Laboratorio de Arquitectura de Computadoras
Problema Productor-Consumidor

Jorge Miguel Machado Ottonelli - 4.876.616-9

Noviembre 2021

Facultad de Ingeniería. Universidad de la República
Montevideo, Uruguay

Resumen

Propuesta y resolución de una variación del problema de sincronización "Productor - Consumidor" en lenguaje Assembler para la arquitectura 8086, con un fuerte énfasis en la interacción con el sistema de entrada y salida (E/S) y el uso de las interrupciones.

Keywords: Problema Productor-Consumidor, Assembler, Aquitectura 8086, First-In-First-Out.

Índice

1. Introducción

Este trabajo plantea una variación del problema de sincronización "Productor - Consumidor", así como una solución en lenguaje Assembler (archivo anexo *obligatorio.asm*) para la arquitectura 8086, con la que se pretende aprovechar la herramienta de interrupciones del sistema en la solución del mismo.

Antes de comenzar, se describe en que consiste este problema de sincronización de manera general, de la siguiente manera:

- Se presentan dos procesos, un productor y un consumidor, que comparten un buffer de tamaño finito.
- El productor debe generar el producto, almacenarlo y comenzar nuevamente con este procedimiento.
- De manera simultanea, el consumidor debe tomar uno a uno los productos del buffer.
- El problema consiste en que el productor no agregue productos al buffer si esta lleno, y que el consumidor no tome productos si esta el buffer vacío.

Con esto, ya se presenta una idea general a lo que se enfrentará en la variante planteada.

1.1. Variante del problema (múltiples parejas productor-consumidor)

Esta variante del problema productor-consumidor plantea la existencia de múltiples parejas de productores y consumidores en un sistema mediado por un *controlador*, el cual sera el encargado de transferir la información de manera consistente entre los pares productor-consumidor.

El sistema presenta P productores y C consumidores que envían o reciben información (en este caso caracteres) a través de sus respectivos puertos de E/S (entrada/salida). Los productores envían esta información a un ritmo variable, mientras los consumidores la reciben a una tasa fija de tics del reloj del sistema.

Cada pareja productor-consumidor es conformada al ingresar una secuencia de comandos por el puerto *ESTADO* del sistema (antes de que este comience la transferencia de datos entre los pares), la cual indica que el puerto productor i es asignado con el puerto consumidor j, como también el puerto de control del productor i y la tasa fija de tics en la cual el consumidor j debe recibir datos. Esta pareja compartirá un buffer de caracteres de tamaño *MAX_BUFFER* en memoria y sigue el método de gestión *FIFO* (First-In First-Out).

Las tareas del *controlador* se divide de la siguiente manera:

1. Cada vez que el bit menos significativo del puerto de control del productor i se "encienda" ($= 1$), significará la presencia de un nuevo carácter disponible por el productor i. El *controlador* deberá ingresar el dato al buffer del par que contiene al productor i si este no esta lleno. En el caso en el que el buffer este lleno, el carácter se descarta y el *controlador* retorna en el puerto *ESTADO* la existencia de este *OVERFLOW* (imprime el puerto del productor).

2. Cada vez que hallan transcurrido la cantidad de tics de la tasa fija del consumidor j, deberá enviarle a este el dato más antiguo presente en el buffer (garantizando que se cumpla la política *FIFO*), en caso en el que el buffer este vacío se le retornará al consumidor que ha ocurrido un *UNDERFLOW* (retornando $'0'$ al consumidor).

2. Solución propuesta

En esta sección, se propone una solución en alto nivel para la variación del problema propuesto anteriormente, se brinda una descripción de la misma y del diseño de la estructura de datos y algoritmos utilizados.

2.1. Solución en alto nivel

```
    // Se definen las constantes para las funciones:
    #define ESTADO equ 128
    #define UNDERFLOW equ 0x0
    #define HAB equ 0x1
    #define CON equ 0x2
    #define MAX_BUFFER equ ...

    void principal() {
1)      List<ProducerConsumer> listaPares = new List<ProducerConsumer>();
2)      cargarDatos(listaPares);
3)      // Deshabilitar las interrupciones.
4)      // Instalar la interrupción 'consumir(listaPares)' como INT 8
        //  (Timer) en el vector de interrupciones.
5)      // Habilitar las interrupciones.
6)      while (true) {
7)          producir(listaPares);
        } // endWhile
    } // end.
```

Se observa en detalle el algoritmo principal. Lo primero que se realiza es la inicialización y carga de una *Lista* (*listaPares*) de objetos de tipo *ProducerConsumer* (líneas 1 y 2), ésta estructura representa y guarda los datos de la pareja Productor-Consumidor (se detalla en profundidad en la siguiente sección 2.1.4).

Luego de cargar todas las parejas al sistema, se procede a deshabilitar las interrupciones para poder instalar *consumir(listaPares)* como interrupcion 8 (*Timer*) del programa (línea 4), de esta manera se asegura que pasada cierta cantidad de instrucciones equivalentes a un *tic* se chequee si un consumidor debe consumir un dato del buffer. Después de instalado, se vuelven a habilitar las interrupciones.

Para finalizar con el algoritmo principal, se tiene un bucle infinito a causa del *while(true)* que contiene una sola operación a realizar por cada iteración (líneas 6 y 7), *producir(listaPares)*. Ésta operación será la encargada de que para cada Productor contenido en

un par Productor-Consumidor, se chequee si este tiene un nuevo dato para agregar al buffer, y en caso de ser afirmativo guardar este dato al final del buffer asociado al par del Productor.

A continuación, se presenta una descripción en detalle de las funciones auxiliares y la operación de interrupción. Se debe tener en cuenta que las constantes y estructuras definidas en esta sección son las mismas.

2.1.1. Función cargarDatos()

Desde esta sección en adelante se utiliza la operación de entrada de datos *in(var, port)*, que obtiene el dato presente en el puerto de lectura *'port'* y lo guarda en la variable *'var'*.

```
    void cargarDatos(List<ProducerConsumer> listaPares) {
1)      int comando;
2)      do {
3)          in(comando, ESTADO);
4)          if (comando == CON) {
5)              ProducerConsumer nuevoPar = new ProducerConsumer();
6)              int dato;
7)              in(dato, ESTADO);
8)              nuevoPar.puertoProducer = dato;
9)              in(dato, ESTADO);
10)             nuevoPar.puertoControl = dato;
11)             in(dato, ESTADO);
12)             nuevoPar.puertoConsumer = dato;
13)             in(dato, ESTADO);
14)             nuevoPar.tics = dato;
15)             nuevoPar.timer = 0;
16)             nuevoPar.tamBuffer = 0;
17)             nuevoPar.firstElem = 0;
18)             nuevoPar.lastElem = 0;
19)             nuevoPar.buffer = new char[MAX_BUFFER];
20)             listaPares.add(nuevoPar);
            } // endIf
        } while (comando != HAB);
    } // end.
```

Esta función es la encargada de leer los datos ingresados al puerto *ESTADO* y generar los pares Productor-Consumidor en el sistema. Gracias a la condición del bucle del *do* se podrán ingresar nuevas parejas hasta que se ingrese el comando de inicio (*HAB*).

Luego, dentro de cada iteración del bucle se chequea que se ingrese el comando *CON* (línea 4), el cual indica que los siguientes cuatro datos ingresados serán la información del nuevo par Productor-Consumidor a añadir al sistema. Al ingresar al bloque del *if* se observa que de las líneas 5 a la 20 se genera un nuevo dato del tipo ProducerConsumer que contendrá la información del par ingresado y un buffer con tamaño maximo *MAX_BUFFER*, para luego añadirla a la lista *listaPares*, pasada por parámetro desde la función principal del programa.

Se destaca el hecho de que esta función toma en cuenta de que los siguientes cuatro datos ingresados luego de utilizar el comando *CON* son correctos.

2.1.2. Función producir()

A partir de esta sección en adelante se utiliza la operación de salida de datos *out(port, var)*, que retorna el dato presente en la variable *'var'* al puerto de escritura *'port'*.

```
    void producir(List<ProducerConsumer> listaPares) {
1)      foreach (ProducerConsumer parPC in listaPares) {
2)          byte ctrl;
3)          in(ctrl, parPC.puertoControl);
4)          if (((ctrl & 1) == 1) && (ctrl != parPC.lastControl)) {
5)              if (parPC.tamBuffer < MAX_BUFFER) {
6)                  parPC.tamBuffer++;
7)                  if (parPC.lastBuffer != MAX_BUFFER) {
8)                      parPC.lastBuffer++;
9)                  } else {
10)                     parPC.lastBuffer = 0;
                    } // endElse
11)                 in(dato, parPC.puertoProducer);
12)                 parPC.buffer[parPC.lastBuffer] = dato;
13)             } else {
14)                 out(ESTADO, parPC.puertoProducer);
                } // endElse
            } // endIf
15)         parPC.lastControl = ctrl;
        } // endForeach
    } // end.
```

Esta operación se encarga de verificar la existencia de un nuevo dato en el puerto del consumidor (el bit menos significativo del puerto control sea igual a 1), y en caso afirmativo lo guardará al final del buffer del par Productor-Consumidor correspondiente, siempre y cuando el buffer no este lleno. Se observa que esta verificación e inserción se realiza para cada par existente, ya que se itera para todos los pares contenidos en la lista *listaPares* (línea 1).

Lo anterior se aplica de la siguiente manera, en la línea 3 obtenemos el dato de control y luego puede ocurrir uno de los siguientes casos:

1. **El bit menos significativo de control es igual a 1 y el buffer no esta lleno:** en este caso incrementamos la posición del ultimo elemento del buffer, si la posición antes del incremento era la casilla final del buffer, movemos la posición al inicio del buffer (no a la posición del primer elemento!). Luego, ingresamos el dato producido por el productor en la posición incrementada anteriormente.

2. **El bit menos significativo de control es igual a 1 y el buffer esta lleno:** en este caso descartamos el dato producido por el productor (ya que no podemos añadirlo

al buffer lleno), y retornamos por el puerto *ESTADO* el puerto del productor que verificamos (indicando el *OVERFLOW*).

3. **El bit menos significativo de control no es 1:** este caso significa que el productor aún no tiene un nuevo dato disponible, por lo tanto no se hace nada.

2.1.3. Interrupción consumir()

La siguiente operación es la cargada en la interrupción 8 (*Timer*) del programa, se escoge esta ya que los consumidores consumen a una tasa fija de tics, donde un tic representa cierta cantidad de instrucciones ejecutadas por el programa.

```
    void consumir(List<ProducerConsumer> listaPares) {
1)      foreach (ProducerConsumer parPC in listaPares) {
2)          parPC.timer++;
3)          if (parPC.timer == tics) {
4)              parPC.timer = 0;
5)              if (parPC.tamBuffer > 0) {
6)                  out(parPC.puertoConsumer, parPC.buffer[parPc.firstElem]);
7)                  parPC.tamBuffer--;
8)                  if (parPC.firstElem != MAX_ELEM) {
9)                      parPC.firstElem++;
10)                 } else {
11)                     parPC.firstElem = 0;
                    } // endElse
12)             } else {
13)                 out(parPC.puertoConsumer,UNDERFLOW);
                } // endElse
            } endIf
        } // endForeach
    } // end.
```

Se observa la idea sencilla que plantea esta operacion, para cada par Productor-Consumidor en la lista de pares, incrementa en uno el timer "interno" del par, y si éste luego del incremento es igual a la tasa de tics del consumidor, reseteamos el timer a cero y realizamos una de las siguientes acciones:

1. **Si el buffer no está vacío:** obtenemos el primer elemento del buffer (por la política *FIFO*) para luego enviarlo al puerto del consumidor; antes de terminar se incrementa en uno el marcador del primer elemento, y si la posición anterior era el final del buffer lo movemos al inicio de éste (de la misma manera que se realizo con el marcador del último elemento en la función anterior).

2. **El buffer esta vacío:** unicamente retornamos al puerto del consumidor el valor $'0'$ (indicando de esta manera un *UNDERFLOW*).

2.1.4. Estructura de datos ProducerConsumer

La siguiente estructura representa un par Productor-Constructor, este contiene no solo la información de los puertos y tasa de tics ingresadas, sino que también, el buffer de tamaño máximo *MAX_BUFFER* compartido por el par con la información auxiliar necesaria (como la posición actual del primer y ultimo elemento). Se detalla a continuación la definición de la estructura, como también la cantidad de *bytes* necesaria para cada elemento:

```
struct ProducerConsumer {
    int puertoProducer          // 1 byte
    int puertoControl           // 1 byte
    int lastControl             // 1 byte
    int puertoConsumer          // 1 byte
    int tics                    // 1 byte
    int timer                   // 1 byte
    int tamBuffer               // 2 bytes
    int firstElem               // 2 bytes
    int lastElem                // 2 bytes
    byte[MAX_BUFFER] *buffer    // MAX_BUFFER bytes
}
```

Observación: el campo *lastControl* representa el valor anterior tomado del puerto *puertoControl*, de esta manera se asegura de no tomar el valor del productor nuevamente si este no bajo a cero aún el bit menos significativo de este puerto.

2.2. Otras opciones analizadas:

Aunque para este trabajo la idea de como realizar la solución se mantuvo igual desde el inicio, si se ha planteado maneras distintas de como llevar la idea a la implementación. Una de estas variaciones planteadas era la manera en como llevar la cuenta actual de tics, la idea original era tener un timer global para todos los pares y comparar que el modulo entre el tiempo actual del timer y la tasa de tics sea igual a cero. Esta alternativa fue descartada por una que, aunque cueste mas espacio en la memoria del programa, resulta mas practica a la hora de implementar la solución en el lenguaje Assembler.

Otra alternativa planteada, era el realizar el control de los consumidores (la operación *consumir()*) en el bucle del *while(true)* del algoritmo principal, e instalar la operación *producir()* en la interrupción 8 (*Timer*). Esto causa un gran problema, ya que de esta manera no se puede garantizar el hecho de que los consumidores consuman a una tasa fija de tics.

3. Experimentación y problemas encontrados

Durante la experimentación con los casos de pruebas brindados no se llego a generar problemas en particular, la implementación en Assembler de la solución propuesta pudo ejecutar los casos sin dificultades o errores. Sin embargo, y con lo alentador que fueron los resultados anteriores, para la experimentación propia se trató de ejecutar casos de pruebas

límites al programa, con el fin de observar cuales son las limitaciones que posee actualmente, y poder brindar posibles mejoras a futuro.

Para esta experimentación se plantearon los considerados "peores casos limites", estos se plantean de la siguiente forma:

1. Se utiliza la mayor cantidad de pares soportada por el sistema.
2. La tasa fija de tics de los consumidores debe estar a 1, con el fin de ejecutar la mayor cantidad de instrucciones durante la interrupción.
3. Los productores producen rápidamente, es decir, deben insertar elementos cada dos interrupciones (debido a que el productor debe apagar el bit del puerto de control en algún momento), de esta manera se ejecutan la mayor cantidad de instrucciones de la operación *producir()*.

Antes de comenzar con la experimentación limite, se realizan pruebas para comprobar la mínima cantidad de instrucciones por tic en la interrupción que permiten el correcto funcionamiento de la solución para una sola pareja Productor-Consumidor. Se muestra primero la manera teórica de obtener una cota mínima de instrucciones, para esto dicha cota debe ser mayor o igual a la mayor cantidad de instrucciones ejecutadas dentro de *producir()* y en la interrupción *consumir()*, es decir:

$$instrucciones\_tic \geq ins(producir()) + ins(consumir()) \tag{1}$$

, donde $ins()$ es la cantidad de instrucciones de una operación.

Y como se tiene que, $ins(producir()) = 33 \times cantidad\_pares$, e $ins(consumir()) = 33 \times cantidad\_pares + 18$, se llega a:

$$instrucciones\_tic \geq 66 \times cantidad\_pares + 18 \tag{2}$$

, donde *cantidad_pares* es la cantidad de pares Productor-Consumidor en el sistema.

Por lo tanto, para una sola pareja se asegura que para valores mayores a 84 instrucciones por tic la solución es correcta. A su vez, esto se comprueba de manera práctica mediante los casos de un solo par ejecutado, donde además de funcionar correctamente para 84 o más instrucciones, también suele funcionar correctamente con una menor cantidad de instrucciones en casos de "Productor lento".

Por ultimo, se realizo una prueba exhaustiva para probar la mayor cantidad de canales con la que la solución logra desempeñarse, esta prueba se realizo con los parámetros de "peores casos limites" comentado anteriormente, con 999 instrucciones por tics en la interrupción (lo máximo soportado por el simulador). La solución logró desempeñarse correctamente para un total de 18 pares de Productor-Consumidor en el sistema, y donde el fallo ocurrido al agregar el par 19 no fue más que la perdida de un solo dato por parte de un solo par del sistema.

4. Conclusiones

Se concluye que aunque el planteamiento de la solución sea correcto, aún existen ciertas limitaciones y cuidados al tener en cuenta a la hora de implementarla en Assembler. Tanto

pequeños errores con el manejo de segmentos que podrían llevar a modificar datos no deseados en memoria, como también tener un control de la cantidad de instrucciones innecesarias utilizadas en las operaciones, que podrían llevar no solo a una mala optimización sino que también a problemas en soluciones que usen las interrupciones de *Timer* para ciertos cálculos (como se vio anteriormente en este trabajo).

La problemática planteada, no solo dio conocimiento sobre ésta y solución, sino que también sirvió como un nexo para la correcta comprensión del manejo de segmentos en Assembler con la implementación de un buffer, como también la importancia y gran aplicabilidad de la herramienta de interrupciones, capaces de irrumpir un proceso que se este ejecutando, pausarlo para ejecutar su propio código y luego volver al punto en el que se estaba en el proceso pausado.

Para finalizar, se comentan ciertas mejoras que se podrían realizar a esta implementación a futuro, que aunque cumpla con los requerimientos pedidos aún mantiene aspectos a mejorar.

4.1. Mejoras a futuro

Al analizar las posibles mejoras a realizar a futuro de la solución, se tienen dos aspectos esenciales a mejorar:

1. Reducir y optimizar la cantidad de instrucciones de cada operación, de esta manera la solución podría trabajar con aún mas pares Productor-Consumidor de manera correcta con una cantidad de instrucciones por tic más reducida.

2. Aproximar los segmentos de memoria asociados a cada par (y por lo tanto sus buffers), con la intención de tener aún más espacio para nuevos pares, como también tener una mejor optimización con respecto a los segmentos de memoria utilizados.

Aunque las limitaciones del simulador con respecto a la cantidad máxima de instrucciones por tic (máximo de 999 instrucciones) no nos permita la ejecución del programa para valores mayores, y comprobar de esta manera que tan "lejos" podría llegar, esto no exenta a realizar las mejoras para alcanzar una solución mas robusta y óptima.

Referencias

[1] Dijkstra, E. W. *Cooperating Sequential Processes*. 1965. https://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF. (pág. 31-33).

[2] Germán Galeano Gil, & Juan A. Gómez Puildo. *Tabla de interrupciones*. Grupo de Arquitectura de Computadores y Diseño Lógico. UEX, 1997. http://ebadillo_computacion.tripod.com/ensamblador/8086_int.pdf

[3] Daniel B. Sedory. *The Segment:Offset Addressing Scheme*. 2007. https://thestarman.pcministry.com/asm/debug/Segments.html. Accedido: 10/2021.