Deshabilitar interrupciones.

Variables de candado

Alternancia Estricta

```
while (TRUE) {
 while(turn != 0) /* esperar */ ;
 critical_region();
 turn = 1;
 noncritical_region();
}
```

(a)

```
while (TRUE) {
 while(turn != 1) /* esperar */ ;
 critical_region();
 turn = 0;
 noncritical_region();
}
```

(b)

Solución de Peterson

```
#define FALSE       0
#define TRUE        1
#define N           2                   /* número de procesos */

int turn;                               /* ¿a quién le toca? */
int interested[N];                      /* todos los valores son inicialmente 0 (FALSE) */

void enter_region(int process);         /* proceso 0 o 1 */
{
 int other;                             /* número del otro proceso */

 other = 1 – process;                   /* lo opuesto de process */
 interested[process] = TRUE;            /* mostrar interés */
 turn = process;                        /* establecer bandera */
 while (turn == process && interested[other] == TRUE) /* instrucción nula */ ;
}

void leave_region(int process)          /* process: quién sale */
{
 interested[process] = FALSE;           /* indicar salida de la región crítica */
}
```

La instrucción TSL

```
enter_region:
        tsl register,lock       | copiar lock en register y asignarle 1
        cmp register,#0         | ¿era lock 0?
        jne enter_region        | si no era cero, se asignó 1 a lock, y se ejecuta el ciclo
        ret                     | volver al invocador; se entró en la región crítica


leave_region:
        move lock,#0            | guardar un 0 en lock
        ret                     | volver al invocador
```

Dormir y despertar

El problema de productor-consumidor

```
#define N 100                                   /* número de ranuras del buffer */
int count = 0;                                  /* número de elementos en el buffer */

void producer(void)
{
 while (TRUE) {                                 /* repetir indefinidamente */
        produce_item();                         /* generar el siguiente elemento */
        if (count == N) sleep();                /* si el buffer está lleno, dormir */
        enter_item();                           /* colocar elemento en el buffer */
        count = count + 1;                      /* incrementar la cuenta de elementos
                                                   en  el buffer */
        if (count == 1) wakeup(consumer);       /* ¿estaba vacío el buffer? */
 }
}

void consumer(void)
{
 while (TRUE){                                  /* repetir indefinidamente */
        if (count == 0) sleep();                /* si el buffer está vacío, dormir */
        remove_item();                          /* remover elemento del buffer */
        count = count –1;                       /* decrementar la cuenta de elementos
                                                   en el buffer */
        if (count == N–1) wakeup(producer);     /* ¿estaba lleno el buffer? */
        consume_item();                         /* imprimir elemento */
 }
}
```

Semáforos

Resolución del problema de productor-consumidor usando semáforos

```
#define N 100                                   /* número de ranuras del buffer */
typedef int semaphore;                          /* los semáforos son un tipo especial de int */
semaphore mutex = 1;                            /* controla el acceso a la región crítica */
semaphore empty = N;                            /* cuenta las ranuras de buffer vacías */
semaphore full = 0                              /* cuenta las ranuras de buffer llenas */

void producer(void)
{
 int item;

 while (TRUE) {                                 /* TRUE es la constante 1 */
        produce_item(&item);                    /* generar algo para ponerlo en el buffer */
        down(&empty);                           /* decrementar el contador empty */
        down(&mutex);                           /* entrar en la región crítica */
        enter_item(item);                       /* colocar el nuevo elemento en el buffer */
        up(&mutex);                             /* salir de la región crítica */
        up(&full);                              /* incrementar el contador de ranuras llenas */
 }
}

void consumer(void)
{
 int item;

 while (TRUE) {                                 /* ciclo infinito */
        down(&full);                            /* decrementar el contador full */
        down(&mutex);                           /* entrar en la región crítica */
        remove_item(&item);                     /* sacar elemento del buffer */
        up(&mutex);                             /* salir de la región crítica */
        up(&empty);                             /* incrementar el contador de ranuras vacías */
        consume_item(item);                      /* hacer algo con el elemento */
 }
}
```

Monitores

Mensajes.

Permite a los procesos intercambiar aquella información que sea necesaria durante el desarrollo normal de su ejecución. Es más un mecanismo de cooperación que de sincronización. El proceso que envía un mensaje lo deposita en una zona de memoria compartida y el receptor lo lee de ella. Las directrices de envío y recepción establecen una sincronización entre los procesos al tener que esperar por dichos mensajes, antes de continuar la ejecución.
Designación directa: siempre que se realice una operación con mensajes, cada emisor debe designar al receptor específico y viceversa, cada receptor debe especificar el emisor del que desea recibir un mensaje. Se asocia un enlace bidireccional único entre cada dos procesos.

Comunicación simétrica.

Send (A, mensaje) envia un mensaje al proceso A.
Receive (B, mensaje) recibe un mensaje del proceso B.

Comunicación asimétrica.

Send (P, mensaje) envia un mensaje al proceso P.
Receive (id, mensaje) recibe un mensaje de cualquier proceso, id se carga con el identificador del proceso que comunica.

Designación indirecta: los mensajes se envían y se recogen desde depósitos específicamente dedicados a ese propósito. Conocidos como buzones. Para que dos procesos puedan comunicarse deben utilizar el mismo buzón. El SO debe proveer herramientas para crear y destruir buzones.

Send (buzon1, mensaje) envia un mensaje al buzon 1.
Receive (buzon2, mensaje) recoge un mensaje del buzón 2.

Hace uso de dos primitivas, SEND y RECEIVE, las cuales son llamadas al sistema más que construcciones de un lenguaje.

Desventajas:

La red puede perder los mensajes. Se puede implementar que el receptor envíe un reconocimiento al emisor que recibió el mensaje. Si el emisor no recibe dicho reconocimiento pasado un tiempo reenvía el mensaje original. Si se pierden todos los reconocimientos el emisor enviará repetidas veces el mensaje original. Para solucionar esto usar números para identificar los mensajes.
Otra cuestión es como se denominan los procesos, de modo que sean únicos. El esquema de nominación generalmente es process@machine.domain.
La autenticación. Con frecuencia puede ser de utilidad proteger los mensajes con una clave que solo conozcan los usuarios autorizados.
Rendimiento. La reproducción de mensajes de un proceso a otro es más lenta que realizar una operación con semáforos o entrar en un monitor.

Resolución del problema de productor-consumidor usando mensajes

```
#define N 100                                   /* número de ranuras del buffer */

void producer(void)
{
 int item;
 message m;                                     /* buffer de mensaje */

 while (TRUE) {
        produce_item(&item);                    /* generar algo que poner en el buffer */
        receive(consumer, &m);                  /* esperar que llegue un mensaje vacío */
        build_message(&m, item);                /* construir un mensaje para enviar */
        send(consumer, &m);                     /* enviar elemento al consumidor */
 }
}

void consumer(void)
{
 int item, i;
 message m;

 for(i = 0; i < N; i++) send(producer, &m);     /* enviar N mensajes vacíos */
 while (TRUE) {
        receive(producer, &m);                  /* obtener mensaje que contiene elemento */
        extract_item(&m, &item);                /* extraer elemento del mensaje */
        send(producer, &m);                     /* devolver una respuesta vacía */
        consume_item(item);                     /* hacer algo con el elemento */
 }
}
```