2.5 Transferencia de mensajes.

Esa otra cosa es la **transferencia de mensajes**. Este método de comunicación entre procesos utiliza dos primitivas **SEND** y **RECEIVE** que, al igual que los semáforos y a diferencia de los monitores, son **llamadas al sistema** y no construcciones del lenguaje. Como tales, es fácil colocarlas en procedimientos de biblioteca, como:

```
send(destino, &mensaje);
receive(origen, &mensaje);
```

La primera llamada envía un mensaje a un destino dado, y la segunda recibe un mensaje de un origen dado (o de cualquiera [**ANY**], si al receptor no le importa). Si no hay un mensaje disponible, el receptor podría bloquearse hasta que uno llegue. Como alternativa, podría regresar de inmedia-to con un código de error.

Aspectos de diseño de los sistemas de transferencia de mensajes.

Los sistemas de transferencia de mensajes tienen muchos problemas y aspectos de diseño complicados que no se presentan con los semáforos ni con los monitores, sobre todo si los procesos en comunicación están en diferentes máquinas conectadas por una red. Por ejemplo, se pueden perder mensajes en la red. Para protegerse contra la pérdida de mensajes, el emisor y el receptor pueden convenir que, tan pronto como se reciba un mensaje, el receptor enviará de regreso un mensaje especial de acuse de recibo o confirmación. Si el emisor no recibe el acuse dentro de cierto intervalo de tiempo, retransmitirá el mensaje.

Consideremos ahora lo que sucede si el mensaje en sí se recibe correctamente, pero se pierde el acuse de recibo. El emisor retransmitirá el mensaje, de modo que el receptor lo recibirá dos veces. Es indispensable que él receptor pueda distinguir un mensaje nuevo de la retransmisión de uno viejo, Por lo regular, este problema se resuelve incluyendo números de secuencia consecutivos en cada mensaje original. Si el receptor recibe un mensaje que tiene el mismo número de secuencia que uno anterior, sabrá que el mensaje es un duplicado y podrá ignorarlo,

Los sistemas de mensajes también tienen que resolver la cuestión del nombre de los procesos, a fin de que el proceso especificado en una llamada **SEND** o **RECEIVE** no sea ambiguo. La verificación de autenticidad es otro problema en los sistemas de mensajes: ¿cómo puede el cliente saber que se está comunicando con el verdadero servidor de archivos, y no con un impostor?

En el otro extremo del espectro, hay aspectos de diseño que son importantes cuando el emisor y el receptor están en la misma máquina. Uno de éstos es el rendimiento. El copiado de mensajes de un proceso a otro siempre es más lento que efectuar una operación de semáforo o entrar en un monitor.

El problema de productor-consumidor con transferencia de mensajes.

Veamos ahora cómo puede resolverse el **problema de productor-consumidor** usando **transferencia de mensajes** y sin compartir memoria. En el **Programa 2.9** se presenta una solución. Suponemos que todos los mensajes tienen el mismo tamaño y que el sistema operativo coloca automáticamente en buffers los mensajes enviados pero aún no recibidos. En esta solución se usa un total de **N mensajes**, análogos a las **N ranuras** de un buffer en memoria compartida. El **consumidor** inicia enviando **N mensajes** vacíos al productor. Cada vez que el productor tiene un elemento que entregar al consumidor, toma un mensaje vacío y devuelve uno lleno. De este modo, el número total de mensajes en el sistema permanece constante y pueden almacenarse en una cantidad de memoria que se conoce con antelación.

Si el **productor** trabaja con mayor rapidez que el consumidor, todos los mensajes quedarán llenos, esperando al consumidor; el productor se bloqueará, esperando la llegada de un mensaje vacío. Si el consumidor trabaja con mayor rapidez, ocurre lo opuesto: todos los mensajes estarán vacíos esperando que el productor los llene; el consumidor estará bloqueado, esperando un mensaje lleno.

```
#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_tem(&m, &item);          /* Extraer elemento del mensaje          */
      send(producer, &m);              /* Devolver una respuesta vacía          */
      consume_item(item);              /* Hacer algo con el elemento            */
      }
}
```

Programa 2.9: Problema de productor-consumidor con **N mensajes**.

La **transferencia de mensajes** puede tener muchas variantes. Para comenzar, veamos cómo se dirigen los mensajes. Una forma es asignar a cada proceso una dirección única y hacer que los mensajes se dirijan a los procesos. Un método distinto consiste en inventar una nueva estructura de datos, llamada **buzón**. Un **buzón** es un lugar donde se almacena temporalmente cierta cantidad de mensajes, que normalmente se especifican cuando se crea el buzón. Si se usan buzones, los parámetros de dirección de las llamadas **SEND** y **RECEIVE** son buzones, no procesos. Cuando un proceso trata de transmitir a un buzón que está lleno, queda suspendido hasta que se retira un mensaje de ese buzón, dejando espacio para uno nuevo.

En el caso del **problema de productor-consumidor**, tanto el productor como el consumidor crearían buzones con espacio suficiente para **N** mensajes. El **productor** enviaría mensajes con datos al buzón del **consumidor**, y éste enviaría mensajes vacíos al buzón del **productor**. Si se usan buzones, el mecanismo de almacenamiento temporal es claro; el buzón de destino contiene mensajes que se han enviado al proceso de destino pero todavía no han sido aceptados. La otra forma extrema de manejar buzones es eliminar todo el almacenamiento temporal. Cuando se adopta este enfoque, si el **SEND** se ejecuta antes que el **RECEIVE**, el proceso emisor queda bloqueado hasta que ocurre el **RECEIVE**, y en ese momento el mensaje podrá copiarse directamente del emisor al receptor, sin buffers intermedios. De forma similar, si el **RECEIVE** se ejecuta primero, el receptor se bloquea hasta que ocurre el **SEND**. Esta estrategia se conoce como **cita** o **rendezvous**; es más fácil de implementar que un esquema de mensajes con almacenamiento temporal, pero es menos flexible, pues se obliga al emisor y al receptor a operar estrictamente sincronizados.

La comunicación entre los procesos de usuario en **UNIX** se efectúa a través de **conductos**, que efectivamente son buzones. La única diferencia real entre un sistema de mensajes con buzones y el mecanismo de **conductos** es que los conductos no preservan los límites de los mensajes. Dicho de otro modo, si un proceso escribe **10 mensajes** de **100 bytes** cada uno en un conducto y otro proceso lee **1,000 bytes** de ese conducto, el lector obtendrá los **10 mensajes** a la vez. Con un verdadero sistema de mensajes, cada **READ** debería devolver sólo un mensaje. Desde luego, si los procesos convienen en leer y escribir siempre mensajes de tamaño fijo del conducto, o en terminar cada mensaje con un carácter especial (salto de línea), no habrá problema.