Programación concurrente

Master de Computación

I Conceptos y recursos para la programación concurrente:

I.6 Sincronización basada en memoria compartida.

J.M. Drake

M. Aldea

Procesos concurrentes y memoria compartida.

- Si los diferentes procesos de un programa concurrente tienen acceso a **variables globales** o **secciones de memoria compartidas**, la transferencia de datos a través de ella es una vía habitual de comunicación y sincronización entre ellos.
- Las primitivas para programación concurrente basada en memoria compartida resuelven los problemas de sincronización entre procesos y de exclusión mutua utilizando la **semántica de acceso a memoria compartida**.
- En esta familias de primitivas, la semántica de las sentencias hace referencia a la exclusión mutua, y la implementación de la sincronización entre procesos se hace de forma indirecta.

Mecanismos basados en memoria compartida

- **Semáforos:** Son componentes pasivos de bajo nivel de abstracción que sirven para arbitrar el acceso a un recurso compartido.

- **Secciones críticas**: Son mecanismos de nivel medio de abstracción orientados a su utilización en el contexto de un lenguaje de programación y que permiten la ejecución de un bloque de sentencias de forma segura.

- **Monitores**: Son módulos de alto nivel de abstracción que sirven para arbitrar el acceso a un recurso compartido.

Definición de semáforo.

- Un semáforo es un **tipo de datos**.

- Como cualquier tipo de datos, queda definido por:
  - Conjunto de **valores** que se le pueden asignar.
  - Conjunto de **operaciones** que se le pueden aplicar.

- Un semáforo tiene asociada una **lista de procesos**, en la que se incluyen los procesos suspendidos a la espera de su cambio de estado.

Valores de un semáforo.

- En función del rango de valores que puede tomar, los semáforos se clasifican en:
  - Semáforos **binarios**: Pueden tomar solo los valores 0 y 1.

    var mutex: **BinSemaphore**;
  - Semáforos **contadores**: Puede tomar cualquier valor Natural (entero no negativo).

    var escribiendo: **Semaphore**;

- Un semáforo con el valor **0** representa un **semáforo cerrado**, y con un valor **mayor que cero** representa un **semáforo abierto**.

- Mas adelante demostraremos que un semáforo contador se puede implementar utilizando semáforos Binarios.

- Los sistemas suelen ofrecer como componente primitivo semáforos contadores, y su uso, lo convierte de hecho en semáforo binario.

Operaciones seguras de un semáforo.

- Un semáforo

  var p: **semaphore**;

  admite dos **operaciones seguras**:

  - **wait**(p): Si el semáforo no es nulo (abierto) decrementa en uno el valor del semáforo. Si el valor del semáforo es nulo (cerrado), el thread que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo.

  - **signal**(p): Si hay algún proceso en la lista de procesos del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al wait que lo suspendió. Si no hay procesos en espera en la lista incrementa en 1 el valor del semáforo.

Operación no segura de un semáforo.

- Un semáforo

  var p: **semaphore**;

  admite una **operación no segura**:

  - **initial(p, Valor_inicial)**: Asigna al semáforo p el valor inicial que se pasa como argumento.

- Esta operación es no segura y por tanto debe ser ejecutada en una fase del programa en la que se tenga asegurada que se ejecuta sin posibilidad de concurrencia con otra operación sobre el mismo semáforo.

Operación wait.

- Pseudocódigo de la operación: **wait**(p);

  if p>0

  then p:= p-1;

  else Suspende el proceso y lo encola en la lista del semáforo.

- Está **protegida contra expulsión** entre el chequeo del valor del semáforo y la asignación del nuevo valor o la suspensión.

- El nombre de la operación **wait es equívoco**. En contra de su significado semántico natural, su ejecución a veces provoca una suspensión pero en otros caso no implica ninguna suspensión.

Operación signal.

- Pseudocódigo de la operación: **signal**(p);

  if Hay algún proceso en la lista del semáforo

  then Activa uno de ellos

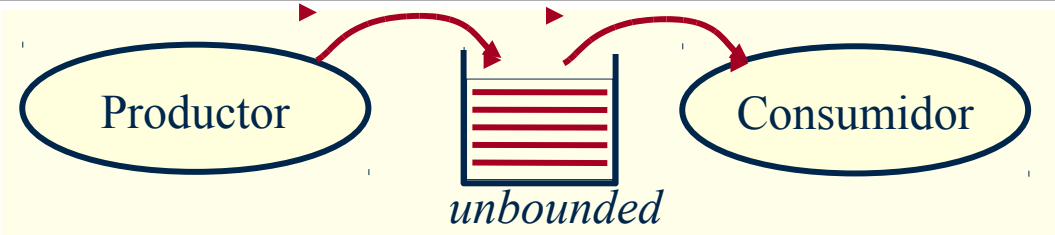
  else p:= p+1;

- Está **protegida contra expulsión** entre la comprobación de si hay proceso bloqueado y la activación o asignación.
- La ejecución de la operación signal(p) nunca provoca una suspensión del thread que lo ejecuta.
- Si hay varios procesos en la lista del semáforo, la operación signal solo activa uno de ellos
  - se elige de acuerdo con un criterio propio de la implementación (FIFO, LIFO, Prioridad, etc.).

Ejemplo: Exclusión mutua.

```
program Exclusion_Mutua;
        var mutex: binsemaphore;
        process type Proceso;
        begin
          repeat
                  wait(mutex);
                     (* Código de la sección crítica *)
                  signal(mutex);
          forever;
        end;
var p, q, r: Proceso;
begin
    initial(mutex,1);
    cobegin p; q; r; coend;
end;
```

Ejemplo: Sincronización productor-consumidor (mal)




```
var datoDisponible: Semaphore:= 0;
```

```
process Productor;
var dato: Tipo_Dato;
begin
  repeat
    produceDato(var dato);
    dejaDato(dato);
    signal(datoDisponible);
  forever;
end:
```

```
process Consumidor;
var dato: Tipo_Dato;
begin
  repeat
      wait(datoDisponible);
      tomaDato(var dato);
      consume(dato);
  forever;
end;
```

Ejemplo: Sincronización productor-consumidor (bien)

```
datoDisponible: Semaphore:=0;
mutex: BinSemaphore:=1;

process type Productor;
  var dato:Tipo_Dato;
begin
  repeat
      dato:=Produce_Dato;
      wait(mutex);
      dejaDato(dato);
      signal(mutex);
      signal(datoDisponible);
  forever;
end;
```

```
process type Consumidor;
  var dato:Tipo_Dato;
begin
  repeat
    wait(datoDisponible);
    wait(mutex);
    dato:=tomaDato;
    signal(mutex);
    consume(dato);
  forever;
end;
```

Productor consumidor con buffer limitado (1).

```
program Productor_Consumidor;
const LONG_BUFFER = 5;
type Tipo_Dat= ....;
var datoDisponible:Semaphore;
      hueco:Semaphore;
      mutex: BinSemaphore;
      buffer: record datos:array[0..LONG_BUFFER-1]of  Tipo_Dato;
                     nextIn, nextOut: Natural:=0; end;
```

```
procedure dejaDato(d:Tipo_Dat);
begin
  buffer.dato[buffer.nextIn]:=D;
  buffer.nextIn:=(buffer.nextIn+1)
        mod LONG_BUFFER;
end;
```

```
procedure tomaDato(d:Tipo_Dat);
begin
  d:=buffer.dato[buffer.nextOut];
  buffer.nextOut:= (buffer.nextOut+1)
        mod LONG_BUFFER;
end;
```

Productor consumidor con buffer limitado (2).

```
process productor;
var dato:Tipo_Dat;
 begin
    repeat
       dato:=produceDato;
       wait(hueco);
       wait(mutex);
       dejaDato(dato);
       signal(mutex);
       signal(datoDisponible);
    forever;
end;
```

```
process consumidor;
var dato: Tipo_Dat;
begin
    repeat
        wait(datoDiponible);
        wait(mutex);
        tomaDato(dato);
        signal(mutex);
        signal(hueco);
        Consume(dato);
    forever;
end;
```

```
begin
    initial(mutex,1); initial(hueco, LONG_BUFFER); initial(datoDisponible,0);
    cobegin productor; consumidor; coend;
end.
```

Implementación de semáforos contadores con binarios

```
type SemCont = record
            mutex, espera: BinSemaphore;
            cuenta: Natural;
end;
procedure initialCont(var s:SemCont, v:Natural);
begin initial(s.mutex,1); v.cuenta:= v;
         if (v=0) then initial(s.espera,0) else initial(s.espera,1);
end;
```

```
procedure waitCont(var s:SemGral);
begin
   wait(s.espera);
   wait(s.mutex);
   s.cuenta:= s.cuenta – 1;
   if (s.cuenta>0) then signal(s.espera);
   signal(s.mutex);
end;
```

```
procedure signalCont(var s:SemCont);
begin
   wait(s.mutex)
   s.cuenta:= s.cuenta +1;
   if (s.cuenta=1) then signal(s.espera);
   signal(s.mutex);
end;
```

Cena de filósofos chinos (1)

```
program Cena_filósofos_chinos;
const    N = 5;
var  palillo: array [1..N] of  BinSemaphore;
       sillasLibres: Semaphore;

process type TipoFilosofo(nombre: Integer); begin ... end;
var filosofo: array [1..N] of TipoFilosofo;
       i: Integer;

begin
   for i:=1 to N do initial(palillo[I],1);
   initial(sillasLibres,N-1);
   cobegin for i:=1 to N do filosofo[I](I); coend;
end.
```

Cena de filósofos chinos (2)

```
process type TipoFilosofo(nombre: Natural);
   derecho=nombre;
   izquierdo=nombre mod (N+1);
begin
    repeat
        sleep(Random(5));                -- Está pensando
        wait(sillaslibres);
        wait(palillo[derecho]);
        wait(palillo[izquierdo]);
        sleep(Random(5));                -- Está comiendo
        signal(palillo[derecho]);
        signal(palillo[izquierdo]);
        signal(pillasLibres);
     forever
end;
```

Implementación de los semáforos.

- La clave para implementar semáforos es disponer de un mecanismo(lock y unlock) que permita garantizar las secciones críticas de las primitivas wait y signal.

Wait

```
lock
  if s>0
  then s:= s-1;
  else Suspende el proceso;
unlock;
```

Signal

```
lock
  if Hay procesos suspendidos
  then Desbloquea uno de los procesos;
  else  s:= s+1;
unlock;
```

- Alternativas para implementarlos mecanismos **lock** y **unlock**:
  - Inhibición de la interrupciones de planificación.
  - Utilizar las instrucciones especiales “Test and Set” (TAS) de que están dotadas algunas CPUs.

Crítica de los semáforos.

- Ventajas de los semáforos:
  - Resuelven todos los problemas que presenta la concurrencia.
  - Estructuras pasivas muy simples.
  - Fáciles de comprender.
  - Implementación muy eficiente.
- Peligros de los semáforos:
  - Son de muy bajo nivel y un simple olvido o cambio de orden conducen a bloqueos.
  - Requieren que la gestión de un semáforo se distribuya por todo el código. La depuración de los errores en su gestión es muy difícil.
- Los semáforos son los componentes básicos sobre los que se pueden construir otros mecanismos de sincronización.

Declaración de una región crítica.

- La definición de una región crítica implica la declaración de dos elementos:
  - Variable compartida respecto de la que se definirán regiones críticas:

    var registro: **shared** Tipo_Registro;
  - Cada bloque de código que se requiere que se ejecute en régimen de exclusión mutua (regiones críticas).

    region registro **do** Bloque_de_código;
- Un proceso que trata de ejecutar una región crítica, compite con otros procesos que también lo intentan:
  - Si gana el acceso ejecuta su bloque en régimen exclusivo.
  - Si pierde el acceso se encola en la lista asociada a la variable compartida y se suspende a la espera de que la región crítica quede libre.
- Regiones críticas relativas a variables compartidas diferentes se pueden ejecutar concurrentemente.

Parque público con regiones críticas

```
program Control_Parque_Publico;
   var cuenta: shared Natural;
process type Torno;
   var visitante:Natural;
begin
   for visitante :=1 to 20 do
      region cuenta do cuenta:=cuenta+1;
   end;
var torno_1, torno_2: Torno;

begin
   region cuenta do cuenta:=0;
   cobegin torno_1, torno_2; coend;
   region cuenta do writeln(cuenta);  -- Aquí ya no hay concurrencia
end;
```

Regiones críticas condicionales.

- Las **regiones críticas condicionales** se definen en función de dos componentes:
  - Declarar una variable compartida

    var variable: **shared** TipoVariable;
  - Declaración del bloque crítico

    region variable **when** expresión_Booleana **do** Bloque_de_Sentencias;
- **Semántica:**
  - El proceso que ejecuta la región crítica debe obtener el acceso exclusivo a la variable compartida.
  - Sin liberar el acceso exclusivo se evalúa la expresión booleana:
    - Si resulta True: Se ejecuta el código del bloque.
    - Si resulta False: El proceso libera la exclusividad y queda suspendido.
  - Cuando un proceso concluye la ejecución del bloque protegido, libera el acceso a la variable y se da acceso a otro proceso de la lista de espera de la variable (que volverá a evaluar la expresión booleana).

Ejemplo: Productor-Consumidor con buffer finito (1).

```
program Productor_consumidor;
  const LONG_BUFFER =  5;
  type TipoDato = ........
  TipoBuffer = record    dato: array [1..LONG_BUFFER] of TipoDato;
                         nextIn, nextOut, cuenta: Integer;
                end;
   var buffer : shared TipoBuffer;
   process Productor begin ... end;
   process Consumidor; begin ... end;
begin
    region buffer do begin
       buffer.nextIn:=1; buffer.nextOut:=1; buffer.cuenta:=0;
    end;
    cobegin Productor; Consumidor; coend;
end.
```

Ejemplo: Productor-Consumidor con buffer finito (2).

```
process Productor;
  var dato: TipoDato;
begin
   repeat
       ....          -- Produce el dato
      region Buffer when (buffer.cuenta < LONG_BUFFER) do begin
         buffer.datos[nextin]:= dato;
         buffer.nextIn := (buffer.nextIn mod LONG_BUFFER) +1;
         buffer.cuenta:= buffer.cuenta + 1;
      end;
   forever;
end;
```

Ejemplo: Productor-Consumidor con buffer finito (3).

```
process Consumidor;
var dato: Tipo_dato;
begin
    repeat
        region buffer when (buffer.cuenta>0) do begin
            buffer.datos[nextIn]:= dato;
            buffer.nextOut := (buffer.nextOut mod LONG_BUFFER) +1;
            buffer.cuenta:= buffer.cuenta  - 1;
        end;
    forever
end;
```

Crítica de las regiones críticas.

- Las regiones críticas presenta un nivel de abstracción muy superior al semáforo, su uso es menos proclive al error.

- Sin embargo presenta los siguientes problemas:
  - Las sentencias relativas a una misma variable compartida están dispersas por todo el código de la aplicación, esto dificulta su mantenimiento.
  - La integridad de las variables compartidas está comprometida, ya que no hay ninguna protección sobre lo que el programador de cualquier módulo puede realizar sobre ella.
  - No son fáciles de implementar. Requieren asociar a cada variable compartida dos listas, la "lista principal" en la que esperan los procesos que tratan de acceder y la "lista de eventos" en la que se encolan los procesos que tras acceder no han satisfecho la condición de guarda.

Monitor

- Son módulos que encierran los recursos o variables compartidas como componentes internos privados y ofrecen una interfaz de acceso a ellos que garantiza el régimen de exclusión mutua.
- La declaración de un monitor incluye:
  - Declaración de las constantes, variables, procedimientos y funciones que son **privados** del monitor (solo el monitor tiene visibilidad sobre ellos).
  - Declaración de los procedimientos y funciones que el monitor **exporta** y que constituyen la interfaz a través de las que los procesos acceden al monitor.
  - **Inicialización del monitor**: bloque de código que se ejecuta al ser instanciado o inicializado. Su finalidad es inicializar las variables y estructuras internas.
- El monitor garantiza el acceso al código interno en régimen de exclusión mutua. Tiene asociada una lista en la que se incluyen los procesos que al tratar de acceder al monitor son suspendidos.

Sintaxis del monitor.

```
monitor Identificador;
    -- Lista de procedimientos y funciones exportadas
   export Identificador_procedimiento_1;
   export Identificador_procedimiento_2;
   ....
    -- Declaración de constantes, tipos y variables internas
   ....
    -- Declaración de procedimientos y funciones
   ....
begin
    -- Sentencias de inicialización
....
end;
```

Aspectos característicos del monitor.

- Las estructuras de datos internas del monitor cuya finalidad es ser compartidas por múltiples procesos concurrentes, solo pueden ser inicializadas, leídas y actualizadas por código propio del monitor.

- Los únicos componentes del monitor públicos (visibles desde módulos externos) son los procedimientos y funciones exportadas.

- El monitor garantiza el acceso mutuamente exclusivo a los procedimientos y funciones de la interfaz. Si son invocados concurrentemente por varios procesos, solo la ejecución de un procedimientos del monitor es permitido. Los procesos no atendidos son suspendidos hasta que la ejecución del procedimiento atendido acabe.

- Dado que todo el código relativo a un recurso o a una variable compartida está incluido en el módulo del monitor, su mantenimiento es mas fácil y su implementación es mas eficiente.

Variables “Condition”

- El monitor resuelve el acceso seguro a recursos compartidos, pero no tiene las capacidades plenas de sincronización
  - Por ello se definen las variables tipo **condition** (que solo pueden declararse dentro de un monitor).

    var inviables: **condition**;
  - **Valores**: No toma ningún tipo de valor, pero tiene asociada una lista de procesos suspendidos
  - **Operaciones**:
    - **delay** suspende el proceso que lo ejecuta y lo incluye en la lista de una variable Condition.
    - **resume** Reactiva un proceso de la lista asociada a una variable condition.
    - **empty** Función que retorna True si la lista de procesos de la variable condition está vacía.

Operación delay.

- Suspende el proceso que la ejecuta (que ha invocado el procedimiento del monitor en que se encuentra) y lo introduce en la cola asociada a la variableCondition.

  delay (variableCondition) ;

- Semántica de la operación:
  - Cuando un proceso ejecuta la operación delay, se suspende **incondicionalmente**.
  - Cuando un proceso ejecuta la operación delay, se libera la capacidad de acceso que dispone el proceso.
  - En la cola de una variable condition pueden encontrarse suspendidos un número ilimitado de procesos.

Operación resume.

- Ejecutada sobre una variable tipo condition, se reactiva uno de los procesos de la lista asociada a la variable.

  resume(variableCondition);

- Semántica:
  - Si la cola de la variable está vacía, equivale a una operación null.
  - Esta operación tiene en su definición la inconsistencia de que tras su ejecución existen dos procesos activos dentro del monitor, lo que contradice su principio de de operación. Diferentes modelos del procedimiento resume, que han sido utilizados:
    - **Reactiva y Continua**: El proceso activado se pone en la cola de espera para entrar al monitor. El proceso que ha invocado resume puede seguir ejecutando dentro del monitor.
    - **Reactivación_Inmediata**: El proceso que ha invocado resume sale del monitor. El único proceso activo dentro del monitor es el reactivado.
    - **Reactiva y espera**: El proceso que ejecuta resume se pone en la cola de espera para entrar al monitor. El proceso activado es el que tiene el acceso sobre el recurso. Cuando el proceso suspendido vuelve a entrar al monitor ejecuta la sentencia que sigue a la sentencia resume que le bloqueó.

Implementación de un semáforo mediante monitores

```
monitor Semaforo;
    export wait, signal, initial;
     var value: Natural;
         bloqueados: condition;

    procedure initial(v:Natural); begin value:=v; end;

    procedure wait;
    begin
      if (value=0) then
           delay(bloqueados);
      value:= value-1;
    end;

procedure Signal;
begin
   value:= value+1;
   resume(bloqueados);
end;

begin
  value:=1;    (* Por defecto semáforo abierto *)
end;
```

Productor Consumidor con buffer limitado.

```
monitor Buffer;
  export pone, toma;
  const SIZE=5;
  var datos: array[0..SIZE-1] of  TipoDato;
        cuenta:Natural;
        lleno,vacio: condition;
        nextIn,nextOut:0..SIZE-1;
  procedure pone(d:Tipo_Dato);
  begin
    if (cuenta>=SIZE) then delay(lleno);
    datos[NextIn]:= d;
    cuenta:= cuenta+1;
    nextIn:= (nextIn+1)mod SIZE;
    resume(vacio);
  end;
```

```
procedure toma(var d:TipoDato);
begin
   if (cuenta=0) then delay(vacio);
   d:= datos[nextOut];
   cuenta:= cuenta-1;
   nextOut:=(nextOut+1) mod SIZE;
   resume(lleno);
end;
```

```
begin cuenta:=0; nextIn:=0; nextOut:=0; end;
```

Comparación entre modelos de concurrencia.

- Utilizamos dos criterios complementarios de comparación:
  - **Poder expresivo**: Es la capacidad de la primitiva para implementar algoritmos o resolver problemas de sincronización en programas concurrentes
    - Si una primitiva tiene capacidad de implementar otra, tiene al menos su capacidad expresiva
    - Todas las primitivas que hemos estudiado tienen la misma capacidad expresiva
  - **Facilidad de uso**: Criterio subjetivo que se refiere a aspectos, tales como:
    - Como de natural es el uso de la primitiva.
    - Como de fácil es combinar la primitiva con otras sentencias del lenguaje.
    - Cuan proclive es la primitiva para que el programador cometa errores.

Facilidad de uso de las primitivas de sincronización.

- La sincronización a través de Semáforos presenta la capacidad plena,
  - Pero su facilidad de uso es muy pobre por su bajo nivel de abstracción y su gestión distribuida.
- La Invocación de Procedimientos Remotos (paso de mensajes) es de alto nivel de abstracción y de uso muy seguro, aunque presenta dos problemas:
  - Implementa los objetos pasivos con un modelo de módulo activo que no corresponde a lo que el programador espera.
  - Conduce a implementaciones ineficientes como consecuencia del gran número de cambios de contexto.
- Las Regiones Críticas se ven superadas por los monitores que presentan todas sus ventajas y ninguno de sus inconvenientes
- Los Monitores incrementan la abstracción y concentran la gestión de los datos compartidos
  - Puede considerarse innecesario introducir esta nueva primitiva cuando con los conceptos de proceso e invocación remota sería suficiente