Conceptos
SCD - Primer Parcial
Temas 1 y 2

Ismael Sallami Moreno

November 2024

Contents

1 Tema 1: Introducción a la Programación Concurrente

1.1 Axiomas de la Programción Concurrente

- Atomicidad y entrelazamientos de las ejecuciones atómicas.
- Consistencia de los datos tras un acceso concurrente
- Irrepetibilidad de las secuencias de instrucciones.
- Independencia de la velocidad de los procesos.
- Hipótesis de Progreso Finito.

1.2 Propiedad de los Sistemas Concurrentes

- Propiedad de seguridad (safety). Afirma un estado inalcanzable.
- Propiedad de vivacidad (liveness). En un momento se alcanzará un estado deseado.

1.3 Demostración de programas. Razonamientos.

- Razonamiento operacional.
- Razonamiento asertivo.

1.4 Axiomas

- Axioma de la sentencia nula. $\{P\}null\{P\}$
- Axioma de la asignación. $\{P_e^x\}$

1.5 Reglas

- **Regla de la consecuencia (1).**

$$\frac{\{P\}S\{Q\} \rightarrow \{R\}}{\{P\}S\{R\}} \tag{1}$$

  Es decir, siempre podemos hacer más débil la poscondición de un triple, de forma que este siga siendo cierto.

- **Regla de la consecuencia (2).**

$$\frac{\{R\} \rightarrow \{P\}, \{P\}S\{Q\}}{\{R\}S\{Q\}} \tag{2}$$

  Es decir, siempre podemos hacer más fuerte la precondición de un triple, manteniendo su veracidad.

- **Regla de la composición.**

$$\frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}} \tag{3}$$

Es decir, podemos condensar dos triples en uno, siempre y cuando la poscondición de uno sea la precondición del otro.

- **Regla de la conjunción.**

$$\frac{\{P_1\}S\{Q_1\}, \{P_2\}S\{Q_2\}}{\{P_1 \wedge P_2\}S\{Q_1 \wedge Q_2\}} \tag{4}$$

Es decir, si tenemos distintas pre y poscondiciones para una misma instrucción, podemos juntarlas todas en conjunción.

- **Regla del if.**

$$\frac{\{P \wedge B\}S_1\{Q\}, \{P \wedge \neg B\}S_2\{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2\{Q\}} \tag{5}$$

De esta forma, siempre que queramos probar que una tripleta de la forma

$$\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2\{Q\} \tag{6}$$

es cierta, tendremos que probar que las tripletas

$$\{P \wedge B\}S_1\{Q\} \quad \text{y} \quad \{P \wedge \neg B\}S_2\{Q\} \tag{7}$$

son ciertas.

- **Regla de la iteración.** Supongamos que una sentencia `while` puede iterar un número arbitrario de veces (incluso 0), tenemos que:

$$\{I \wedge B\}S\{I\} \tag{8}$$

$$\{I\} \text{ while } B \text{ do } S \text{ end } \{I \wedge \neg B\} \tag{9}$$

Donde a la proposición I la llamaremos invariante.

1.6 No Interferencia

$$NI(A, a) \equiv \{A \wedge pre(a)\}a\{A\}$$

1.7 Regla de la composición concurrente segura de procesos

Previamente debemos de comprobar que se cumple la regla de la NI.

$$\frac{\{P_i\}S_i\{Q_i\} \quad \text{son triples libres de interferencia} \quad 1 \leq i \leq n}{\{P_1 \wedge P_2 ... \wedge P_n\} \quad \text{cobegin} \quad (S_1 \parallel S_2 \parallel .... \parallel S_n) \quad \text{coend} \quad \{Q_1 \wedge Q_2 ... \wedge Q_n\}}$$

1.8 Verificación usando invariantes globales

- 1º Paso: Debemos de demostrar que el invariantes es cierto al inicio del programa.
- 2º Paso: Debemos de ir comprobando línea por línea en el código que el invariante se cumple antes y después de cada instrucción.

1.9 Demostrar que un programa acaba

- Demostrar que las variables adoptan valores diferentes a lo largo de la ejecución del programa.
- Identificar los valores como miembros del vector variante (vector que obtiene los valores posibles del programa), en caso contrario, notar que el programa no terminará.
- Si la salida esta en el vector variante, razonar que se alcanzará este estado.

2 Tema 2: Monitores y Exclusión Mutua

2.1 Dijkstra

2.1.1 Condiciones de Dijkstra

- No hacer suposiciones sobre el número de procesos soportados por el procesador.
- No hacer suposiciones sobre la velocidad de ejecución de los procesos.
- Si un proceso no esta en la sección crítica no puede impedir a otros procesos entrar en ella.
- Propiedad de alcanzabilidad: se garantiza que vaya a entrar un proceso de los que quieran entrar.

2.1.2 Algoritmo

```
var
    c : array[0..n-1] of (pasivo, solicitando, en_SC);
    turno : 0..n-1;

Process Pi();
begin
    while true do
    begin
        { Acceso a la secci n  cr tica }
        repeat
            c[i] := solicitando;
            { A }
            while turno <> i do
            begin
                if c[turno] = pasivo then
                    turno := i;
            end do
            c[i] := en_SC;
            { B }
            j := 0;
            while (j < n) and (j = i or c[j] <> en_SC) do
            begin
                j := j + 1;
```

```
24              end do
25          until j >= n
26          { Secci n  cr tica }
27          c[i] := pasivo;
28      end do
29  end
```

Listing 1: Algoritmo de exclusión mutua

2.2 Método de refinamiento sucesivo

- Primera etapa: entrada a la SC condicionada por una variable turno.
- Segunda etapa: Asociar a cada proceso una clave que lo defina.
  - Estado pasivo: no intenta acceder, representado con un 0.
  - ... con un 1.
- Tercera etapa: Cambiar el valor de la clave a 0 antes de que entre en la SC.
- Cuarta etapa: Comprueba que si el otro proceso también ha cambiado el valor de la clave a 0, este pasa a poner dicho valor a 1.

2.3 Algoritmo de Dekker

- Se puede considerar como una 5º etapa del método de refinamiento sucesivo.
- Mezcla entre las condiciones 1 y 4 del algoritmo de Dijkstra.
- Propiedades de corrección:
  - EM.
  - Alcanzabilidad en la SC.
  - Vivacidad: se garantiza usando Dekker, pero puede causar inanición.
  - Equidad: depende del hardware.

2.4 Algoritmo de Knuth

Descripción

El algoritmo de Knuth utiliza un arreglo de dos enteros y un indicador global para gestionar la entrada a la sección crítica. La clave es que cada proceso debe verificar su turno antes de ingresar.

Código

```
var
    turno : 0..1;                    { Indica el turno }
    desea : array[0..1] of boolean; { Deseo de entrar de cada
       proceso }

Process Pi (i : 0..1);
begin
    while true do
    begin
        desea[i] := true;            { El proceso i desea entrar }
        while turno <> i do
        begin
            if not desea[1-i] then  { Verifica si el otro
               proceso no desea entrar }
                turno := i;          { Cambia el turno a i }
        end do;
        { Secci n cr tica }
        desea[i] := false;           { Sale de la secci n
           cr tica }
    end do;
end;
```

Listing 2: Algoritmo de Knuth

2.5 Algoritmo de Peterson

Descripción

El algoritmo de Peterson es simple y elegante, combinando un indicador de turno y un arreglo de booleanos para garantizar la exclusión mutua y evitar interbloqueos.

Código

```
var
    turno : 0..1;                    { Indica el turno }
    desea : array[0..1] of boolean; { Deseo de entrar de cada
       proceso }

Process Pi (i : 0..1);
begin
    while true do
    begin
        desea[i] := true;            { El proceso i desea entrar }
        turno := 1 - i;              { Cede el turno al otro
           proceso }
        while desea[1-i] and (turno = 1 - i) do
            skip;                    { Espera activa si el otro
               desea entrar }
        { Secci n cr tica }
```

```
14          desea[i] := false;          { Sale de la secci n
               cr tica }
15      end do;
16  end;
```

Listing 3: Algoritmo de Peterson

2.6 Anotaciones Generales

- Dekker: para 2 procesos, pero no garantiza que no haya inanición.
- Dijkstra es para n procesos, pero no garantiza que no haya inanición.
- Knuth: Si garantiza que no haya inanición, pero es de orden de 2^n.
- Peterson: similar al anterior, pero de orde de n^2.