Lección 9: Programación concurrente con semáforos

- Semáforos: sintaxis y semántica
- Técnicas de programación con semáforos:
  - el ejemplo de la sección crítica
  - un ejemplo de barreras
  - el caso de productores/consumidores con buffer de capacidad 1
  - el caso de productores/consumidores con buffer de capacidad k>1
- Más sobre sincronización en Ada: objetos protegidos

Semáforos: sintaxis y semántica

- La sincronización por espera activa tiene inconvenientes:
  - los algoritmos son difíciles de diseñar y verificar
  - se mezclan variables del problema a resolver y del método de resolución
  - en los casos de multi-programación, se ocupa innecesariamente el procesador "haciendo nada"
- Los semáforos son una de las primeras soluciones
  - son una solución conceptual, que puede implemetarse de diferentes formas
  - Propuestos por Dijkstra en 1968

Semáforos: sintaxis y semántica

- **Semáforo**: instancia de un tipo abstracto de dato, con sólo dos operaciones...:
  - ***wait***: el proceso se detiene hasta que ocurra cierto evento
  - ***send***: el proceso avisa de la ocurrencia de un evento

P

V
signal

- ... y la siguiente propiedad:

  Invariante de un semáforo: Sea `s` un semáforo con un valor inicial asociado entero s_0, sean `nW` y `nS` los números de operaciones ***wait*** y ***send*** ejecutadas sobre `s`. En todos los estados visibles del programa se ha de cumplir que

$$I_s: nW <= nS + S_0$$

Notar que puede obligar a que una operación wait se retrase: permite la sincronización

Semáforos: sintaxis y semántica

- Una forma de ver un semáforo **S**:
  - variable entera no negativa con un valor inicial S_0
  - el valor del semáforo es $S = S_0 + nS - nW$
  - con lo que el invariante se puede escribir $I_S: S >= 0$
- Sintaxis y semántica de los operadores ***wait*** y ***send***:

```
wait(S): <await (S>0)
            S:=S-1>
```

```
send(S): <S:=S+1>
```

Semáforos: sintaxis y semántica

- Caso particular: **semáforos binarios**

```
IB: 0 <= B <= 1
```

- ***wait*** y ***send*** para semáforos binarios:

```
wait(B): <await (B>0)
                B:=B-1>
```

```
send(B): <await (B<1)
                B:=B+1>
```

- Asumiremos en nuestro lenguaje el tipo predefinido ***semáforo***:

```
Vars mutex:sem := 0
     m:vector(1..n) de sem := (1..n,3)
```

Semáforos: sintaxis y semántica

- Reglas para semáforos generales:

$$\frac{(Q \wedge S>0) \rightarrow R_S^{S-1}}{\{Q\}\ \textit{wait}(s)\ \{R\}}$$

$$\frac{Q \rightarrow R_S^{S+1}}{\{Q\}\ \textit{send}(s)\ \{R\}}$$

- Reglas para semáforos binarios:

$$\frac{(Q \wedge B>0) \rightarrow R_B^{B-1}}{\{Q\}\ \textit{wait}(B)\ \{R\}}$$

$$\frac{(Q \wedge B<1) \rightarrow R_B^{B+1}}{\{Q\}\ \textit{send}(B)\ \{R\}}$$

- Los semáforos se pueden usar para implementar protocolos de sección crítica
  - vamos a ver algunos ejemplos

El ejemplo de la sección crítica

```
Vars en:vector(1..n)
      de Ent:=(1..n,0)
P(i:1..n)::
Mq TRUE
        --en(i)=0
   en(i):=1
         --en(i)=1
   secCrit
   en(i):=0
         --en(i)=0
   secNoCrit
FMq
```

```
I: en(1)+..+en(n)<=1
```

```
Vars en:vector(1..n)
      de Ent:=(1..n,0)
P(i:1..n)::
Mq TRUE
        --en(i)=0 ∧ I
   <await ????
      en(i):=1
   >
        --en(i)=1 ∧ I
   secCrit
   <await ????
      en(i):=0
   >
        --en(i)=0 ∧ I
   secNoCrit
FMq
```

El ejemplo de la sección crítica

```
I: en(1)+..+en(n)<=1
```

```
Vars en:vector(1..n)
       de Ent:=(1..n,0)
P(i:1..n)::
Mq TRUE
         --en(i)=0 ∧ I
   <await en(1)+..+en(n)=0
      en(i):=1
   >
         --en(i)=1 ∧ I
   secCrit
   en(i):=0
         --en(i)=0 ∧ I
   secNoCrit
FMq
```

```
Vars libre:semBin:=1
P(i:1..n)::
Mq TRUE
         --en(i)=0 ∧ I
   wait(libre)
         --en(i)=1 ∧ I
   secCrit
   send(libre)
         --en(i)=0 ∧ I
   secNoCrit
FMq
```

El ejemplo de la sección crítica

- Bajo determinadas condiciones, acciones atómicas se pueden sustituir por operaciones sobre semáforos
- Es necesario que concurran las siguientes circunstancias:
  - guardas distintas se refieren a variables distintas, y éstas sólo se referencian en inst. atómicas
  - cada guarda es (se puede poner) de la forma "`expr > 0`"
  - cada inst. atómica guardada contiene una asignación decrementando el valor de "`expr`"
  - cada inst. atómica no guardada incremente "`expr`" en una unidad
- Entonces
  - un semáforo para cada guarda
  - variables en las guardas se convierten en auxiliares
  - cada inst. atómica se puede poner como inst. sobre semáforos

Un ejemplo de barreras

- **Problema**: algoritmo para dos procesos iterativos que sincronizan cada iteración mediante una barrera
- **Sincronización**:
  - dos variables por proceso que contabilizan el número de veces que cada proceso alcanza o deja atrás la barrera de sincronización
  - $\texttt{llega}_i$: proceso "i" llega a la barrera (está esperando al otro)
  - $\texttt{deja}_i$: proceso "i" pasa la barrera (y vuelve a iterar)
- **Objetivo**: que un proceso no pueda pasar la barrera antes de que el otro llegue a la misma

```
I: deja1<=llega2 ∧ deja2<=llega1
```

Un ejemplo de barreras

```
Vars llega1:Ent:=0,deja1:Ent:=0
     llega2:Ent:=0,deja2:Ent:=0
P1::
Mq TRUE
  ....
      --llega1=deja1
  <llega1:=llega1+1>
      --llega1=deja1+1
  <deja1:=deja1+1>
      --llega1=deja1
  ....
FMq
```

```
P2::
Mq TRUE
  ....
      --llega2=deja2
  <llega2:=llega2+1>
      --llega2=deja2+1
  <deja2:=deja2+1>
      --llega2=deja2
  ....
FMq
```

```
I: deja1<=llega2 ∧ deja2<=llega1
```

Un ejemplo de barreras

```
I: deja1<=llega2 ∧ deja2<=llega1
```

```
Vars llega1:=0,deja1:=0
     llega2:=0,deja2:=0
--I
P1::
Mq TRUE
  ....
      --I ∧ llega1=deja1
  <llega1:=llega1+1>
      --I ∧ llega1=deja1+1
  <await deja1<llega2
      deja1:=deja1+1
  >   --I ∧ llega1=deja1
  ....
FMq
```

```
P2::
Mq TRUE
  ....
      --I ∧ llega2=deja2
  <llega2:=llega2+1>
      --I ∧ llega2=deja2+1
  <await deja2<llega1
      deja2:=deja2+1
  >   --I ∧ llega2=deja2
  ....
FMq
```

Un ejemplo de barreras

I: deja1<=llega2 $\wedge$ deja2<=llega1

```
Vars bar1:sem:=0
     bar2:sem:=0
P1::                     P2::
Mq TRUE                  Mq TRUE
   ....                     ....
   send(bar1)               send(bar2)
   wait(bar2)               wait(bar1)
   ....                     ....
FMq                      FMq
```

El caso de los productores-consumidores....

- Problema
  - tenemos un sistema con M productores y N consumidores
  - para comunicar, disponen de un "buffer" compartido con capacidad para un único mensaje
  - un productor pone mensajes en el "buffer"
    - operación "**pon**"
  - un consumidor consume mensajes del "buffer"
    - operación "**quita**"
  - requisitos:
    - no debe haber pérdida de mensajes
    - un mensaje no debe ser leído por más de un consumidor

El caso de los productores-consumidores....

Productor::
Vars m:Mensaje
Mq TRUE
prepara m
pon m
FMq

Consumidor::
Vars m:Mensaje
Mq TRUE
quita m
usa m
FMq

```
I: eP<=tC+1 ∧ eC<=tP
```

```
Vars buff:Buffer
     eP:=0,tP:=0,eC:=0,tC:=0

Productor(1..M)::
Vars m:Mensaje
Mq TRUE
  prepara m
  <await eP<tC+1
      eP:=eP+1
  >
  buff:=m
  <tP:=tP+1>
FMq

Consumidor(1..N)::
Vars m:Mensaje
Mq TRUE
  <await eC<tP
      eC:=eC+1
  >
  m:=buff
  <tC:=tC+1>
  usa m
FMq
```

El caso de los productores-consumidores....

- Una solución con semáforos

```
I: huecoVacio+huecoLleno∈{0,1}

Vars buff:Buffer
     huecoVacio:sem:=1,huecoLleno:sem:=0

Productor(1..M)::             Consumidor(1..N)::
Vars m:Mensaje                Vars m:Mensaje
Mq TRUE                       Mq TRUE
  produce m                     wait(huecoLleno)
  wait(huecoVacio)              buff:=m
  buff:=m                       send(huecoVacio)
  send(huecoLleno)              usa m
FMq                           FMq
```

El caso de los productores-consumidores....

- El problema es el mismo que antes, pero
  - el buffer es capaz de almacenar K mensajes
  - entonces:
    - no K+1 “pon” seguidos, ni K+1 “quita” seguidos
    - todo “quita” debe ir precedido por al menos “pon”
- La solución es análoga con las salvedades:
  - una cola de hasta K mensajes
  - el invariante será `I: eP<=tC+K ∧ eC<=tP`

El caso de los productores-consumidores....

un productor y un consumidor

I: huecoVacio+huecoLleno∈{0..K}

```
Vars buff:cola(K)
     huecoVacio:sem:=K,huecoLleno:sem:=0

Productor::
Vars m:Mensaje
Mq TRUE
  prepara m
  wait(huecoVacio)
  encolar(m,buff)
  send(huecoLleno)
FMq

Consumidor::
Vars m:Mensaje
Mq TRUE
  wait(huecoLleno)
  desencolar(m,buff)
  send(huecoVacio)
  usar m
FMq
```

El caso de los productores-consumidores....

```
I: eP<=tC+K ∧ eC<=tP
```

```
Vars buff:cola(K)
     eP:=0,tP:=0,eC:=0,tC:=0
```

```
Productor(1..M)::
Vars m:Mensaje
Mq TRUE
  prepara m
  <await eP<tC+K
      eP:=eP+1
  >
  <encolar(m,buff)>
  <tP:=tP+1>
FMq
```

```
Consumidor(1..N)::
Vars m:Mensaje
Mq TRUE
  <await eC<tP
      eC:=eC+1
  >
  <desencolar(m,buff)>
  <tC:=tC+1>
  usa m
FMq
```

El caso de los productores-consumidores....

M productores
N consumidores

```
I: huecoVacio+huecoLleno∈{0..K} ∧ ...

Vars buff:cola(K)
     huecoVacio:sem:=K,huecoLleno:sem:=0
     unoPone:sem:=1,unoQuita:sem:=1
```

```
Productor(1..M)::
Vars m:Mensaje
Mq TRUE
  prepara m
  wait(huecoVacio)
  wait(unoPone)
  encolar(m,buff)
  send(unoPone)
  send(huecoLleno)
FMq
```

```
Consumidor(1..N)::
Vars m:Mensaje
Mq TRUE
  wait(huecoLleno)
  wait(unoQuita)
  desencolar(m,buff)
  send(unoQuita)
  send(huecoVacio)
  usar m
FMq
```

El caso de los productores-consumidores....

M productores
N consumidores

```
I: huecoVacio+huecoLleno∈{0..K} ∧ ...

Vars buff:cola(K)
     huecoVacio:sem:=K,huecoLleno:sem:=0
     unoPone:sem:=1,unoQuita:sem:=1
```

```
Productor(1..M)::
Vars m:Mensaje
Mq TRUE
  prepara m
  wait(huecoVacio)
  wait(mutexBuff)
  encolar(m,buff)
  send(mutexBuff)
  send(huecoLleno)
FMq
```

```
Consumidor(1..N)::
Vars m:Mensaje
Mq TRUE
  wait(huecoLleno)
  wait(mutexBuff)
  desencolar(m,buff)
  send(mutexBuff)
  send(huecoVacio)
  usar m
FMq
```

Ejercicios

- **Ejercicio 1**: dar una solución al problema de la barrera para n procesos
- **Ejercicio 2**: completar los procesos siguientes de manera que "d" no se puede ejecutar hasta que "a" haya acabado

```
P1::  | P2::
  a   |   c
  b   |   d
```

- **Ejercicio 3**: completar los procesos siguientes de manera que P2 puede ejecutar "d" si se ha ejecutado "e" ó "a"

```
P1::       | P2::       | P3::
Mq TRUE    | Mq TRUE    | Mq TRUE
  a        |   c        |   e
  b        |   d        |   f
FMq        | FMq        | FMq
```

Implementación de semáforos en Ada

- Dos alternativas básicas para implementar semáforos:
  - como tarea
  - uso de objetos protegidos

```
task type semaforo(valorInicial: natural)
is
   entry wait;
   entry send;
end semaforo;
```

```
task body semaforo is
   elValor: natural :=
                    valorInicial;
begin
   loop
      select
         when elValor > 0 =>
         accept wait do
            elValor := elValor-1;
         end wait;
      or
         accept send do
            elValor := elValor+1;
         end send;
      or
         terminate;
      end select;
   end loop;
end semaforo;
```

Implementación de semáforos en Ada

- Como paquete, mejor:

```
package semaforos is
  task type semaforo
        (valorInicial: natural) is
     entry wait;
     entry send;
  end semaforo;
end semaforos;
```

```
package body semaforos is
  task body semaforo is
    elValor: natural :=
                     valorInicial;
  begin
   loop
      select
         when elValor > 0 =>
         accept wait do
            elValor := elValor-1;
         end wait;
      or
         accept send do
            elValor := elValor+1;
         end send;
      or
         terminate;
      end select;
   end loop;
  end semaforo;
end semaforos;
```

... Ada

```
with semaforos; use semaforos;

procedure productoresConsumidoresSimple is
  m: constant integer := 3;
  n: constant integer := 5;

  buff: integer;
  huecoVacio: semaforo(1);
  huecoLleno: semaforo(0);

  task type productor;
  task type consumidor;

  task body productor is
    mens: integer;
  begin
    loop
      mens := ...;
      huecoVacio.wait;
      buff := mens;
      huecoLleno.send;
    end loop;
  end productor;
```

```
  task body consumidor is
    mens: integer;
  begin
    loop
      huecoLleno.wait;
      mens := buff;
      huecoVacio.send;
    end loop;
  end consumidor;

  productores: array(1..m) of productor;
  consumidores: array(1..n) of consumidor;
begin
  null;
end productoresConsumidoresSimple;
```

Más sincronización en ADA: tipos protegidos

- Métodos de sincronización en ADA:
  - variables compartidas
  - citas
  - objetos protegidos
- Objeto protegido
  - encapsula datos
  - restringe el acceso a los procs/funcs protegidos y a las "entry" protegidas
  - ADA asegura
    - que *la ejecución de los procs/funcs y "entry" se lleva a cabo en exclusión mutua*
      - implica acceso en mutex a lectura/escritura sobre las variables
    - funciones: no pueden modificar las variables
      - varios accesos concurrentes a funciones (respetando lo anterior)

Más sincronización en ADA: tipos protegidos

- Ejemplo: un entero con acceso en exclusión mutua

```
protected type shared_integer(initial_value: integer) is
   function read return integer;
   procedure write(new_value: integer);
   procedure increment(by: integer);
private
   the_data: integer :=
                 initial_value;
end shared_integer;
```

- es un tipo “limited” (no “=“ “:=“)
- interfaz:
  - procs, funcs, “entries”
  - discriminante: discreto/puntero
- ¿Orden de servicio?
  - el ARM no lo especifica
- Todos los datos en la parte “privada” (no en el body)

```
protected body shared_integer is
   function read return integer is
   begin
      return the_data;
   end;
   procedure write(new_value: integer) is
   begin
      the_data:= new_value;
   end;
   procedure increment(by: integer)is
   begin
      the_data:= the_data + by;
   end;
end shared_integer;
```

Más sincronización en ADA: tipos protegidos

- Sobre las "entries" protegidas:
  - también se ejecutan en mutex
  - puede leer/escribir sobre las variables privadas
  - siempre está guardada (barrera)
    - si barrera no activa, la tarea que llama se queda "en suspenso" hasta que se haga cierta y no haya otra tarea activa en el cuerpo
    - implementación natural para la sincronización por condición
  - una estrategia FIFO (salvo cambio forzado, claro)
  - el que llama puede poner la llamada en el marco de un *select* (con las mismas restricciones que quien acudía a citas ofertadas por tareas)
- Un ejemplo: un buffer limitado compartido por varias tareas

Más sincronización en ADA: tipos protegidos

```
buffer_size: constant integer := 10;
type index is mod buffer_size;
subtype count is natural range 0..buffer_size;

type buffer is array(index) of data_item;

protected type bounded_buffer is
   entry get(item: out data_item);
   entry put(item: in data_item);
private
   first: index := index'first;
   last: index := index'last;
   number_in_buffer: count := 0;
   buf: buffer;
end bounded_buffer;
```

```
my_buffer: bounded_buffer

.....
my_buffer.Put(my_item)
.....
```

```
protected body bounded_buffer is
   entry get(item: out data_item)
                when number_in_buffer /= 0 is
   begin
      item := buf(first);
      first := first + 1;
      number_in_buffer := number_in_buffer - 1;
   end;

   entry put(item: in data_item)
                when number_in_buffer < buffer_size is
   begin
      last := last + 1;
      buf(last) := item;
      number_in_buffer := number_in_buffer + 1;
   end;
end bounded_buffer;
```

Más sincronización en ADA: tipos protegidos

- Las barreras de las entradas protegidas se (re) evalúan cuando:
  - una tarea invoca una *entry* vacía
  - una tarea termina de ejecutar un procedimiento o entrada protegida y quedan tareas en la cola de alguna entrada cuya barrera usa alguna variable modificada desde la última vez que se evaluó
  - ¿Qué pasa en el caso de las funciones?
    - ojo con el uso de variables modificables por una función interna o una tarea externa
  - Ojo: no se reevalúan cuando se modifiquen variables globales
    - por lo tanto, no conviene usar variables globales en barreras

Más sobre uso de objetos protegidos

- ¿Qué pasa cuando una tarea invoca un procedimiento/entrada protegida de un obj. protegido que está siendo usado por otra tarea?
  - reglas definidas por el ARM (ver [Burns-Wellings])
- Si alguien está ejecutando una función, el objeto protegido tiene un "*cerrojo de lectura*" (*read lock*) activo
- Si alguien está ejecutando un proc. o entrada protegida, el objeto protegido tiene un "*cerrojo de lectura-escritura*" (*read-write lock*) activo
- Lo que sigue, en orden, es una "película" de lo que ocurre cuando una tarea trata de invocar algo de un objeto protegido:

Más sobre uso de objetos protegidos

```
1) si el objeto tiene un cerrojo de lectura activo, y se invoca una función,
la función se ejecuta, y nos vamos al punto 14
2) si el objeto tiene un cerrojo de lectura activo y se invoca un proc. o
entrada protegida, la ejecución de la llamada se retrasa hasta que no haya
ninguna tarea usando el objeto
3) si el objeto tiene un cerrojo de lectura-escritura activo, la tarea se
retrasa mientras hay tareas con requisitos de acceso en conflicto con él
4) si el objeto no tiene ningún cerrojo activo, y se invoca una función, se
activa el cerrojo de lectura, y vamos a 5
5) la función se ejecuta, y vamos a 14
6) si el objeto no tiene ningún cerrojo activo y se invoca un proc. o entrada
protegida, se activa el cerrojo de lectura-escritura, y vamos a 7
7) si la llamada es un proc., se ejecuta, y vamos a 10
8) si la llamada es una entrada, la barrera asociada se evalúa; si da cierto,
el cuerpo se ejecuta, y vamos a 10
9) si la barrera da falso, la llamada va a la cola asociada a la entrada, y
vamos a 10
```

Más sobre uso de objetos protegidos

```
10) todas las barreras con tareas en espera y que referencian variables que
han podido cambiar desde la última vez que se evaluaron, son re-evaluadas, y
vamos a 11
11) si hay varias barreras abiertas, se elige una, se ejecuta el cuerpo
asociado, y vamos a 10
12) si no hay barreras abiertas con tareas en la cola, vamos a 13
13) si hay tareas esperando para acceder al objeto protegido, o bien una única
que necesite el cerrojo de lectura-escritura accede al objeto, o bien todas
que deseen ejecutar una función se activan, ejecutándose los pasos 5), 7) u
8), según sea el caso. Si no hay tareas esperando, el protocolo termina
14) si no hay ninguna tarea activa usando el objeto protegido vamos a 13. En
caso contrario, el protocolo termina.
```

Implementación de semáforos mediante objetos protegidos

```
protected type semaforo (vI: natural) is
   entry wait;
   procedure send;
private
   val: natural := vI
end semaforo;

protected body semaforo is
begin
   entry wait when val>0 is
   begin
      val := val-1;
   end;
   procedure send is
   begin
      val := val+1;
   end;
end semaforo;
```

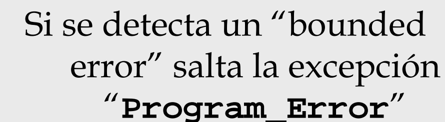
```
s: semaforo(3);

...
s.wait;
...
s.send;
...
```

- ¿tarea u objeto protegido?
  - tarea: elemento activo
    - terminación dependiendo de estructura
  - objeto: elemento pasivo
    - desaparece al terminar el bloque en que se declara

La "buena educación" en el uso de objetos protegidos

- **Recomendación**: el código a ejecutar en un proc/func/entry de un objeto protegido debe ser tan pequeño como se pueda
  - durante su ejecución, otras tareas están esperando
- Se considera un "*bounded error*" ejecutar operaciones "*potencialmente bloqueantes*":
  - instrucción "select"
  - instrucción "accept"
  - llamada a una "entry"
  - instrucción "delay"
  - creación o activación de tareas
  - llamar a un subprograma con instrucciones potencialmente bloqueantes



La “buena educación” en el uso de objetos protegidos

- También es de mala educación llamar a un proc. externo que invoque a un proc./entry interno
  - el cerrojo de lectura-escritura ya está activo, por lo que seguro que se bloquea

- “**Bounded errors**: The language rules define certain kinds of errors that need not be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. The errors of this category are called bounded errors. The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception **Program_Error**.”

ARM95

La “buena educación” en el uso de objetos protegidos

```
procedure malaEducacion is

protected type uno is
   procedure p;
end;
u: uno;

protected type dos is
   entry e;
private
   n: integer := 1;
end;
d: dos;

protected body uno is
   procedure p is
   begin
      d.e;
   end;
end;
```

```
protected body dos is
   entry e when n>3 is
   begin
      null;
   end;
end;

begin
   u.p;
end;
```

```
$> gnatmake malaEducacion
gcc -c malaeducacion.adb
malaeducacion.adb:27:18: warning: potentially
         blocking operation in protected operation
ganatbind -x malaeducacion.ali
gnatlink malaeducacion.ali
```