Resúmen: Primer Mitad de Dijkstra EWD123

Alumno: Axel Daniel Malváez Flores

Materia: Computación Concurrente

Profesor: Óscar Alejandro Esquivel Flores

INTRODUCTION

Se introducirán aplicaciones en las cuales la actividad de la computadora debe incluir la reacción apropiada ante la posibilidad de una gran variedad de mensajes que pueden ser mandados en momentos impredicibles, situaciones en las cuales ocurre un control de proceso, control de tráfico, control de pila, banco de aplicaciones, servicios de cómputo centralizados y finalmente todos los sitemas de información en los cuales un número de computadoras están acopladas entre sí.

Se presentarán una variedad de problemas, con su solución. Y en el momento de su discusión abordar conceptos relevantes.

ON THE NATURE OF SEQUENCIAL PROCESSES

Para entender lo que es un proceso secuencial se presenta la comparación de dos máquinas que ejecutan la misma tarea, una es no secuencial y la otra sí es secuencial.

Supongamos que se tienen las cantidades siguientes:

"a[1] = 7, a[2] = 12, a[3] = 2, a[4] = 9"

si quisieramos conocer el valor más grande, la salida sería **a[2]**. Sin embargo la salida deseada se volvería incompleta.

- Obs 1. Si la solicitud requerida fuera devolver el valor máximo que ocurre entre los valores dados (7, 12, 2, 12), entonces la respuesta sería 12.
- Obs 2. La restricción “De los cuatro valores ninguno es igual dos a dos” está limitada por la palabra “igual”. Por lo que la diferencia estará en que dados dos valores cualesquiere serán largamente comparados con “el poder de resolución” de los comparadores.

Para empezar a resolver primero construimos nuestra máquina no-secuencial. En donde los dos valores a comparar cualesquiera se representan con corrientes.

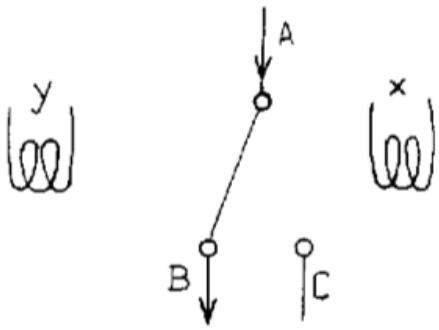


Fig.1. x < y

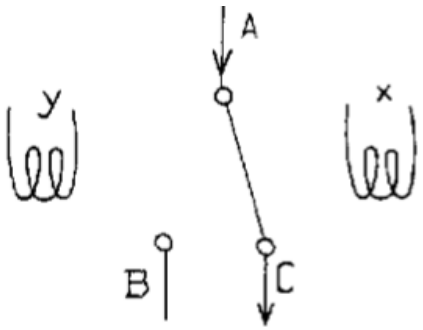
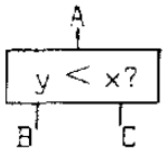


Fig.2. y < x

Cuando la corriente “y” es más grande que “x”, el electroimán izquierdo empujará más fuerte que el derecho que hará que el swith cambie a la izquierda. Si fuera de la otra manera el derecho empujará más fuerte hacia la derecha.

Una convención muy utilizada para representar las comparaciones es con una pequeña caja



en donde la entrada (A) está conectada con el lado derecho (C) cuando la respuesta a la pregunta o comparación es **sí**, y estará conectada a la izquierda si la respuesta es **no**.

Se construye la máquina comparadora como sigue

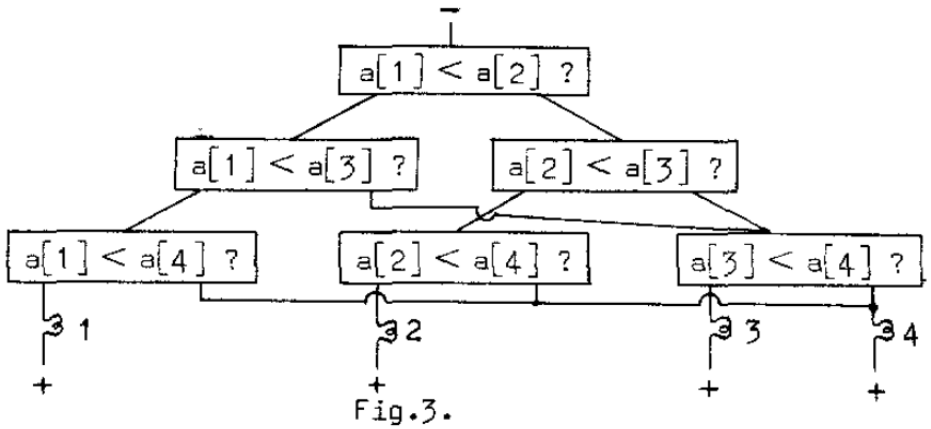


Fig.3.

En el uso de nuestra máquina, el progreso dle tiempo es únicamente reflejado en la obvia relación “before-after”, la cual nos dice que no podemos esperar una respuesta antes de que la pregunta haya sido propiamente puesta. Finalmente se dice que esta máquina es llamada “no-secuencial”.

Una segunda interpretación que se aborda es con un electron entrando al tope de la entrada preguntándose hacia dónde dirigirse. Se pregunta primero una vez dada la respuesta, este avanza en su camino.

En la primera representación de nuestra comparación todos los 6 comparadores empiezan a trabajar al mismo tiempo, apesar de que finalmente solo tres cambios de posición importan. En la segunda interpretación solo tres comparaciones son

evaluadas, sin emabrgo el precio de esta ganancia es que se tienen que ejecutar una detrás de la otra, la salida de la anterior decide qué preguntar después. La segunda interpretación se hace en **secuencia**.

Existe una téctina comúnmente aceptada para la escritura de algoritmos sin la necesitad de representar los procesos con imágenes y es ALGOL 60 (Algorithmic Language) y un ejemplo de ello aplicado al problema sería:

```
"       i:= 1; j:= 1;
 back: if j ≠ n then
         begin  j:= j + 1;
                if a[i] < a[j] then i:= j;
                goto back
         end" .
```

LOOSELY CONNECTED PROCESSES

Cuando dos o más procesos secuenciales tienen que cooperar entre ellos, deben de estar conectados, es decir, que deben poder comunicarse entre ellos con el fin de intercambiar información. Además se estipuló que los procesos deben estar sencillamente conectados; esto quiere decir que aparte de los momentos explícitos de intercomunicación, los procesos individuales por sí mismos deben ser considerados como completamente independientes entre ellos.

A Simple Example

Consideramos dos procesos secuenciales, “proceso 1” y “proceso 2” para los cuales para nuestros propósitos pueden ser considerados como cíclicos. En cada ciclo ocurre una llamada “sección crítica”. Para efectuar esta exclusión mutua, los dos procesadores tienen acceso a un número en común de variables. Se postula que inspeccionar el valor presente de dicha variable en común y asignarle un nuevo valor debe ser considerado como **indivisible**. Es decir cuando dos procesos asignan un nuevo valor a la misma variable “simultáneamente”, entonces las tareas deben ser consideradas como hechas una detrás de la otra, el valor final de la

variable será uno de los dos valores asignados, pero nunca una “mezcla” de los dos.

Se hace una extención a ALGOL 60, para que pueda aceptar la descripción de ejecución en paralelo:

- Cuando una secuencia de sentencias (separadas por ; como es usual) es rodeada por la sentencia especial “parbegin” y “parend” y se interpreta como ejecución en paralelo.

```
"begin  S1; parbegin  S2; S3; S4  parend; S5  end"
```

Dada la convención de arriba entonces describimos nuestra primera solución:

```
"begin  integer turn; turn:= 1;
        parbegin
        process 1: begin  L1: if  turn = 2  then  goto L1;
                              critical section 1;
                              turn:= 2;
                              remainder of cycle 1; goto L1
                   end;
        process 2: begin  L2: if  turn = 1  then  goto L2;
                              critical section 2;
                              turn:= 1;
                              remainder of cycle 2; goto L2
                   end
        parend
end"  .
```

No obstante esta solución, pensada como correcta, es, innecesariamente resctrictiva, pues después de la completud de la sección crítica 1, el valor de la variable “turn” se convierte en “2”, y debe ser 1 de nuevo, antes de la siguiente entrada en la sección crítica 1. Para poder expresar bien que esta no es la solución que queremos, impondremos la siguiente condición “Si uno de los procesos está bien parado fuera de su sección crítica, no se permite ir hacia un potencial bloqueo de otros procesos”. Lo cual hace que nuestra solución anterior sea inaceptable.

Se dan diversas propuestas de solución en el formato ALGOL 60, sin embargo todas son rechazadas debido a problemas que inciden en las secciones críticas. No obstante, se muestra una propuesa que por primera vez fue correcta y fue dada por el matemático holandés Th.J.Dekker.

THE GENERALIZE MUTUAL EXCLUSION PROBLEM

El problema de la sección anterior tiene una generalización natural: dados procesos *n*-cíclico, cada uno con una sección crítica, la pregunta a responder ahora es, ¿podemos construir los procesos de tal modo que el que en ningún momento a lo más uno esté comprometido con su sección crítica?

Nuestra solución debe de satisfacer los mismos requerimientos; el parar bien un proceso fuera de su sección crítica, no debería de ninguna manera, restringir la libertad de los otros procesos, y que si más de un proceso está por entrar a su sección crítica, debe ser imposible de imaginar para ellos tal velocidad finita, que su decisión el cual uno de ellos es el primero en entrar a su sección crítica, puede ser pospuesta hasta la eternidad.

Nuestra solución para N fijos es la siguiente:

```
"begin   integer  array b, c[0 : N];
         integer turn;
         for turn:= 0 step 1 until N do
               begin b[turn]:= 1; c[turn]:= 1 end;
         turn:= 0;
         parbegin
         process 1: begin.....................end;
         process 2: begin.....................end;
             .
             .
             .
             .
         process N: begin.....................end
         parend
 end"  .
```

Ejemplo de un proceso i:

```
"process i: begin  integer j;
              Ai: b[i]:= 0;
              Li: if  turn ≠ i  then
                  begin c[i]:= 1;
                        if b[turn] = 1 then turn:= i;
                        goto Li
                  end;
                  c[i]:= 0;
                  for j:= 1 step 1 until N do
                     begin  if j ≠ i and c[j] = 0 then  goto Li end;
                  critical section i;
                  turn:= 0; c[i]:= 1; b[i]:= 1;
                  remainder of cycle i; goto Ai
            end"   .
```

THE MUTUAL EXCLUSION PROBLEM REVISITED

Se retoma el problema de exclución mutua en tiempo de secciones críticas, introducidas en el capítulo anterior. Esta sección trata con una técnica más eficiente para resolver este problema

La Necesidad de una Solución Más Realista

La solución propuesta en la anterior sección resulta interesante en lo mucho que muestra que las fuentes de comunicación restringidas, desde un punto de vista teórico, son suficientes para resolver el problema. Sin embargo desde otro punto de vista, es inadecuado.

Nos permitiremos tomar un cierto periodo de tiempo durante el cual un proceso se encuantra en su sección crítica. Ningún otro proceso puede entrar a su sección crítica y que, si así lo quiere, deben esperar hasta que la ejecución de la actual sección crótoca haya sido completada. Podemos verlo como si los demás procesos “se fueran a dormir”.

En la actualidad, las computadoras tienen al menos dos caminos en los cuales la acción de esperar puede ser muy costosa. Divididas en dos partes “el procesador”, que es donde se realizan las operaciones lógicas y “la memoria” que no es procesada en el momento mismo, sino que la información se almacena para futuras referencias. En resumen el proceso computacional de la información es transportado desde el almacecnamiento hacia el procesador lo más rápido posible como se tenga un rol activo.

Tal computadora es una herramienta bastante flexible para la implementación de procesos secuenciales. Incluso una computadora con un único procesador puede ser usada para implementar un número de procesos secuenciales concurrentes.

Aparte del compartimiento de procesador, el compartimiento de memoria puede hacer la innecesaria la actividad de esperar un proceso no deseable. Asumamos que queremos que la inspección de una tarea a una “variable en común” implique el

acceso a la unidad de información. Acceder a una palabra *word* en un almacenamiento de core toma tiempo finito y para razones técnicas una palabra puede ser accesada a la sola vez. En caso de inminente coincidancia, el almacenamiento de solicitudes de acceso del distinto proceso están garantizados por una regla de prioridad: *The lower* acceso preferencial. El resultado es que inspección frecuente de variables en común pueden alentar el proceso.

The Synchronizing Primitives

El origen de los problemas recae en el hecho de que los accecesos indivisibles a variables en común son siempre del tipo “tráfico de información en una dirección”: Pues un proceso puede asignar un valor u obtener un valor actual. No obstante dicha obtención, no deja pasar a otros procesos y la consecuencia es que, cuando un proceso quiere reaccionar con respecto al valor actual de la variable en común, su valor puede ser cambiado por otros procesos entre el momendo de la obtención y las siguientes acciones efectuadas. Pero eso es inadecuado para resolver nuestro problema, veamos mejores alternativas adaptadas:

- semáforos (van entre las variables enteras de propósito especial en común)
- entre el repertorio de acciones, desde los cuales los procesos individuales son construidos, dos nuevos primitivos, “P-operation” y “V-operation”. La última operación siempre opera sobre un semáforo y representa el único camino en el cual el proceso actual puede acceder a los semáforos.

Los semáforos son esencialmente enteros no-negativos; cuando se usan solo para resolver el problema de exclusión mutua, el rango de sus valores será restringido a “0” y “1”. Cuando existe la necesidad de distinción, debemos hablar de “semáforos binarios” y “semáforos generales”.

- La “V-operation” es una operación de un único argumento, la cual debe ser el identificador de un semáforo. Su función es incrementar el valor de su argumento semáforo (Si S_1 es semáforo $V(S_1)$) en 1; esa operación es una op. indivisible.

- La “P-operation es una operación de un único argumento, el cual debe ser un identificador de un semáforo. Su función es decrementar el valor del semáforo pasado como argumento en 1 tan pronto como el valor resultante sea no-negativo; es una operación indivisible.

La “P-operation”, representa el retraso potencial. Cuando un proceso inicia una “P-operation” en un semáforo, el cual en ese momento su valor es 0, en este caso la P-operation no puede ser compledada hasta que otro proceso haya ejecutado una V-operation en el mismo semáforo y le haya dado un valor de “1”

The Synchronizing Primitives Applied to the Mutual Exclusion Problem

Por tanto la solución para N procesos, cada uno con una sección crítica es trivial. Puede ser realizado con la ayuda de un solo semáforo binario digamos de nombre “free”, su valor iguala el número de procesos que tienen permitido entrar a su sección crítica ahora:

```
"free = 1" means: none of the processes is engaged in its critical section
"free = 0" means: one of the processes is engaged in its critical section.
```

The General Semaphore

Typical Uses of the General Semaphore

Sean dos procesos, uno el “productor” y otro el “consumidor”. El productor es un proceso cíclico que cada vez que va sobre su ciclo, produce una cierta porción de información, que tiene que ser procesada por el consumidor. El “consumidor” es también un proceso cíclico que cada vez que va sobre su ciclo, procesa la siguiente porción de información que fue recibida del “producer”.

La relación $producer \rightarrow consumer$ implica un canal de comunicación de una sola dirección entre los procesos sobre la cual las porciones de información serán transmitidas.

En esta sección no nos pelearemos con ver las técnicas de implementación de buffer’s. Un buffer debe poder contener porciones sucesivas de información, debe por lo tanto ser un medio de almacenamiento, acceso a ambos procesos. Además, no solo debe contener las porciones de información, además debe representar su orden lineal. Un buffer también puede ser llamado “First-In-First-Out-Memory”

- Cuando el “productor” ya prepara su siguiente porción de información, se indica “add portion to buffer”
- Similarmente el “consumidor” ejecutará la acción “tomar porción del buffer”, donde se entiende que será la porción más vieja (almacenada en la cola), que aún está en el buffer.

```
"begin  integer  number of queuing portions;
        number of queuing portions:= 0;
        parbegin
        producer: begin
                again 1: produce the next portion;
                         add portion to buffer;
                         V(number of queuing portions);
                         goto again 1
                  end;
        consumer: begin
                again 2: P(number of queuing portions);
                         take portion from buffer;
                         process portion taken;
                         goto again 2
                  end
        parend
end"
```

En el caso de que la implementación del buffer sea por fuentes de encadenamiento, se requiere una nueva implementación para atacar este problema

```
"begin integer number of queuing portions, buffer manipulation;
      number of queuing portions:= 0;
      buffer manipulation:= 1;
      parbegin
      producer: begin
              again 1: produce next portion;
                       P(buffer manipulation);
                       add portion to buffer;
                       V(buffer manipulation);
                       V(number of queuing portions);
                       goto again 1
                end;
      consumer: begin
              again 2: P(number of queing portions);
                       P(buffer manipulation);
                       take portion from buffer;
                       V(buffer manipulation);
                       process portion taken;
                       goto again 2
                end
      parend
end"
```

The Superfluity of the General Semaphore

Se explicará y mostrará lo superfluo que es un semáforo general, sea el código

```
"begin  integer numqueupor, buffer manipulation, consumer delay;
        numqueupor:= 0; buffer manipulation:= 1; consumer delay:= 0;
        parbegin
        producer: begin
                again 1: produce next portion;
                         P(buffer manipulation);
                         add portion to buffer;
                         numqueupor:= numqueupor + 1;
                         if numqueupor = 1 then V(consumer delay);
                         V(buffer manipulation);
                         goto again 1
                 end;
        consumer: begin  integer oldnumqueupor;
                 wait:   P(consumer delay);
                 go on:  P(buffer manipulation);
                         take portion from buffer;
                         numqueupor:= numqueupor - 1;
                         oldnumqueupor:= numquepor;
                         V(buffer manipulation);
                         process portion taken;
                         if oldnumqueupor = 0 then  goto wait else  goto go on
                 end
       parend
end" .
```

En este programa la ocurrencia más relevante fue el periodo con un buffer vacío. Esta implementación es irrelevante, amenos que solo cuando el “consumer” pueda-quiera

tomar una siguiente porción, la cual está ausente. Se implementa una nueva versión:

```
"begin  integer numqueupor, buffer manipulation, consumer delay;
        numqueupor:= 0; buffer manipulation:= 1; consumer delay:= 0;
        parbegin
        producer: begin
                again 1: produce next portion;
                         P(buffer manipulation);
                         add portion to buffer;
                         numqueupor:= numqueupor + 1;
                         if numqueupor = 0 then
                            begin  V(buffer manipulation);
                                   V( consumer delay) end
                                        else
                            V(buffer manipulation);
                         goto again 1
                  end;
        consumer: begin
                again 2: P(buffer manipulation);
                         numqueupor:= numqueupor - 1;
                         if numqueupor = - 1 then
                            begin  V(buffer manipulation);
                                   P(consumer delay);
                                   P(buffer manipulation) end;
                         take portion from buffer;
                         V(buffer manipulation);
                         process portion taken;
                         goto again 2
                  end
        parend
end"
```

Este programa descrito es bien conocido como “El barbero durmiente”. Al final la exclución mutua se garantiza.

Finalmente ambos programas presentados anteriormente dan un fuerte hint a la conclución de que el semáforo general, es en efecto, superfluo (sin funcionalidad). Sin embargo no debemos intentar eliminarlo pues la restricción de sincronización de un lado expresada es muy común y la comparación de las soluciones con y sin el semáforo general muestran convinventemente que sebe ser considerado una herramienta adecuada.