2021

LABORATORIO DE ARQUITECTURA DE COMPUTADORAS

PROBLEMA PRODUCTOR-CONSUMIDOR EN INTEL 8086

AGUSTÍN RECOBA CLAUDIO – C.I.: 5.469.187-5

Índice

Introducción

La consigna del trabajo es diseñar e implementar un programa para solucionar el problema de administrar múltiples canales productor-consumidor. Dicho programa debe ser capaz de recibir las n configuraciones del sistema y actuar acorde a ellas. Cada configuración de la 1 hasta la n consiste de:

- el puerto por el que se reciben los datos del canal (al que llamaré PPi),
- el puerto por el que se avisará que hay un dato listo para ser recibido (al que llamaré PPi_control),
- el puerto por el que se enviarán los datos (al que llamaré PCi), y
- la frecuencia a la que se deberán enviar los datos de este canal (en tics).

El programa tendrá un puerto ESTADO preestablecido por el cual recibe los datos de configuración y emite algunos de los errores de ejecución del programa. Además, se contará con las constantes MAX_BUFFER y CANT_CANALES que sirven como cota superior a la cantidad de elementos pendientes de ser enviados y la cantidad de canales que es posible configurar como máximo respectivamente.

Los datos deberán ser enviados de acuerdo a la política FIFO (first in, first out).

Se le llamó overflow al error producido al querer leer un dato de PPi y que el canal esté lleno (MAX_BUFFER elementos esperando a ser enviados). En este caso se definió que el dato leído deberá ser descartado y que se deberá enviar el valor de PPi al puerto ESTADO para avisar del error.

Se le llamó underflow al error producido al querer enviar un dato por PCi y que el canal esté lleno (0 elementos esperando para ser enviados). En este caso se definió que se deberá enviar el valor 0 por el puerto PCi.

El programa diseñado deberá ser implementado en el lenguaje ensamblador de 8086 dado en el curso, teniendo en cuenta que el sistema será usado exclusivamente para esta tarea (sistema dedicado).

Solución propuesta

Posibles soluciones

A grandes rasgos, toda solución del problema debe contener los siguientes módulos:

I. Inicialización del sistema y lectura de configuraciones.
II. Lógica de la lectura de datos por los PPi.
III. Lógica del mantenimiento de los datos que esperan a ser enviados.
IV. Lógica para enviar datos a PCi.
V. Conteo de tics del sistema.

Para decidir dónde (o cuándo) ejecutar cada módulo del programa, se pueden elegir tres opciones: el inicio del programa, un pooling iterativo infinito o en una interrupción. Para la parte I, parecería indiscutible que se debe hacer al comienzo del programa. Pasa lo mismo con la parte V, la unica manera de contar 'tics' es con una interrupción de tipo timer. Restaría ver qué opciones hay para II, III y IV.

Para III, teniendo en cuenta que la política de envío de datos es FIFO, resulta obvio que los datos habría que guardarlos en una especie de cola o queue y con esto tendríamos cubierta la lógica mencionada. En cuanto a implementación, un buffer circular como el visto en el curso puede ser lo más simple, pero también es posible implementarlo con listas doblemente enlazadas o simples.

Para II, parecería que tenemos al menos dos opciones: pooling o dentro de la interrupción de timer. Sin embargo, colocarlo en el timer no es correcto: si la frecuencia en la que se reciben datos (el tiempo entre dos flancos ascendentes del byte PPi_control) fuese menor a la frecuencia a la que interrumpe el timer, se perderían todos los datos que se mandaron entre cada interrupción. Por esto entiendo que la única alternativa es hacer la lectura de los datos e inserción en el buffer desde el pooling en el main().

Para IV tenemos más opciones y no estamos limitados como recién, ya que el consumo de datos siempre se realiza a una tasa fija mayor que un tic. Entonces se admiten ambas implementaciones: en el timer (luego de contar los tics) o en el pooling (luego o antes de la lectura de los datos).

Habiendo determinado las opciones para la lógica general, queda administrar el uso de variables y estructuras del programa. Es claro que los datos de configuración deben guardarse y quedar asociados entre sí de alguna manera, y lo mismo sucede con el buffer o la cola. En un principio esto se podría hacer de dos maneras: un struct que incluya todo (valores de configuración, buffer y variables asociadas al buffer) o un array para cada uno de los valores que incluía el struct mencionado (los elementos de los array quedan vinculados entre sí por el índice de acceso).

Solución elegida

Para el programa decidí colocar la lógica de envío de datos dentro del timer y no en el programa principal. Esto me permite mandar el dato apenas sé que es el turno de mandarlo. ¿Cómo sé cuándo es el turno de mandarlo? Esto lo hice introduciendo una nueva variable para cada canal: tics restantes (tics_rest) para envío. La variable se inicializa en la frecuencia de consumo definida en la configuración y se decrementa en uno por cada interrupción del timer. Cuando esta llega a cero, se llama al procedimiento encargado de mandar los datos a PCi. Luego de mandado el dato, se vuelve a asignar el valor inicial.

La lectura de las configuraciones se hace inmediatamente después de configurar el vector de interrupciones: es un while que corre hasta recibir la instrucción HAB o se hayan agregado CANT_CANALES. Se asume que la configuración se realiza correctamente.

La lectura de datos se realiza en el pooling del main(), iterando sobre los canales configurados y chequeando que no se haya producido un flanco ascendente en PP_control, para esto se necesita saber si antes hubo control 0 y para ello se guarda el byte controlAnt para cada canal.

Para mantener los datos que están esperando para enviarse, utilicé un array circular con política FIFO. Para esto mantengo dos índices del array: dondeQuitar y dondePoner. Y para controlar los casos borde (buffer lleno o vacío), también llevo la cuenta de la cantidad de datos que hay activos: cantIngresados.

La elección tomada sobre la representación de las estructuras en memoria es la segunda que propuse: un array para cada variable que precisa un canal para funcionar. Estas variables son: PP, PP_control, PC, frec_consumo, tics_rest, controlAnt, buffer, dondePoner, dondeQuitar y cantIngresados. Todas estas variables se pueden acceder con var[i], excepto el buffer. En el archivo se especifica cómo se accede al mismo pero, en resumen: es un array de largo CANT_CANALES*MAX_BUFFER donde cada buffer es contiguo al siguiente.

El siguiente pseudocódigo resume las decisiones tomadas:

```
1.  disable()
2.  Configurar el vector de interrupciones
3.  While (entrada i= HAB)
4.  |     Leer configuraciones
5.  |     Inicializar canales
6.  endwhile
7.  enable()
8.  while (true)
9.  |     Para cada canal configurado:
10. |     |     Si (hay flanco ascendente en PP_control):
11. |     |     |     Agregar dato al buffer.
12. |     |     Guardar el dato PP_control
13. fin while
14.
15. Procedimiento timer():
16. Para cada canal configurado:
17. |     Decrementar tics restantes
18. |     Si (tics restantes == 0):
19. |     |     Quitar dato del buffer.
20. |     |     Enviar el dato por PC
21. |     |     tics restantes := frecuencia consumo.
22. |     fin si
23. fin procedimiento
```

Nota: Quitar dato y Agregar dato deben verificar las condiciones de underflow y overflow respectivamente.

Aspectos pendientes o irresueltos

Primero, el algoritmo puede quedar en un loop infinito en el que enviará todos los datos actualmente almacenados y luego dará underflow eternamente. Esto sucede cuando la rutina del timer "dura" más interrupciones que la frecuencia de interrupción del mismo: cada vez que complete una interrupción, entrará inmediatamente otra vez y así hasta que suceda un error con el stack o algo por el estilo. La solución a esto sería que el timer interrumpa cada más instrucciones o que la rutina asociada sea más corta.

Segundo, de querer escalar el sistema y hacer que este se encargue de otras funciones (ya no sería dedicado), deberíamos tener que mover toda la lógica a las interrupciones. Esto puede empeorar el problema recién mencionado y sucede lo que se mencionó cuando exploré la posibilidad de colocar la lectura de datos en el timer. Una solución a ambos problemas sería asociar una interrupción al PP_control de cada canal. De esa manera se podría leer cada dato apenas se detecta el flanco y se reduciría la carga de instrucciones en el timer.

El último problema irresuelto es el hecho de que el problema es escalable en los parámetros MAX_BUFFER y CANT_CANALES siempre que la memoria lo permita. Se pueden distribuir los datos entre los distintos segmentos para aprovechar al máximo la memoria, pero siempre está el limitante de 1MB. El uso de dispositivos de almacenamiento por E/S y usar la memoria solo para los elementos que están próximos a ser utilizados (una especie de caché) podría ser una solución a este problema.

Pruebas realizadas

Se realizaron pruebas del algoritmo presentado a través del software ArquiSim en su versión 1.3.6. Las pruebas tuvieron como objetivo testear las funcionalidades básicas del programa, sin usar más de 16 canales en simultáneo y con MAX_BUFFER en 8 máximo.

Descripción de las pruebas

Cada prueba consistió en:

- un valor para el parámetro MAX_BUFFER,
- la cantidad de instrucciones entre cada interrupción del timer,
- qué colocar en la sección .ports del simulador (esto incluye las configuraciones, los flancos de PP_control y los valores de PP), y
- qué salida se debería obtener en la consola al ejecutar el programa

Además de las pruebas dadas por los docentes, realicé pruebas extra modificando las dadas y observando si las modificaciones alteraban la salida como yo esperaba. La mayoría consistió en alterar el parámetro MAX_BUFFER para buscar si ocurría overflow donde no debería ocurrir. Otros tantos de pruebas consistieron en agregar más canales y verificar si el programa soportaba escalarlo de esta manera.

No realicé pruebas que estuvieran fuera del foco del trabajo, por ejemplo, configurar mal los canales o asignar el mismo puerto productor o consumidor a un mismo canal.

Análisis de los resultados obtenidos

Los resultados obtenidos son conforme a las salidas esperadas. Lo mismo para los test propios.

Conclusiones

El programa realizado y su performance me resultó satisfactorio, sin embargo, mi acercamiento al trabajo me llevó a bastantes frustraciones. Programé todo el programa en C antes de pasarlo a ensamblador y aunque al principio pareció ser una estrategia provechosa, resultó una pérdida de tiempo. En total realicé cinco versiones del programa, cada una con cambios importantes en la lógica y siempre con la esperanza de que esa versión fuera la 'definitiva'. Por ejemplo: implementé la versión con un struct para cada canal en vez de usar arrays, probé a colocar la lógica de consumidor y productor en el pooling y en la interrupción, intenté usar un contador de tics global y consumir cuando (tics % frec_consumo) == 0, entre otras opciones más.

Entre los errores que cometí se encuentran: confundir una L con una X y leer de a palabra en vez de a byte, sobreescribir el vector de interrupciones con los elementos cargados en el buffer, sobreponer el stack con las variables en memoria y no guardar el contexto adecuadamente al usar procedimientos.

Gracias al tiempo dedicado al trabajo puedo decir que aprendí de todos estos problemas y que en el futuro seré menos propenso a repetirlos. Sin embargo, sigue habiendo lagunas en mi conocimiento sobre la arquitectura y el set de instrucciones dado: por ejemplo, no logré usar satisfactoriamente las operaciones DIV y MUL pese a haber realizado múltiples intentos, incluso algunas de las decisiones de diseño del programa se vieron influenciadas por esto.