

Background

Process Synchronization is a technique which is used to coordinate the process that use shared Data.

Independent Process –

The process that does not affect or is affected by the other process while its execution then the process is called Independent Process. Example The process that does not share any shared variable, database, files, etc.

Cooperating Process –

The process that affect or is affected by the other process while execution, is called a Cooperating Process. Example The process that share file, variable, database, etc are the Cooperating Process

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills the buffer. We can do so by having an integer **count** that keeps track of the number of elements in the buffer. Initially, count is set to 0. It is incremented by the producer after it produces a new item into the buffer and is decremented by the consumer after it consumes the item in the buffer.



Producer

```
while (true) {
    /* produce an item and put in nextProduced */
    while (counter == BUFFER_SIZE)
        ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```



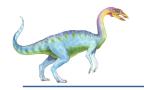


Consumer

```
while (true) {
    while (counter == 0)
    ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;

    /* consume the item in nextConsumed
}
```





Race Condition

counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
count = register2
```

□ Consider this execution interleaving with "count = 5" initially:

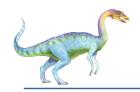
```
S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {count = 6}
S5: consumer execute counter = register2 {count = 4}
```





- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.
- ensure that only one process at a time can be manipulating the variable counter
- the problem with critical sections is called a race condition, where the outcome of the program depends on the order in which concurrent processes or threads execute
- □ Race conditions can result in a wide range of issues, including data corruption, deadlocks, and resource starvation.





Critical Section Problem

- □ Consider system of n processes $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has critical section segment of code
 - Process may be changing common variables, updating table, writing file, etc
- When one process is executing in its critical section, no other process is to be allowed to execute in critical section. That is, no two processes are executing in their critical sections at the same time. Critical section problem is to design protocol to solve this.
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels





Critical Section

☐ General structure of process p_i is

```
entry section

critical section

exit section

remainder section

while (TRUE);
```

Figure 6.1 General structure of a typical process P.





Solution to Critical-Section Problem

- 1. **Mutual Exclusion** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- 2. **Progress** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.(decision must be taken within a finite time)
- 3. **Bounded Waiting** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.





First Version of Dekker's Algorithm

- Succeeds in enforcing the mutual exclusion.
- The processor's must enter and exit their critical sections in strict alteration (so inefficient).
- Enforces lockstep synchronization problem.

It means each process depends processnumber:=2; on other to complete its execution. If one of the two processes completes its execution, then the second process runs.

processnumber:=2; otherstuffone end end;

program versionone; var processnumber: integer;

procedure processone;
begin
while true do
begin
while processnumber=2
do;
critical_section_one;
processnumber:=2;
otherstuffone
end
end;

procedure procestwo;
begin
while true do
begin
while processnumber=1
do;
critical_section_two;
processnumber:=1;
otherstufftwo
end
end;

begin
processnumber:=1;
parbegin
processone;
processtwo;
parend
end;

Same process can't enter CS continuously





Second Version of Dekker's Alg

program versiontwo;

var p1inside, p2inside : boolean;

procedure processone;
begin
while true do
begin
while p2inside do;
plinside:=true;
critical_section_one;
plinside:=false;
otherstuffone
end
end;

procedure procestwo;
begin
while true do
begin
while plinside do;
p2inside:=true;
critical_section_two;
p2inside:=false;
otherstufftwo
end
end;

begin
plinside:=false;
p2inside:=false;
parbegin
processone;
processtwo;
parend
end;

Lockstep **synchronization** removed(but it creates problem; both simultaneously enters into the CS). Lockstep synchronization removed by using two flags to indicate its current status and updates them accordingly at the entry and exit section.

statements are generally atomic, but series of statements are not

Mutual exclusion is not guaranteed



```
program versionthree;
var plwantstoenter, p2wantstoenter:boolean;
procedure processone;
    begin
         while true do
         begin
             plwantstoenter := true;
             while p2wantstoenter do;
             criticalsectionone;
             plwantstoenter := false;
             otherstuffone
         end;
    end;
procedure processtwo
     begin
         while true do
          begin
              p2wantstoenter := true;
              while plwantstoenter do;
              criticalsectiontwo;
              p2wantstoenter := false;
              otherstufftwo
          end:
     end;
begin
     plwantstoenter := false;
     p2wantstoenter := false;
     parbegin
          processone;
          processtwo;
     parend
end.
```

Mutual exclusion is guaranteed.

Introduces two-process deadlock.

Both threads could get flag simultaneously and they will wait for infinite time.

> Both threads should not set the flag simultaneously, introduce randomness in the next version(4)

> > Version 3



```
program versionfour;
var plwantstoenter, p2wantstoenter:boolean;
procedure processone;
     begin
          while true do
          begin
              plwantstoenter := true;
              while p2wantstoenter do
                   begin
                        plwantstoenter := false;
                        delay(random, fewcycles);
                        plwantstoenter := true;
                   end;
               criticalsectionone;
               plwantstoenter := false;
               otherstuffone
          end;
     end;
procedure processtwo
    begin
        while true do
        begin
             p2wantstoenter := true;
             while plwantstoenter do
                 begin
                      p2wantstoenter := false;
                      delay(random, fewcycles);
                      p2wantstoenter := true;
                 end;
             criticalsectiontwo;
             p2wantstoenter := false;
             otherstufftwo
        end;
    end;
```

Mutual exclusion is guaranteed.

Deadlock cannot occur.

Indefinite postponement could occur.(random delay)

Version 4 is unacceptable.

Version 4



Dekker's algorithm

```
program dekkersalgorithm;
                                                   procedure processtwo;
var favoredprocess: (first, second);
                                                   begin
    plwantstoenter, p2wantstoenter: boolean;
                                                       while true do
procedure processone;
                                                       begin
begin
                                                            p2wantstoenter := true;
    while true do
                                                            while plwantstoenter do
    begin
        plwantstoenter := true;
                                                                 if favoredprocess = first then
        while p2wantstoenter do
                                                                 begin
        begin
                                                                     p2wantstoenter := false;
             if favoredprocess = second then
                                                                     while favoredprocess = first do;
             begin
                                                                     p2wantstoenter := true
                  plwantstoenter := false;
                                                                 end;
                  while favoredprocess = second do;
                                                            end;
                  plwantstoenter := true
                                                            criticalsectiontwo;
             end;
                                                            favoredprocess := first;
        end;
                                                            p2wantstoenter := false;
        criticalsectionone:
                                                            otherstufftwo
        favoredprocess := second;
                                                       end;
        plwantstoenter := false;
                                                   end;
        otherstuffone
                                                   begin
    end;
                                                        plwantstoenter := false;
end;
                                                        plwantstoenter := false;
                                                        favoredprocess := first;
                                                        parbegin
                                                              processone;
                                                              processtwo;
                                                        parend
```

end.

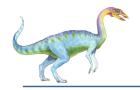
Mutual exclusion guaranteed.

Lockstep Synchroniz ation not enforced.

Deadlock is avoided.

Resolves possibility of indefinite postponeme nt.





Dekker's algorithm of ME Primitive

Global storage

 Mutual exclusion quaranteed.

Lockstep
 Synchronizat
 ion not
 enforced.

- Deadlock is avoided.
- Resolves
 possibility
 of indefinite
 postponeme
 nt.

```
Procedure processOne
begin
   while TRUE do begin
      plWantsToEnter := TRUE;
      while (p2WantsToEnter) do begin
         if (favoredProcess = 2) then begin
            plWantsToEnter := FALSE;
            while (favoredProcess = 2) do;
            plWantsToEnter := TRUE;
         end;
      end:
      // do critical section of 1.
      favoredProcess := 2;
      plWantsToEnter := FALSE;
      // do remainder sections of 1.
   end:
end;
```

plWantsToEnter

p2WantsToEnter

favoredProcess

```
Procedure processTwo
begin
  while TRUE do begin
      p2WantsToEnter := TRUE;
     while (plWantsToEnter) do begin
         if (favoredProcess = 1) then begin
            p2WantsToEnter := FALSE;
            while (favoredProcess = 1) do;
            p2WantsToEnter := TRUE;
         end;
      end;
      // do critical section of 2.
      favoredProcess := 1;
     p2WantsToEnter := FALSE;
      // do remainder sections of 2.
  end:
end;
```

plWantsToEnter

p2WantsToEnter

favoredProcess