Semáforos

Lecturas:
Ben-Ari, secciones 4.1, 4.2, 4.3, 4.6
Andrews, intro. cap. 4 y sección 4.1

Manuel Carro

Universidad Politécnica de Madrid

Este texto se distribuye bajo los términos de la *Creative Commons License*

Semáforos
Definición

- Tipo abstracto de datos (Dijkstra)
- Especificación de comportamiento:

sem $\in \mathcal{N}$

```
Init(sem, n)    ≡ ⟨ sem := n ⟩
Wait(sem)       ≡ ⟨ AWAIT sem>0 → sem := sem-1 ⟩
Signal(sem)     ≡ ⟨ sem := sem + 1 ⟩
```

- Código entre $\langle \cdots \rangle$: **acción atómica**
- **Init(sem, n)**: sólo inicialización valores
- **AWAIT** *Cond* suspende tarea hasta cumplimiento de *Cond*
- **Wait(sem)**: suspende hasta que **sem** $>$ **0**, entonces decrementa
- **Signal(sem)**: incrementa **sem**

Exclusión mutua con semáforos

```
X := ...;   --Compartida
Init (Mutex, 1);  --Nadie en sección crítica
```

```
Wait(Mutex);
X := X + X;     -- S.C.
Signal(Mutex);
```

Invariante:
`Mutex = 0` $\leftrightarrow$ algún proceso en sección crítica

Wait:
- Puede decrementar si sección crítica está libre
- Suspende en otro caso

Signal:
- Siempre puede incrementar
- Puede rearrancar alguna tarea suspendida

Semáforos

Clases e implementación

- Binarios ($\texttt{sem} \in \{0,1\}$): exclusión mutua
- Generales ($\texttt{sem} \in \mathcal{N}$): sincronización más avanzada
- Semáforos binarios: **Signal** cuando ya tiene un valor 1 $\longrightarrow$ efecto indefinido

- No son parte nativa de Ada
- Implementación (entre las librerías de la asignatura) como tipos en el paquete `Semaphores` (`semaphores.{ads,adb}`):
  Tipo `Semaphore` : define un tipo paramétrico (para establecer el máximo valor del semáforo)
  (Sub)Tipo `Bin_Semaphore` : binarios
  (Sub)Tipo `Gen_Semaphore` : generales

Semáforos en Ada

Paquete `Semaphores`

- Establecimiento del máximo valor del semáforo en la declaración de variable:

```
Sem: Semaphores.Semaphore (Max => 5);
```

- Inicializados al **máximo** valor por defecto
- Subtipos para semáforos binarios y generales:

```
subtype Bin_Semaphore is Semaphore (Max => 1);
subtype Gen_Semaphore is Semaphore (Max => Natural'Last);
```

- **Importante:** la tarea *madre* o una de las tareas hijas debe inicializarlo con el valor deseado si éste no coincide con el máximo: `Init (Sem, N)`

Semáforos en Ada

Sólo diferencias sintácticas con el esquema ya visto:

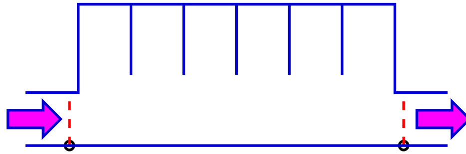
```
X: Natural := 1;        -- Variable compartida

-- Semáforo binario inicializado a 1
Mutex: Semaphores.Bin_Semaphore;

task type Add;
task body Add is
begin
   Wait (Mutex);       -- Entrada en sección crítica
   X := X + X;         -- Sección crítica
   Signal(Mutex);      -- Salida de sección crítica
end Add;
```

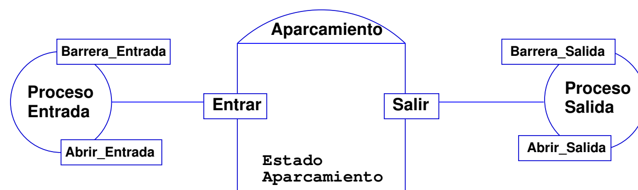
Semáforos generales: sincronización condicional

- Binarios: exclusión mutua (ya sabíamos hacerla)
- Generales: misma definición (excepto que **sem** $\in \mathcal{N}$), más potencia
- Ejemplo: aparcamiento 1 entrada / 1 salida



- Barreras / detectores de entrada y salida
- Permitir entrar coches (si hay sitio)
- *Descontar* los que vayan saliendo

Operaciones externas, procesos, recursos



procedure Barrera_Entrada: detecta coche a la entrada; **bloqueante**

procedure Abrir_Entrada: abre la barrera de entrada

procedure Barrera_Salida: detecta coche a la salida; **bloqueante**

procedure Abrir_Salida: abre barrera saluda

`Estado Aparcamiento`: depende de la implementación

Aparcamiento: solución 0

Programamos controlador barreras aparcamiento

```
N_C: Tipo_Tam := 0;

loop -- Salida
   Barrera_Salida;
   N_C := N_C - 1;
   Abrir_Salida;
end loop;
```

```
loop -- Entrada
   Espacio := False;
   Barrera_Entrada;
   while not Espacio loop
      if N_C < Max then
         N_C := N_C + 1;
         Espacio := True;
      end if;
   end loop;
   Abrir_Entrada;
end loop;
```

Aparcamiento: solución 1

Con un semáforo binario

```
Mutex: Bin_Semaphore;
N_C: Tipo_Tam := 0;
```

```
loop  -- Tarea salida
   Barrera_Salida;
   Wait(Mutex);
   N_C:= N_C - 1;
   Signal(Mutex);
   Abrir_Salida;
end loop;
```

```
loop  -- Tarea entrada
   Barrera_Entrada;
   Espacio:= False;
   while not Espacio loop
      Wait(Mutex);
      if N_C < Max then
         N_C := N_C + 1;
         Espacio:= True;
      end if;
      Signal(Mutex);
   end loop;
   Abrir_Entrada;
end loop;
```


Solución 1: protección de N_C

- Recurso compartido: número coches aparcados
- Acceso en exclusión mutua
- Pero: aparcamiento lleno ⟶ espera activa

```
while not Espacio loop
   Wait(Mutex);
   if N_C < Max then
      N_C := N_C + 1;
      Espacio:= True;
   end if;
   Signal(Mutex);
end loop; -- Hay espacio
```

- **¿Llevar mutex fuera de bucle principal?**
- **¿Encapsular sólo** N_C := N_C + 1**?**
- **¿No encapsular** Espacio := True**?**

Solución 2

```
Mutex, No_Lleno: Bin_Semaphore;
N_C: Tipo_Tam := 0;
```

```
loop -- Salida
   Barrera_Salida;
   Wait(Mutex);
   N_C := N_C - 1;
   if N_C = MAX - 1 then
      Signal(No_Lleno);
   end if;
   Signal(Mutex);
   Abrir_Salida;
end loop;
```

```
loop -- Entrada
   Barrera_Entrada;
   Wait(No_Lleno);
   Wait(Mutex);
   N_C := N_C + 1;
   if N_C < Max then
      Signal(No_Lleno);
   end if;
   Signal(Mutex);
   Abrir_Entrada;
end loop;
```

Solución 2: fuego cruzado

- Mutex: protege sección crítica
- No_Lleno: sincroniza (sincronización condicional)
- No_Lleno = 0 ⟷ N_C = MAX
  (⟶ bloqueo si aparcamiento lleno)
- No_Lleno = 0 *sii* ha entrado el último coche
- No obvio — pero no hace espera activa
- **¿Intercambiar Wait(No_Lleno) y Wait(Mutex)?**

Solución 3

```
Huecos: Semaphore(Max);

loop  -- Salida                 loop  -- Entrada
   Barrera_Salida;                 Barrera_Entrada;
   Signal(Huecos);                 Wait(Huecos);
   Abrir_Salida;                   Abrir_Entrada;
end loop;                       end loop;
```

- Wait(Huecos) permite paso si hay espacio
- Signal(Huecos) libera espacio
- ¿El poder de los semáforos?
- **Codificar con Peterson**
- **Repetir análisis suponiendo que hay dos (o más) entradas**

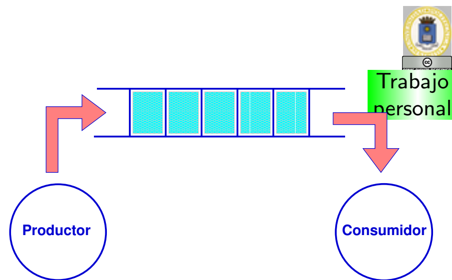
Semáforos desde la lejanía

- Algo truculentos y de bajo nivel
- Pero ampliamente disponibles (POSIX, API Win32)
- Sincronización condicional *ad-hoc*
- Linealización siempre posible, pero a veces no clara
- Mezclada con exclusión mutua
- Para implementar abstracciones de nivel superior

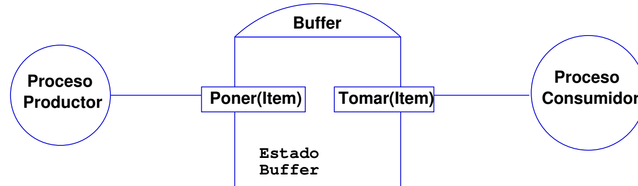
Buffer acotado

Trabajo personal

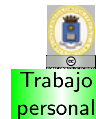
- Cola acotada
- Procesos leen y escriben



- *Buffer* lleno: productor espera
- *Buffer* vacío: consumidor espera



Solución 1: sólo exclusión mutua



Trabajo personal

```
Mutex:  Bin_Semaphore;
Buffer: Tipo_Buffer;
```

```
loop
   Producir(Un_Dato);
   Wait(Mutex);
   << Inserta en buffer >>
   Signal(Mutex);
end loop;
```

```
loop
   Wait(Mutex);
   << Quita de buffer >>
   Signal(Mutex);
   Utilizar(Un_Dato);
end loop;
```

Precondiciones secuenciales
vs.
Precondiciones concurrencia

Solucion 2

Trabajo personal

```
Mutex  : Bin_Semaphore;
Buffer: Tipo_Buffer;
Vacios: Semaphore(Max);
Llenos: Semaphore(Max);
-- Init(Llenos,0);
```

- Semáforos con dos misiones diferentes
- Esencialmente, aparcamiento simétrico
- Pero: estructura de datos separada de semáforo

```
loop -- Productor
   Producir(Un_Dato);
   Wait(Vacios);
   Wait(Mutex);
   <<Inserta en buffer>>
   Signal(Mutex);
   Signal(Llenos);
end loop;
```

```
loop -- Consumidor
   Wait(Llenos);
   Wait(Mutex);
   <<Quita de buffer>>
   Signal(Mutex);
   Signal(Vacios);
   Consumir(Un_Dato);
end loop;
```

Buffer de un dato

```
Dato:   Tipo_Dato;
Lleno:  Bin_Semaphore;  --
con Init(Lleno,0)
Vacio:  Bin_Semaphore;
```

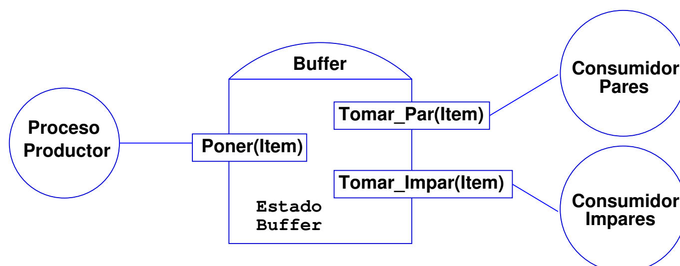
```
loop -- Productor
   Producir(Un_Dato);
   Wait(Vacio);
   Dato := Un_Dato;
   Signal(Lleno);
end loop;
```

```
loop -- Consumidor
   Wait(Lleno);
   Este_Dato:= Dato;
   Signal(Vacio);
   Consumir(Este_Dato);
end loop;
```

- **Establecer invariantes**
- **¿Por qué es más simple que el buffer general?**

Ejercicio: *buffer* par/impar

- Productor de números pares e impares (en cualquier orden)
- Consumidores:
  - Uno necesita números pares
  - Otro necesita números impares



- Usar **un** *buffer* de **un** dato