Algoritmo di Dekker

```
shared int turn = P;
shared boolean needp = false;
shared boolean needq = false;
cobegin P // Q coend
```

```
process P {
  while (true) {
    needp = true;
    while (needq)
      if (turn == Q) {
        needp = false;
        while (turn == Q);
        needp = true;
      }
    critical section
    needp = false;
    turn = Q;
    non-critical section
  }
}
```

```
process Q {
     while (true) {
     needq = true;
     while (needp)
       if (turn == P) {
          needq = false;
          while (turn == P);
          needq = true;
       }
     critical section
     needq = false;
     turn = P;
     non-critical section
     }
}
```

Algoritmo di Peterson

```
shared boolean needp = false;
shared boolean needq = false;
shared int turn;
cobegin P // Q coend
```

```
process P {
  while (true) {
    needp = true;
    turn = Q;
    while (needq && turn != P);
    critical section
    needp = false;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    needq = true;
    turn = P;
    while(needp && turn != Q);
    critical section
    needq = false;
          non-critical section
  }
}
```

TEST & SET

Esempio

$F(X,Y)= <X_1=X;\ X=Y\&\&X;\ Y=X_1||Y>$

$X_2=X_a$ && Y_a (perchè posso scambiare le posizioni di Y_a && X_a)

$Y_2=X_a$ || Y_a

	Libera	Occupata
1) X = G (Globale)	G	
$G_z = G_a$		
$L_z = G_a$ || L_a	L	

L_a = c = le assegno un valore TRUE o FALSE

	Libera	Occupata
X = G (Globale)	G	
$G_z = G_a$		
$L_z = G_a$ || FALSE	L	FALSE

L_a = c = FALSE

Se questo tentativo non dovesse funzionare provare ponendo la X come Locale.

Una volta confermata la validità della scelta, sostituire opportunamente i valori nel seguente codice:

```
G = libera;
.
.
do{
  L = c;
  F(G,L);
}
while (L Є A);
.
.
G = libera;
.
.
```

Semafori Normali

```
class Semaphore{
     private int valore;

     Semaphore (v){
          valore = v;
     }

     void P() {
       [enter CS]
       value--;
       if (value < 0) {
          int pid = <id del processo che ha invocato P>;
          queue.add(pid);
          suspend(pid);
       }
       [exit CS]
     }

     void V() {
      [enter CS]
       value++;
       if (value <= 0){
          int pid = queue.remove();
          wakeup(pid);
       }
      [exit CS]
     }
}
```

Semafori Binari

```
class BinarySemaphore {
  private int value;
  Queue queue0 = new Queue();
  Queue queue1 = new Queue();

  BinarySemaphore() {
     value = 1;
  }

  void P() {
    [enter CS]
    int pid = <process id>;
    if (value == 0) {
      queue0.add(pid);
      suspend(pid);
    }
    value--;
    if (queue1.size() > 0) {
      int pid = queue1.remove();
      wakeup(pid);
    }
    [exit CS]
  }

  void V() {
     [enter CS]
     int pid = <process id>;
     if (value == 1) {
          queue1.add(pid);
          suspend(pid);
     }
     value++;
     if (queue0.size() > 0) {
          int pid = queue0.remove();
          wakeup(pid);
     }
     [exit CS]
  }
}
```

Produttore e Consumatore

```
class SemaphorePC {
shared Object buffer;

Semaphore empty = new Semaphore(1);
Semaphore full = new Semaphore(0);

cobegin
   Producer  //  Consumer
coend
}
```

```
process Producer {
  while (true) {
    Object val = produce();
    empty.P();
    buffer = val;
    full.V();
  }
}
```

```
process Consumer {
  while (true) {
    full.P();
    Object val = buffer;
    empty.V();
    consume(val);
  }
}
```

Proprietà da Garantire:

- **Producer** non deve scrivere nuovamente l'area di memoria condivisa prima che Consumer abbia effettivamente utilizzato il valore precedente
- **Consumer** non deve leggere due volte lo stesso valore, ma deve attendere che Producer abbia generato il successivo
- **Assenza di Deadlock**

Buffer Limitato

```
class SemaphoreBuff_Limit{

Object buffer[SIZE];
int front = 0;
int rear = 0;
Semaphore mutex = new Semaphore(1);
Semaphore empty = new Semaphore(SIZE);
Semaphore full = new Semaphore(0);

cobegin
   Producer // Consumer
coend
}
```

```
process Producer {
     while (true) {
     Object val = produce();
     mutex.P();
     buf[front] = val;
     front = (front + 1) % SIZE;
     mutex.V();
     full.V();
     }
}
```

```
process Consumer {
     while (true) {
     full.P();
     mutex.V();
     Object val = buf[rear];
     rear = (rear + 1) % SIZE;
     mutex.V()
     empty.V();
     consume(val);
     }
}
```

Array circolare:
- si utilizzano *due indici* **front** e **rear** che indicano rispettivamente il **prossimo elemento da scrivere** e il **prossimo elemento da leggere**
- gli indici vengono **utilizzati in modo ciclico** (modulo l'ampiezza del buffer)

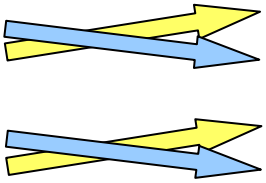
Proprietà da garantire:
- **Producer** non deve sovrascrivere elementi del buffer prima che Consumer abbia effettivamente utilizzato i relativi valori
- **Consumer** non deve leggere due volte lo stesso valore, ma deve attendere che Producer abbia generato il successivo
- **assenza di deadlock**
- **assenza di starvation**

Filosofi a Cena

```
class SemaphorePhilo{

Semaphore chopsticks =  {new Semaphore(1),
                         new Semaphore(1),
                         new Semaphore(1),
                         new Semaphore(1)};
```

```
process Philo[0] {
 while (true) {
  think
  chopstick[1].P();  //sx
  chopstick[0].P();  //dx
  eat
  chopstick[1].V();  //sx
  chopstick[0].V();  //dx
 }
}
```



```
process Philo[i] {  /* i = 1...4 */
     while (true) {
          think
  //dx    chopstick[i].P();
  //sx    chopstick[(i+1)%5].P();
          eat
  //dx    chopstick[i].V();
  //sx    chopstick[(i+1)%5].V();
          }
}
```

Lettori e Scrittori

```
class SemaphoreRW{

/* Variabili condivise */
int nr = 0;
Semaphore rw = new Semaphore(1);
Semaphore mutex = new Semaphore(1);

void startRead() {
  mutex.P();
  if (nr == 0)
    rw.P();
  nr++;
  mutex.V();
}
void startWrite() {
  rw.P();
}
```

```
void endRead() {
  mutex.P();
  nr--;
  if (nr == 0)
    rw.V();
  mutex.V();
}
void endWrite() {
  rw.V();
}
```

Monitor

Solo un processo alla volta può essere all'interno del monitor
- poter sospendere i processi in attesi di qualche condizione
- far uscire i processi dalla mutua esclusione mentre sono in attesa
- permettergli di rientrare quando la condizione è verificata

Dichiarazione di variabili di condizione (CV):

- **condition c;**

Le operazioni definite sulle CV sono:

- **c.wait()**
  - viene rilasciata la mutua esclusione
  - il processo che chiama **c.wait()** viene sospeso in una coda di attesa della condizione **c**
- **c.signal()**
  segnala che la condizione e' vera
  - causa la riattivazione immediata di un processo(secondo FIFO)
  - il chiamante viene posto in attesa
  - verra' riattivato quando il processo risvegliato avra' rilasciato la mutua esclusione (con unam **wait()**).
  - se nessun processo sta attendando c la chiamata non avrà nessun effetto

Semafori attraverso i Monitor

```
monitor Semaphore {
  int value;
  condition c;        /* value > 0 */

  Semaphore(int init) {
    value = init;
  }

  procedure entry void P() {
    value--;
    if (value < 0)
      c.wait();
  }

  procedure entry void V() {
    value++;
    c.signal();
  }


}
```

Lettori e Scrittori attraverso i Monitor

```
monitor RWController{
  int nr;               /* number of readers */
  int nw;               /* number of writers */
  condition okToRead;  /* nw == 0 */
  condition okToWrite; /* nr == 0 && nw == 0 */

     RWController() { /* Constructor */
       nr = nw = 0;
     }

     procedure entry void startRead() {
       if (nw != 0) okToRead.wait();
       nr = nr + 1;
       okToRead.signal();
     }

     procedure entry void endRead() {
       nr = nr - 1;
       if (nr == 0) okToWrite.signal();
     }

     procedure entry void startWrite() {
       if (!(nr=0 && nw =0)) okToWrite.wait();
       nw = nw + 1;
     }

     procedure entry void endWrite() {
       nw = nw - 1;
       okToRead.signal();
       if (nw == 0 && nr == 0) okToWrite.signal();
     }
}
```

Produttore/Consumatore attraverso Monitor

```
monitor PCController {
  Object buffer;
  condition empty;
  condition full;
  boolean   isFull;

  PCController() {
    isFull=false;
  }

procedure entry Object read() {
  if (!isFull)
    full.wait();
  int retvalue = buffer;
  isFull = false;
  empty.signal();
  return retvalue;
}
```

```
procedure entry void write(int val)
  {
    if (isFull)
      empty.wait();
    buffer = val;
    isFull = true;
    full.signal();
  }
}
```

Buffer Limitato attraverso i Monitor

```
monitor PCController {
  Object[] buffer;
  condition okRead, okWrite;
  int count, rear, front;

  PCController(int size) {
    buffer = new Object[size];
    count = rear = front = 0;
  }

  procedure entry Object read() {
      if (count == 0)
          okRead.wait();
      int retval = buffer[rear];
      cont--;
      rear = (rear+1) % buffer.length;
      okWrite.signal();
      return retval;
  }

  procedure entry void write(int val){
       if (count == buffer.length)
         okWrite.wait();
       buffer[front] = val;
       count++;
       front = (front+1) % buffer.length;
       okRead.signal();
     }
}
```

Filosofi a cena attraverso Monitor

```
monitor DPController {
  condition[] oktoeat = new condition[5];
  boolean[]   eating = new boolean[5];

  DPcontroller() {
    for(int i=0; i<5; i++) eating[i] = false;
  }

  procedure entry void startEating(int i) {
    if (eating[i-1] || eating[i+1]) oktoeat[i].wait();
    eating[i] = true;
  }

  procedure entry void finishEating(int i) {
    eating[i] = false;
    if (!eating[i-2]) oktoeat[i-1].signal();
    if (!eating[i+2]) oktoeat[i+1].signal();
  }
}
```

Message Passing

Send Sincrona Bloccante: **ssend(msg, dest);**
il mittente **src** spedisce il messaggio **msg** al processo **dest**, restando bloccato fino a quando q non esegue l'operazione **sreceive(msg, src)**.

Receive Sincrona Bloccante: **msg = sreceive (src);**
il destinatario **dest** riceve il messaggio **msg** dal processo **src**; se il mittente (src) non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio. **src** può non essere specificato (utilizzando *).

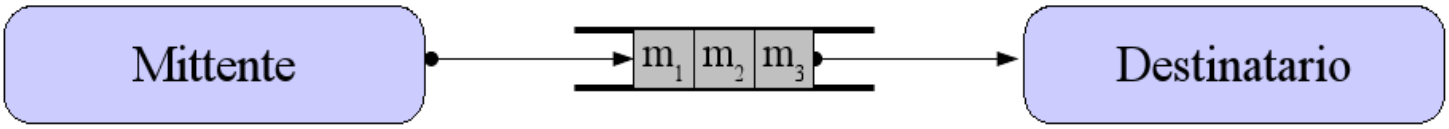
Send Asincrona NON Bloccante: **asend(msg, dest);**
il mittente **src** spedisce il messaggio **msg** al processo **dest**, senza bloccarsi in attesa di una **areceive(msg, src)** dal destinatario. I messaggi non ricevuti verranno aggiunti a una coda di attesa.

Receive Asincrona Bloccante: **msg = areceive(src);**
il destinatario **dest** riceve il messaggio **msg** dal processo **src**; se il mittente (src) non ha ancora spedito alcun messaggio, il destinatario si blocca in attesa di ricevere un messaggio. **src** può non essere specificato (utilizzando *).

Send Asincrona NON Bloccante: **asend(msg, dest);**
il mittente **src** spedisce il messaggio **msg** al processo **des**t, senza bloccarsi in attesa di una **areceive(msg, src)** dal destinatario. I messaggi non ricevuti verranno aggiunti a una coda di attesa.

Receive Asincrona NON Bloccante: **msg = nb-receive(src);**
il destinatario **dest** riceve il messaggio **msg** dal processo **src**; se il mittente (src) non ha ancora spedito alcun messaggio, ritornerà NULL. **src** può non essere specificato (utilizzando *).

Message passing asincrono



Message passing sincrono