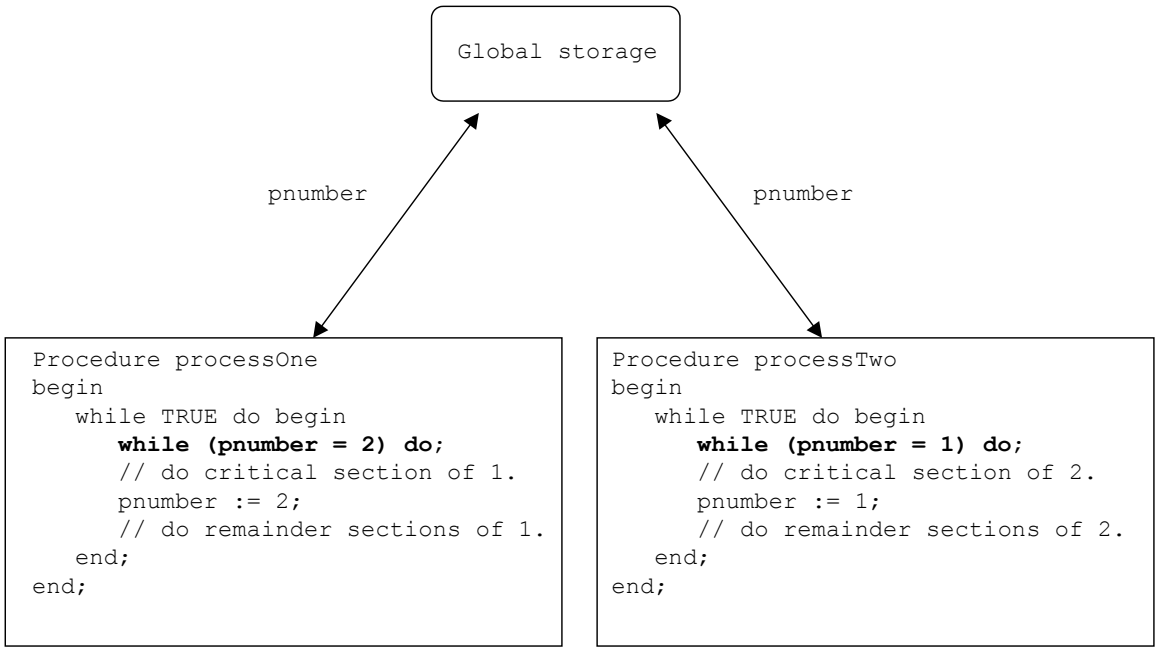
EXAMPLE – A first effort (lockstep synchronization).



Pnumber = 1 initially

- Characteristics:
  1. Mutual exclusion is guaranteed.
  2. Deadlock is avoided.
  3. Can only manage 2 processes.
  4. The processes must enter and exit their critical sections in strict alternation (inefficient).
  5. Bounded waiting is not guaranteed. The termination of one causes the remaining to be indefinitely delayed. And, if one process no longer requires execution of its critical section, the other process is indefinitely delayed.

EXAMPLE – lockstep synchronization implemented in Windows/DOS.

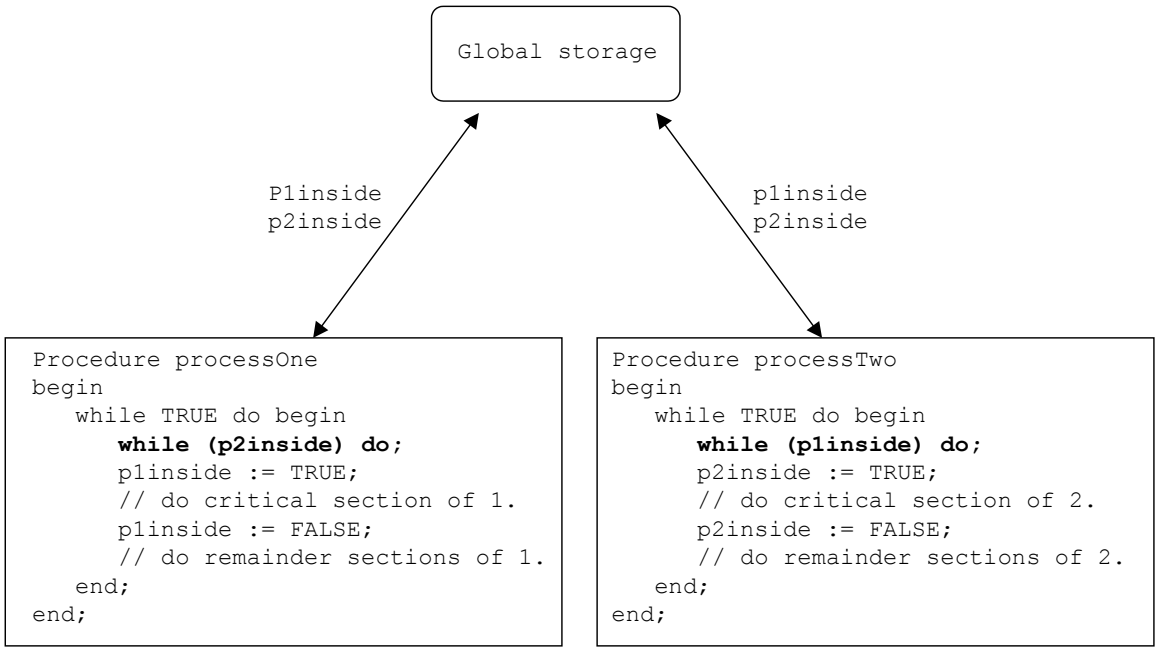
```
// Master - Run first.
#include <dos.h>
void main(void){
   poke(0x0000, 0x0412, 0);
}

// Process 0.
#include <iostream.h>
#include <dos.h>
void main(void){
int counter = 0;
   while (counter < 1000) {
      while (peek(0x0000, 0x0412) == 0);
      cout << "Critical section in 0.\n";
      delay(1000);
      poke(0x0000, 0x0412, 0);
      cout << "Other stuff in 0.\n";
      counter++;
   }
}

// Process 1.
#include <iostream.h>
#include <dos.h>
void main(void){
int counter = 0;
   while (counter < 1000) {
      while (peek(0x0000, 0x0412) == 1);
      cout << "Critical section in 1.\n";
      delay(1000);
      poke(0x0000, 0x0412, 1);
      cout << "Other stuff in 1.\n";
      counter++;
   }
}
```

*** NOTE. Address 412 is only used by the IBM PCjr. Therefore, address 412 can be used to pass info (bytes) between (otherwise independent) DOS programs running under Windows.**

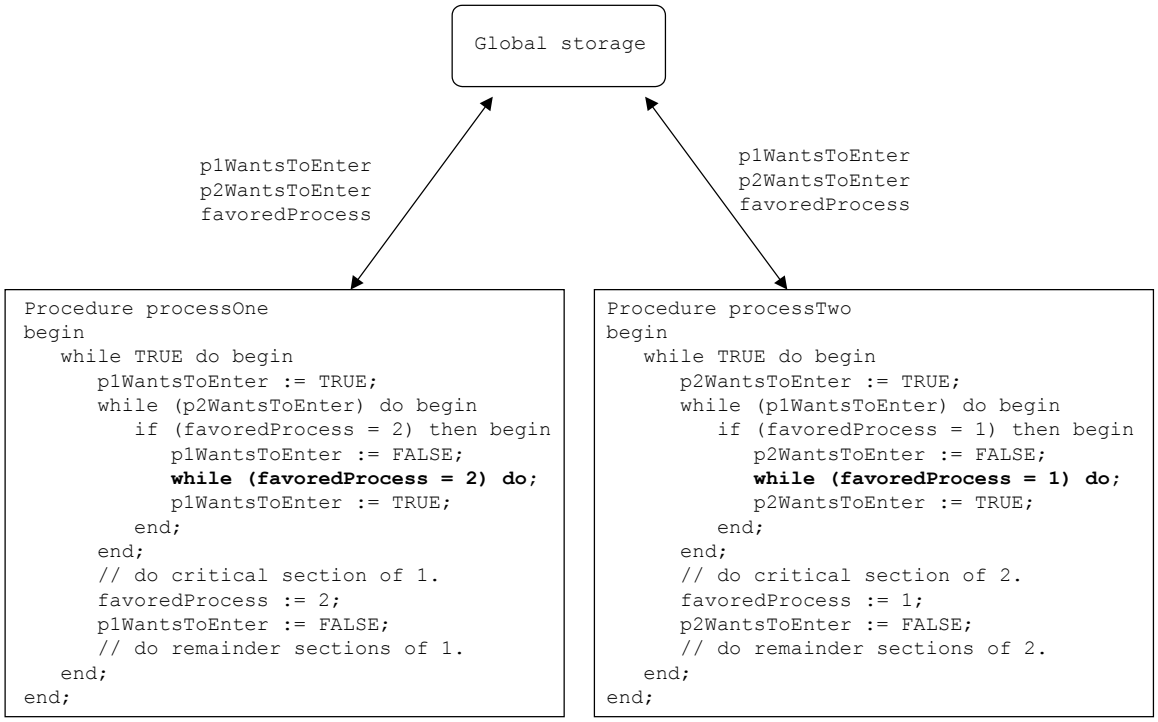
EXAMPLE – A second effort.




```
P1inside = false initially
p2inside = false initially
```

- Characteristics:
  1. Can only manage 2 processes.
  2. Mutual exclusion is not guaranteed. Due to concurrent processing, between the time a process determines (in the while loop) that it can go ahead (into its critical section) and the time that the process sets a flag to indicate it is in its critical section, there is enough time (statements are generally atomic, but series of statements are not) for the other process to test its flag and slip into its critical section.
  3. Bounded waiting is not guaranteed. The termination of one causes the remaining to be indefinitely delayed.

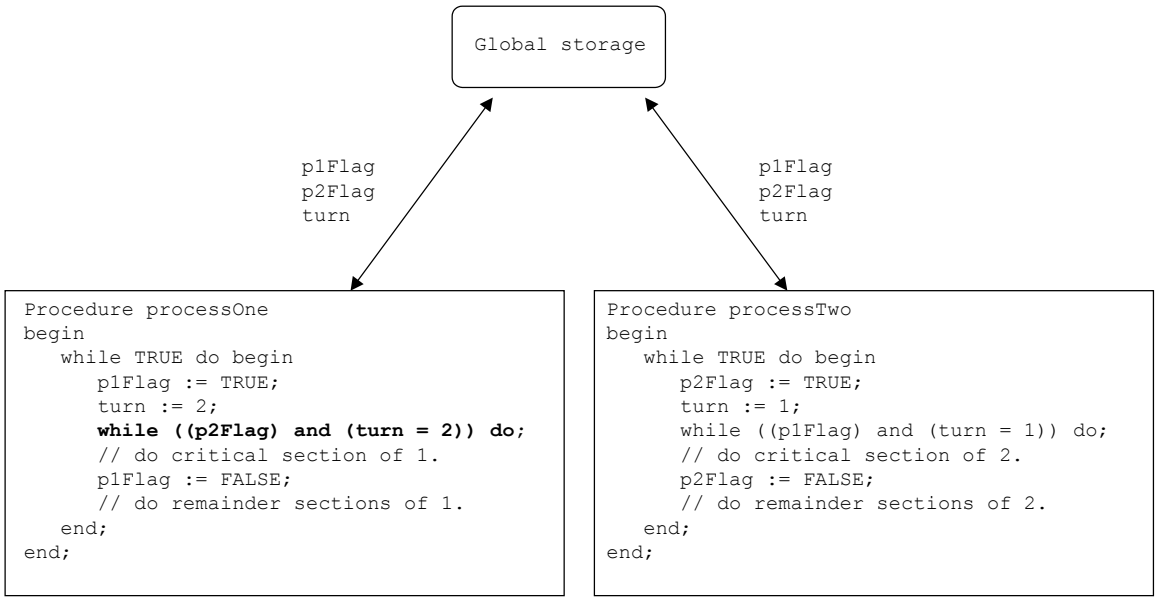
DEKKER'S ALGORITHM:




```
p1WantsToEnter := FALSE initially.
p2WantsToEnter := FALSE initially.
favoredProcess := 1 initially.
```

- Characteristics:
  1. Mutual exclusion is guaranteed.
  2. Deadlock is avoided.
  3. Indefinite postponement avoided.
  4. A software-only solution.
  5. Can only manage 2 processes.
  6. More efficient as no ordering is forced.

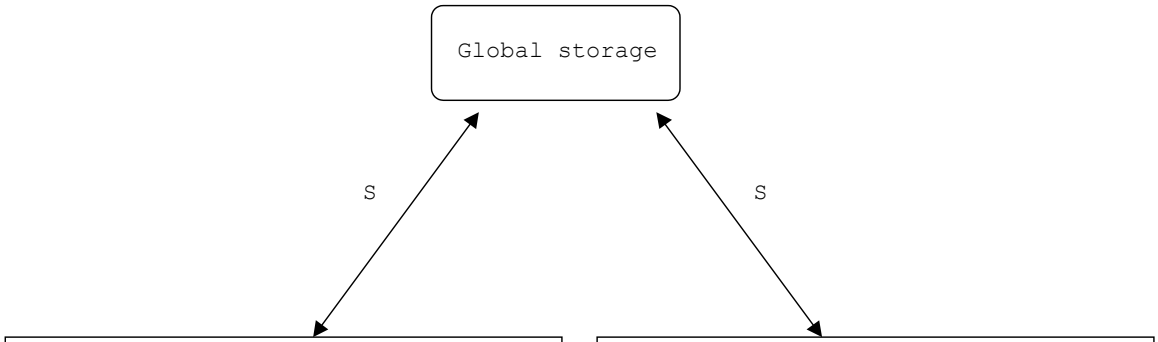
PETERSON'S ALGORITHM:




```
p1Flag := FALSE initially.
p2Flag := FALSE initially.
turn := 2 initially.
```

- Characteristics:
  1. Mutual exclusion is guaranteed.
  2. Deadlock is avoided.
  3. Indefinite postponement avoided.
  4. A software-only solution.
  5. Can only manage 2 processes.

EXAMPLE – lockstep synchronization modified with binary semaphores.




```
Procedure processOne
begin
   while TRUE do begin
      while (S <= 0) do;
      S--;
      // do critical section of 1.
      S++;
      // do remainder sections of 1.
   end;
end;
```

```
Procedure processTwo
begin
   while TRUE do begin
      while (S <= 0) do;
      S--;
      // do critical section of 2.
      S++;
      // do remainder sections of 2.
   end;
end;
```

```
S = 1 initially
```

- Characteristics:
  1. Mutual exclusion is guaranteed.
  2. Progress guaranteed.
  3. Indefinite postponement avoided. Or is it? What assumption is required? Hint, consider 3 or more processes.
  4. Can manage n (n > 2) processes.

EXAMPLE – lockstep synchronization modified with binary semaphores implemented in Windows/DOS.

```
// Master - Run first.
#include <dos.h>
void main(void){
   poke(0x0000, 0x0412, 1);
}

// Process I.
#include <iostream.h>
#include <dos.h>
void main(void){
int counter = 0;
while (counter < 1000) {
      while (peek(0x0000, 0x0412) <= 0);
      poke(0x0000, 0x0412, peek(0x0000, 0x0412)-1);
      cout << "Critical section in I.\n";
      delay(1000);
      poke(0x0000, 0x0412, peek(0x0000, 0x0412)+1);
      cout << "Other stuff in I.\n";
      counter++;
   }
}
```

1. Why isn't the above a "true" semaphore example?

EXAMPLE – A message passing – handshaking - PRODUCER-CONSUMER method implemented in Windows/DOS.

```
// Producer – start first.
#include <iostream.h>
#include <dos.h>
void main(void){
int counter = 0;
poke(0x0000, 0x0412, 0);
   while (counter < 256) {
      while (peek(0x0000, 0x0412) != 0);
      cout << "Produce a value.\n";
      delay(500);
      poke(0x0000, 0x0412, (char)counter);
      counter++;
   }
}

// Consumer.
#include <iostream.h>
#include <dos.h>
void main(void){
int counter = 0;
   while (counter < 256) {
      while (peek(0x0000, 0x0412) == 0);
      cout << "Consume the value.\n";
      cout << "Value: " << peek(0x0000, 0x0412) << endl;
      delay(1000);
      poke(0x0000, 0x0412, 0);
      counter++;
   }
}
```

- Characteristics:
  1. The semaphore and data-item are the same.
  2. If either the producer or the consumer terminate, the other is left indefinitely postponed (violates bounded waiting).
  3. Can only manage 2 processes (is this really a problem?).
  4. The processes must enter and exit their critical sections (produce / consume) in strict alternation. Would a better

approach be to apply Dekker's or, maybe, Peterson's algorithm?

EXAMPLE – A fixed capacity storage facility PRODUCER-CONSUMER method (handshaking) implemented in Windows/DOS.

```
#include <iostream.h>
#include <conio.h>
#include <dos.h>
// Producer.
void main(void){
int counter = 0;
   poke(0x0000, 0x0412, 0);
   while (counter < 1) {
      while (peek(0x0000, 0x0412) == 255) {
         cout << "Buffer full - delayed...\n";
      }
      cout << "Produce another value.\n";
      poke(0x0000, 0x0412, (peek(0x0000, 0x0412)+1));
   }
}
```

```
#include <iostream.h>
#include <conio.h>
#include <dos.h>
// Consumer.
void main(void){
int counter = 0;
   while (counter < 1) {
      while (peek(0x0000, 0x0412) == 0) {
         cout << "Waiting for product.\n";
      }
      cout << "Consume another value.\n";
      poke(0x0000, 0x0412, (peek(0x0000, 0x0412)-1));
   }
}
```

- Characteristics:
  1. The semaphore (counting semaphore) indicates only the number of items in the buffer to be consumed.
  2. Bounded waiting not OK.
  3. Can manage N processes.
  4. The processes do not have to enter and exit their critical sections (produce / consume) in strict alternation.

A semaphore example for UNIX.

```
/*************************************************************/
/* semabinit.c                                               */
/* - initialize a semaphore for use by programs sema and semb */
/*************************************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>

/*
   The semaphore key is an arbitrary long integer which serves as an
   external identifier by which the semaphore is known to any program
   that wishes to use it.
*/

#define KEY (1492)

void main()
{
   int id; /* Number by which the semaphore is known within a program */

   /*
   The next thing is an argument to the semctl() function. Semctl()
   does various things to the semaphore depending on which arguments
   are passed. We will use it to make sure that the value of the
   semaphore is initially 0.
   */

   union semun {
        int val;
        struct semid_ds *buf;
        ushort * array;
   } argument;

   argument.val = 0;

   /*
   Create the semaphore with external key KEY if it doesn't already
   exists. Give permissions to the world.
   */

   id = semget(KEY, 1, 0666 | IPC_CREAT);

   /* Always check system returns. */

   if (id < 0) {
      fprintf(stderr, "Unable to obtain semaphore.\n");
      exit(0);
   }

   /*
   What we actually get is an array of semaphores. The second
```

```
   argument to semget() was the array dimension - in our case
   1.
   */

   /*
   Set the value of the number 0 semaphore in semaphore array
   # id to the value 0.
   */

   if (semctl(id, 0, SETVAL, argument) < 0) {
      fprintf( stderr, "Cannot set semaphore value.\n");
   } else {
      fprintf(stderr, "Semaphore %d initialized.\n", KEY);
   }
}


/*************************************************************/
/* Semaphore example program A (sema.c)                      */
/* - We have two programs, sema and semb. Semb may be        */
/*   initiated at any time, but will be forced to wait until */
/*   sema is executed. Sema and semb do not have to be       */
/*   executed by the same user!                              */
/*************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY (1492)
/* This is the external name by which the semaphore is known to any
   program that wishes to access it. */

void main()
{
   int id;  /* Internal identifier of the semaphore. */
   struct sembuf operations[1];

   /* An "array" of one operation to perform on the semaphore. */

   int retval; /* Return value from semop() */

   /* Get the index for the semaphore with external name KEY. */
   id = semget(KEY, 1, 0666);
   if (id < 0) {
      /* Semaphore does not exist. */
      fprintf(stderr, "Program sema cannot find semaphore, exiting.\n");
      exit(0);
   }

   /* Do a semaphore V-operation. */
   printf("Program sema about to do a V-operation. \n");

   /* Set up the sembuf structure. */
   /* Which semaphore in the semaphore array : */
   operations[0].sem_num = 0;
```

```
   /* Which operation? Add 1 to semaphore value : */
   operations[0].sem_op = 1;

   /* Set the flag so we will wait : */
   operations[0].sem_flg = 0;

   /* So do the operation! */
   retval = semop(id, operations, 1);
   if (retval == 0) {
      printf("Successful V-operation by program sema.\n");
   } else {
      printf("sema: V-operation did not succeed.\n");
      perror("REASON");
   }
}

/*
Think carefully about what the V-operation does. If sema is executed
twice, then semb can execute twice.
*/


/*************************************************************/
/* Semaphore example program B (semb.c)                      */
/*  - We have two programs, sema and semb. Semb may be       */
/*    initiated at any time, but will be forced to wait until */
/*    sema is executed. Sema and semb do not have to be      */
/*    executed by the same user!                             */

/* HOW TO TEST:                                              */
/*  Execute semb &                                           */
/*  The & is important - otherwise you would have have to    */
/*  move to a different terminal to execute sema.            */

/*  Then execute sema.                                       */
/*************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY (1492)
/* This is the external name by which the semaphore is known to any
   program that wishes to access it. */

void main()
{
   int id;  /* Internal identifier of the semaphore. */
   struct sembuf operations[1];
   /* An "array" of one operation to perform on the semaphore. */

   int retval; /* Return value from semop() */

   /* Get the index for the semaphore with external name KEY. */
   id = semget(KEY, 1, 0666);
   if (id < 0) {
      /* Semaphore does not exist. */
```

```
      fprintf(stderr, "Program semb cannot find semaphore, exiting.\n");
      exit(0);
   }

   /* Do a semaphore P-operation. */
   printf("Program semb about to do a P-operation. \n");
   printf("Process id is %d\n", getpid());

   /* Set up the sembuf structure. */
   /* Which semaphore in the semaphore array : */
   operations[0].sem_num = 0;

   /* Which operation? Subtract 1 from semaphore value : */
   operations[0].sem_op = -1;

   /* Set the flag so we will wait : */
   operations[0].sem_flg = 0;

   /* So do the operation! */
   retval = semop(id, operations, 1);

   if (retval == 0) {
      printf("Successful P-operation by program semb.\n");
      printf("Process id is %d\n", getpid());
   } else {
      printf("semb: P-operation did not succeed.\n");
   }
}

/*
Think carefully about what the V-operation does. If sema is executed
twice, then semb can execute twice.
*/
```