TEMA-1.pdf

Blancabril

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

TEMA 1

1. Conceptos básicos

→ PROGRAMA SECUENCIAL: código fuente ejecutado en una sentencia.

→ PROGRAMA CONCURRENTE: conjunto de programas secuenciales que se pueden ejecutar a la vez.

→ PROCESO: ejecución programa secuencial
  → Físico: varias CPU's y cada una ejecuta un programa.
  → Lógico: varios programas, va dedicando poco tiempo a cada uno de ellos y va cambiando.

→ CONCURRENCIA: potencial y técnicas de programación usadas para simplificar y reparar errores.

→ Programación concurrente: conjunto de notaciones y técnicas para simplificar y reparar errores. Es independiente del paralelismo.

2. Programación Paralela, Distribuida y Tiempo Real

→ Programación paralela: acelera la resolución mediante aprovechar capacidad de procesamiento paralelo.

→ Programación distribuida: varios componentes software localizados en diferentes ordenadores trabajen juntos.

→ Programación de tiempo real: sistemas que funcionan continuamente recibiendo y enviando desde componentes hardware que trabajan con restricciones estrictas.

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

3. Beneficios Programación Concurrente:

→ Mejora Pa eficacia : aprovechar mejor recursos software.

→ 1 solo procesador: cuando la tarea tiene el control del procesador → E/S y cede el controlador a otra. Además permite que varios usuarios usen el sistema de forma interactiva.

→ Varios procesadores: se reparten las tareas entre los procesadores → ↓ tiempo de ejecución.

→ Mejoras de calidad: varios procesos secuenciales ejecutandose como un único programa secuencial.

4. Modelo Abstracto y Consideraciones sobre el Hardware

- Programación Concurrente

- Dependen de arquitectura.
- Consideran una máquina virtual → base común para modelar la ejecución de los procesos concurrentes.
- El tipo de paralelismo afecta a la eficacia de la ejecución pero no a la corrección de los programas.

- Multiprogramación: el SO gestiona como los múltiples procesos se reparten ciclos de CPU.

  - Mejor aprovechamiento CPU
  - Servicio interactivo varios usuarios
  - Sincronización y comunicación → variables compartidas

- Sistemas Multiprocesador Memoria Compartida:
  - Pueden compartir física (o no) memoria, pero comparten espacio de direcciones.
  - Variables compartidas alojadas en direcciones del espacio compartido.

- Sistemas Distribuidos:
  - No existe memoria compartida.
  - Procesadores interaccionan ⟶ red de interconexión.
  - Programación distribuida: tratamiento fallos, transparencia, ...

5. Sentencias Atómicas y No Atómicas

- Atómica: ejecución / instrucción que se ejecuta de principio a fin sin que le afecten otras sentencias en ejecución.

  → No se ve afectada si no depende de la forma de ejecución de las otras instrucciones.

  → El funcionamiento de una instrucción es el efecto en el estado de ejecución del programa justo cuando acaba.

  → Estado ejecución → valores de variables y registros de los procesos.

Ejemplo

El resultado no depende de otras instrucciones.

Si r=v, si se ejecuta la instrucción de escritura r en x, sabemos que x=v (incluso aunque haya procesos accediendo).

↳ load x → add r,a → store x

No atómicas: Ejemplo

$x = x + 1$

1. Lee valor x y carga registro
2. +1 el valor almacenado en r.
3. Escribir valor de r en x.

→ No se puede predecir el resultado final.

6. Abstracción

Es un modelo basado en el estudio de de todas las posibles secuencias de ejecución entrelazadas.

→ Solo se tienen en cuenta las características relevantes → resultado final

→ Simplificar análisis y estudio de los programas.

Se ignora:

→ Estado memoria cada proceso.

→ Registros que acceden a cada proceso.

→ Política concreta planificación de procesos.

7. Independencia del entorno de ejecución

Una instrucción individual sobre un dato no depende de la ejecución.

Ejemplo

Un programa tiene dos instrucciones I_0 e I_1 → concurrente ($I_0 = 1$; $I_1 = 2$)

- Si no acceden a la misma celda de memoria → orden ejecución no afecta.
- Si acceden a la misma, o N = 2 o N = 1 pero no se suman.

8. Velocidad de Ejecución. Hipótesis del Progreso Finito

No se puede hacer una suposición de la velocidad de un proceso. Solo sabemos que es positivo (+0).

Un programa concurrente → acciones y componentes → no tener en cuenta el entorno de ejecución.

Si la velocidad de ejecución $\neq 0$:

→ Punto de vista global: al menos existirá un proceso preparado.

→ Punto de vista local: cuándo un proceso comienza la ejecución de una sentencia, la completará en un intezualo finito.

9. Estados e Historias de Ejecución de un Programa Concurrente

- Estado de un programa concurrente: valozes de las vaziables del programa en un momento dado (vaziables explícitamente y vaziables ocultas).

  Comienza su ejecución en estado inicial y los pzocesos van modifi-cando el estado cuándo se ejecutan sus sentencias atómicas.

Historia: secuencia de estados: $s_0, s_1, \ldots s_n$.

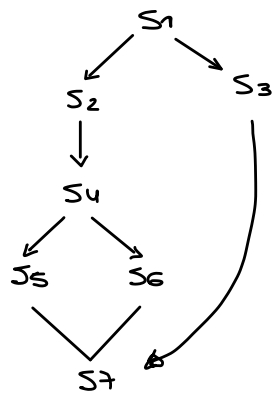
10. Notación Expresar Ejecución Constante

- Sistemas Estáticos:
  - El número de pzocesos está fijado en la fuente del programa.
  - Los pzocesos se activan al lanzar el programa.
- Sistemas Dinámicos
  - Número hebras que se pueden activar.

11. Grafo de Sincconitación

Es un Grafo Dirigido Cíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa.

Tenemos 2 actividades S_1 y S_2, una flecha desde S_1 hacia S_2 significa que S_2 no puede comenzar su ejecución hasta que S_1 haya finalizado.



→ S7 no se ejecuta hasta que terminen S2 - S4 - S5 - S6 y S3.

- Definición Estática de Procesos

```
var ...   ← variables compartidas
process Uno;
var ...   ← variables locales
begin
  ...     } código
end;
process Dos;
  ⋮
```

- El programa acaba cuándo se acaban todos los procesos.

- Definición Estática de Vectores de Procesos

```
var ...   ← variables compartidas
process NomP [ind: a..b];
var ...   ← variables locales
 begin
    ...
  // ind vale a, a+1, ... b)
end;
```

- Creación Procesos No Estructurada Fork - Join

→ Fork: es una sentencia que especifica que la rutina nombrada puede comenzar su ejecución, al mismo tiempo que comienza la sentencia siguiente.

→ Join: es una sentencia que espera la terminación de la rutina nombrada, antes de comenzar la sentencia siguiente.

```
procedure P1;
begin
    A;
    fork P2;        ──→  procedure P2;
    B;                   begin
                             D;
    join P2;        ←──  end
    C;
end
```

→ Mientras se ejecuta P2 se hace B pero hasta que no termina no empieza C.

→ Ventajas: práctica y potente

→ Desventajas: no estructurado → difícil comprensión.

- Creación de Procesos Estructurada → Cobegin - Coend

Comienzan todas las sentencias a la vez:

→ Coend: espera a que se terminen todas las sentencias.

→ Hace explícita rutinas se ejecutan de forma concurrente.

```
begin
    A;
    cobegin
        B; C; D;
    coend;
    E;
end
```

→ Comienza A, despues B, C y D a la vez pero hasta que estas no terminen, no empieza E.

→ Ventajas: más fáciles de entender.

→ Inconveniente: menor potencia que fork - join.

12. Exclusión Mutua y Sincronización

- Condiciones de Sincronización: restricción en el orden en el que pueden entremezclar las instrucciones que generan los procesos de un programa.

12.1 Exclusión mutua

Son secuencias comunes de instrucciones consecutivas que aparecen en varios procesos de un programa concurrente.

→ Sección crítica: Conjunto de secuencias de instrucciones.

→ Exclusión mutua: en cada instante, hay un/ningún proceso ejecutando alguna parte de la sección crítica y solo funciona correctamente si se hace así.

La EM ocurre en procesos en los que la memoria esta compartida, entonces los procesos acceden a leer/modificar variables.

13. Traducción y Ejecución de Asignaciones

Ejemplo: $x = x + 1$

1) load $r_i \leftarrow x$ Cargamos el valor x en una variable.
2) add $r_i, 1$ Añadimos una unidad.
3) store $r_i \rightarrow x$ Guardar el valor en la posición x de memoria.

- Cada proceso tiene su registro.
- Comparten variable x, por lo que se pueden pisar.

13.1 Instrucciones Compuestas

- No atómicas: x = {-1, 0, 1}

```
begin
   x := 0;
   cobegin
      x := x + 1;
      x := x - 1;
   coend
end
```

- Atómicas: x = {0, 1}

```
begin
   x := 0;
   cobegin
      < x := x+1 >;
      < x := x-1 >;
   coend
end
```

14. Condición de sincronización

En un programa concurrente, una condición de sincronización establece que todas las posibles secuencias atómicas son correctas.

→ Suele ocurrir cuándo en el programa, uno o varios procesos deben esperar a que se cumpla una condición global.

Ejemplo:

Hay una variable compartida x y unos van creando y otros consumiendo.

```
var x;  // vari. comp.
process Productor;
   var a: integer; // variable local
begin
   while true do begin
      a := ProducirValor(); // crea valor
      x := a; // guarda memoria
   end
end
```

```
process Consumidor;
   var b: integer; // variable loca
begin
   while true do while
      b := x; // guardamos valor pos
              memoria
   UsarValor(b);
   end
end
```

15. Trazas Incorrectas de un Programa Concurrente

Los procesos para que funcionen correctamente se caracterizan como lectura (L) y escritura (E).

- Para que sea correcta, no se lee hasta que se escribe de nuevo en x y no se escribe hasta que el valor no ha sido leído.

16. Concepto de Corrección de un Programa Concurrente

- Propiedad de un programa concurrente: atributo programa que es cierto para todas las posibles trazas del programa.

  - Seguridad: son condiciones que deben cumplirse en cada instante.
    - Requeridas en especificaciones estéticas del programa
    - Fáciles de demostrar.
  - Vivacidad: propiedades que se deben cumplir en algún momento del futuro.
    - Propiedades dinámicas y más difícil de demostrar.

17. Enfoque operacional

Se trata de un análisis exhaustivo de casos y se comprueba la corrección de todas las posibles trazas

Problema: limita su utilidad cuando hay programas concurrentes complejos ya que se multiplica exponencialmente.

18. Enfoque Axiomático

Es un sistema lógico que permite establecer propiedades a base de axiomas y reglas de inferencia.

- Las sentencias atómicas actúan como transformadores de aserto.

$\{P\}\ S\ \{Q\}$

Si "S" comienza a ejecutarse en algún momento en el que el aserto P (precondición) es true, entonces el aserto Q (poscondición) será true

- Menor complejidad: el trabajo que conlleva la prueba de corrección es proporcional al nº de sentencias atómicas.

19. Triples de Hoare

- C es programa del lenguaje cuyos programas están siendo especificados.
- P, Q son aserto definidos con las variables del programa C.

$\{P\}\ C\ \{Q\}$

- Satisface a C si satisface a P y si C termina de ejecutarse, satisface a Q.

Ejemplo: $\{x==1\}\ x=x+1\ \{x==2\}$

El aserto de P dice que x es 1, C vale 2, y como el aserto Q dice que x es 2, eso es cierto.

$\{x==1\}\ x=x+1\ \{x==2\}$ ✓

$\{x==1\}\ x=x+1\ \{x==3\}$ ✗

$\{x==1\}$ WHILE T DO NULL $\{Y==3\}$ ✓

Leyes distributivas: $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$

$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$

Leyes De Morgan: $\neg(P \wedge Q) = \neg P \vee \neg Q$

$\neg(P \vee Q) = \neg P \wedge \neg Q$

And: $(P \wedge Q) \rightarrow P$

Or: $P \rightarrow (P \vee Q)$