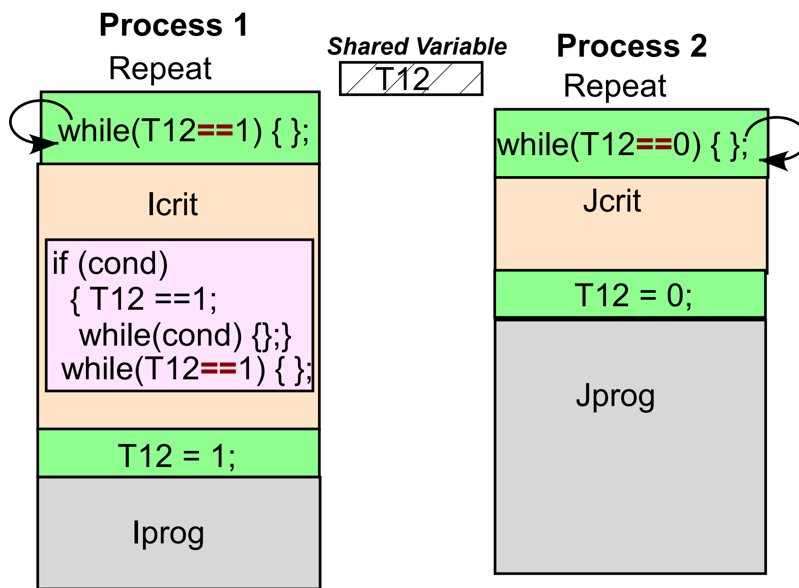
P1, P3 Are in Their Region of Desire, P2 is in Non-CRIT2.	P1, P2 Are In Their Region of Desire. P3 is in Non-CRIT3
D2 = 0, so closing one of the upper pairs of paths in P1. always leaves a path to the lower pairs of paths in P3, It is as though P2 did not exist.	D3 = 0 so closing one of the lower pairs of paths in P1 and P2. This leaves P1 and P2 in the same state as they would be if P3 did not exist.

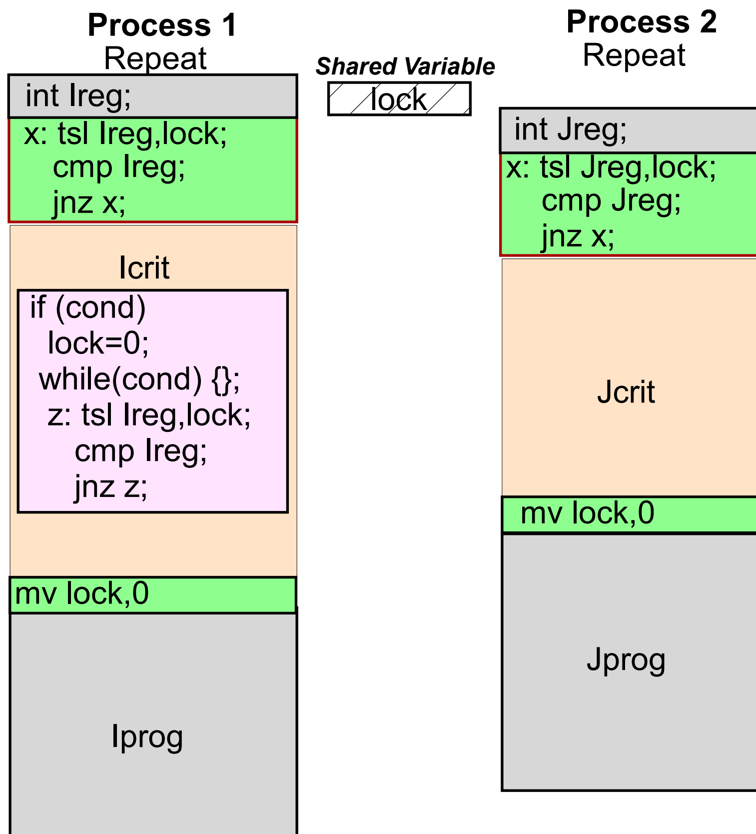


Dj	The jth process is in its "region of desire"
T12	Process 2 has reached its first "while" after process 1 has reached its
$\overline{T12}$	Process 1 has reached its first "while" after process 2 has reached its
T12.3	Process 3 has reached its the second "while" after Processes 1 or 2 has reached its.
$\overline{T12.3}$	Process 1 or 2 has reached its the second "while" after Process 3 has reached its.

PETERSON'S ALGORITHMTHREE PROCESS: SWITCHING MODEL

© M C Paull 2005





Synchronization Problem With Busy Waiting Solutions:
Process, P, running in its Critical Region detects “cond” under which it needs to block in its Critical Region until “cond” is no longer true.“cond” will be made false only after a related critical region in anoher Process runs. When “cond “ is no longer true P is ready continue and must do so eventually.

Busy Waiting AND SYNCHRONIZATION

© M C Paull 2005