Tema 2: Procesos y sincronización

Elvira Albert

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

elvira@sip.ucm.es

Madrid, Febrero, 2021

Introducción

- Los programas concurrentes dependen de componentes compartidos:
  - Variables compartidas (read/write)
  - Canales de comunicación compartidos (send/receive)
- La comunicación trae la necesidad de sincronización:
  - exclusión mutua: turno en acceso a datos compartidos
  - sincronización condicional: espera hasta que quede libre
- PARTE 1: programas concurrentes con sincronización usando memoria compartida
- Ejecutan en máquinas de memoria compartida, o distribuidas y con memoria compartida

Contenidos Tema 2

2.1 Conceptos básicos

- Estado: valores de las variables del programa
- Acciones atómicas: se ejecutan de manera indivisible
- Historia: entrelazado de instrucciones atómicas
- Ejecución: cada ejecución produce una historia
- Indeterminismo: debido al entrelazado el número de historias es enorme
- Sincronización: reducir el número de historias que se pueden producir
  - exclusión mutua: convertir secuencias de acciones en secciones críticas (como si fueran atómicas)
  - sincronización condicional: retrasar la ejecución
- Propiedades que se cumplan para todas las trazas:
  - safety: el programa nunca entra en mal estado
  - liveness: eventualmente entra en un estado bueno

- Ejemplos de propiedades:
  - corrección parcial (safety): asumiendo que termina el resultado es correcto
  - exclusión mutua (safety): varios procesos no ejecutan secciones críticas a la vez
  - terminación (liveness): la longitud de todas las trazas es finita
  - eventualmente se entra en la sección crítica (liveness)
  - corrección total $\equiv$ corrección parcial $+$ terminación
- Verificación de sistemas concurrentes:
  - testing/debugging
  - razonamiento operacional
  - análisis abstracto
    - aserciones: fórmulas que caracterizan el conjunto de estados
    - transformadores: definen como las acciones atómicas cambian el estado
    - trabajo del análisis proporcional al número de instrucciones atómicas

2.2 Paralelización

- Objetivo: encontrar formas de paralelizar
- Independencia de procesos
- Problema: buscar instancias de patrón en un fichero
  - solución secuencial
  - usando dos variables locales
  - patrón productor-consumidor (buffer compartido)

2.3 Sincronización

- Búsqueda del elemento máximo de un array
- Problema que requiere sincronización entre procesos
  - solución secuencial
  - examinar cada elemento en paralelo
  - convertir acciones separadas en atómicas
  - double check

2.4 Acciones atómicas e instrucciones AWAIT

- Atomicidad de grano fino: implementada directamente por el hardware
- La ejecución de las instrucciones que operan sobre variables locales parece atómica
- Propiedad "at-most-once":
  - referencia crítica: es una referencia que es modificada por otro proceso
  - la expressión x=e cumple la propiedad at-most-once si:
    1. e contiene como máximo una referencia crítica y x no es leída por otros procesos
    2. e no contiene ninguna referencia crítica y x puede ser leída por otros procesos
  - la ejecución de x=e parece atómica si cumple la propiedad at-most-once

2.4 Acciones atómicas e instrucciones AWAIT

- $< e >$: Cuando una instrucción no cumple la propiedad at-most-once tenemos que ejecutarla de manera atómica
- Si se cumple la propiedad at-most-once $< e > \equiv e$
- e puede contener varias instrucciones
- exclusión mutua + sincronización condicional: <await (B) S;>
- ESPERA ACTIVA: sólo delays <await (B);>
- sólo exclusión $< S; >$
- si B cumple la propiedad at-most-once, <await (B);>$\equiv$ while (not B);
- usar la instrucción await para modelar problemas productor-consumidor

2.5 Semántica de los programas concurrentes

Lógica de programación (PL)

- sistema lógico formal que permite establecer y demostrar propiedades de los programas
- PL contiene símbolos, fórmulas, axiomas y reglas de inferencia
- Fórmulas: $\{P\}\ S\ \{Q\}$

  si la ejecución empieza en un estado que satisface P, la ejecución de S termina en un estado que satisface Q

Reglas de inferencia instrucciones secuenciales:

Composición $\dfrac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$

Condicional $\dfrac{\{P \wedge B\}S\{Q\} \quad (P \wedge \neg B) \implies Q}{\{P\}if\,(B)S\{Q\}}$

Iteración $\dfrac{\{I \wedge B\}S\{I\}}{\{I\}while\ (B)\ S; \{I \wedge \neg B\}}$

Consecuencia $\dfrac{P' \implies P \quad \{P\}S\{Q\} \quad Q \implies Q'}{\{P'\}S; \{Q'\}}$

Reglas de inferencia instrucciones concurrentes:

Await $\dfrac{\{P \wedge B\} S \{Q\}}{\{P\} < await(B) S; > \{Q\}}$

Instrucción co

$$\frac{\{P_i\} S_i \{Q_i\} \text{son libres de interferencia}}{\{P_1 \wedge \ldots \wedge P_n\} co\ S_1; //S_2 \ldots //S_n\ oc \{Q_1 \wedge \ldots \wedge Q_n\}}$$

Un proceso interfiere con otro si ejecuta una asignación que invalida la aserción de otro proceso, es decir, si un proceso asigna a una variable compartida e invalida la hipótesis de un proceso, la prueba no es válida

Propiedades de seguridad:

- Ejemplos de propiedades de seguridad
  - Exclusión mutua
  - Deadlock
- Garantía de seguridad:
  - Especificar BAD: estado malo
  - El programa es seguro si BAD no se cumple en ningún estado
- Formas de mostrarlo:
  - Recorrer todas las trazas y estados...
  - Asegurar que $\neg$BAD es un invariante global
  - Exclusión de configuraciones

2.6 Propiedades de seguridad y viveza

Propiedades de viveza (liveness):

- La mayoría de propiedades de viveza dependen de la noción de "justicia": todos los procesos tienen oportunidad de ejecutar
- Política de planificación: cuando hay varios procesos con acciones atómicas que se podrían ejecutar, determina cúal

Grados de "justicia":

- Incondicional: toda acción atómica incondicional será eventualmente ejecutada
  round-robin y scheduling paralelo son indicionalmente justas
- Débil: es incondicionalmente justa y toda acción atómica condicional <await (B) S;> que es elegible se ejecuta eventualmente asumiendo que B se hace true y permanece true hasta que se comprueba
  round robin y time slicing son débilmente justas
- Fuerte: es incondicionalmente justa y toda acción atómica condicional <await (B) S;> asumiendo que la condición B se pone a cierto un número infinito de veces

Consideraciones finales:

- Imposible que sea práctica y fuertemente justa
- Alternar acciones entre los dos procesos es fuertemente justa
- Round robin y time slicing son prácticas pero no son fuertemente justas
- Ejecución en paralelo tampoco es fuertemente justa

Livelock:

- Livelock: es una forma de starvation (inanición o postergación indefinida) en la que los bucles de espera iteran infinitamente, es decir, el programa esta vivo pero los bucles no van a ninguna parte

```
string line;
read a line of input from stdin into line;
while (!EOF) {    # EOF is end of file
   look for pattern in line;
   if (pattern is in line)
      write line;
   read next line of input;
}
```

Finding Patterns: Sequential Program

```
string line1, line2;
read a line of input from stdin into line1;
while (!EOF) {
  co look for pattern in line1;
     if (pattern is in line1)
        write line1;
  // read next line of input into line2;
  oc;
}
```

Finding Patterns: Disjoint Processes

```
string buffer;  # contains one line of input
bool done = false;  # used to signal termination
co  # process 1:  find patterns
  string line1;
  while (true) {
    wait for buffer to be full or done to be true;
    if (done) break;
    line1 = buffer;
    signal that buffer is empty;
    look for pattern in line1;
    if (pattern is in line1)
      write line1;
  }
//  # process 2:  read new lines
  string line2;
  while (true) {
    read next line of input into line2;
    if (EOF) {done = true; break; }
    wait for buffer to be empty;
    buffer = line2;
    signal that buffer is full;
  }
oc;
```

Figure 2.1 Finding patterns in a file.

```
int buf, p = 0, c = 0;
process Producer {
  int a[n];
  while (p < n) {
    ⟨await (p == c);⟩
    buf = a[p];
    p = p+1;
  }
}
process Consumer {
  int b[n];
  while (c < n) {
    ⟨await (p > c);⟩
    b[c] = buf;
    c = c+1;
  }
}
```

Figure 2.2 Copying an array from a producer to a consumer.

```
int buf, p = 0, c = 0;
{PC: c <= p <= c+1 ∧ a[0:n-1] == A[0:n-1] ∧
          (p == c+1) ⇒ (buf == A[p-1])}

process Producer {
  int a[n];    # assume a[i] is initialized to A[i]
  {IP: PC ∧ p <= n}
  while (p < n) {
    {PC ∧ p < n}
    ⟨await (p == c);⟩   # delay until buffer empty
    {PC ∧ p < n ∧ p == c}
    buf = a[p];
    {PC ∧ p < n ∧ p == c ∧ buf == A[p]}
    p = p+1;
    {IP}
  }
  {PC ∧ p == n}
}

process Consumer {
  int b[n];
  {IC: PC ∧ c <= n ∧ b[0:c-1] == A[0:c-1]}
  while (c < n) {
    {IC ∧ c < n}
    ⟨await (p > c);⟩    # delay until buffer full
    {IC ∧ c < n ∧ p > c}
    b[c] = buf;
    {IC ∧ c < n ∧ p > c ∧ b[c] == A[c]}
    c = c+1;
    {IC}
  }
  {IC ∧ c == n}
}
```

Figure 2.4 Proof outline for the array copy program.