



Explicación Práctica de Semáforos



Ejercicios

SEMÁFOROS - Sintaxis

- **Declaraciones de Semáforos** si o si es mayor o igual que 0

sem s; → NO. Si o si se deben inicializar en la declaración

sem mutex = 1;

sem espera[5] = ([5] 1); arreglo de 5 sem, inicializamos cada uno en 1

- **Operaciones de los Semáforo**

$P(s)$ → $\langle \text{await } (s > 0) \ s = s-1; \rangle$ demora el proceso hasta que el valor de $s > 0$ y en forma atomica lo decrementa

$V(s)$ → $\langle s = s+1; \rangle$ en forma atomica, NO DEMORA, incrementa el valor del sem



EJERCICIO 1

Hay C chicos y hay una bolsa con caramelos que nunca se vacía. Los chicos de a UNO van sacando de a UN caramelo y lo comen. Los chicos deben llevar la cuenta de cuantos caramelos se han tomado de la bolsa.

```
int cant = 0;
```

Process Chico[id: 0..C-1]

```
{ while (true)
  { -- tomar caramelo
    cant = cant + 1;
    -- comer caramelo
  }
}
```

se necesita proteger el
incremento
del contador

```
int cant = 0;
```

```
sem mutex = 1;
```

Process Chico[id: 0..C-1]

```
{ while (true)
  { -- tomar caramelo
    P(mutex);
    cant = cant + 1;
    V(mutex);
    -- comer caramelo
  }
}
```

```
int cant = 0;
```

```
sem mutex = 1;
```

Process Chico[id: 0..C-1]

```
{ while (true)
  { P(mutex);
    -- tomar caramelo
    cant = cant + 1;
    -- comer caramelo
    V(mutex);
  }
}
```

Se puede
generar una
interferencia

¿Nos asegura que
NO puede haber más
de un chico tomando
un caramelo de la
bolsa?

Reducción de la
conurrencia

EJERCICIO 1

Lo que no pueden hacer al mismo tiempo es trabajar con el recurso compartido (acceder a la bolsa de caramelos e incrementar *cant*), pero SI que más de un chico coma al mismo tiempo → en la solución anterior NO SE MAXIMIZA LA CONCURRENCIA.

sem --> se inicializa en 1
porque la SC esta vacia

```
int cant = 0;
sem mutex = 1;

Process Chico[id: 0..C-1]
{ while (true)
    { P(mutex);
      -- tomar caramelo
      cant = cant + 1;
      V(mutex);
      -- comer caramelo
    }
}
```

No maximiza la
concurrency, porque al
comer un caramelo los
demás no pueden
ingresar a la SC hasta
que el chico actual
termine

En la sección Crítica se debe
hacer sólo lo que sea necesario
realizar con Exclusión Mutua, el
resto debe ir fuera de la SC para
maximizar la concurrencia

En este caso hacemos que la
variable cantidad no se incremente
al mismo tiempo, pero un chico
podría estar agarrando un caramelo
al mismo tiempo



EJERCICIO 2

Hay C chicos y hay una bolsa con caramelos **limitada a N caramelos**. Los chicos de a UNO van sacando de a UN caramelo y lo comen. Los chicos deben llevar la cuenta de cuantos caramelos se han tomado de la bolsa.

Comenzamos modificando la solución del Ejercicio 1 para que no intenten sacar más caramelos si la bolsa quedó vacía ($cant = N$)

```
int cant = 0;  
sem mutex = 1;
```

Process Chico[id: 0.. $C-1$]

```
{ while ( $cant < N$ )  
    { P(mutex);  
      -- tomar caramelo  
      cant = cant + 1;  
      V(mutex);  
      -- comer caramelo  
    }  
}
```

Se podrán sacar más de N caramelos.



EJERCICIO 2

El chequeo de la condición que indica que se debe tomar otro caramelo se debe proteger en una SC que también incluya la modificación de esa condición → en este caso el chequeo y $cant = cant + 1$

```
int cant = 0;
sem mutex = 1;
```

Process Chico[id: 0..C-1]

```
{ P(mutex);
  while (cant < N)
  { -- tomar caramelo
    cant = cant + 1;
    -- comer caramelo
  }
  V(mutex);
}
```

Un único chico
tomará los N
caramelos

```
int cant = 0;
sem mutex = 1;
```

Process Chico[id: 0..C-1]

```
{ P(mutex);
  while (cant < N)
  { -- tomar caramelo
    cant = cant + 1;
    V(mutex); p/ q el 1ero no coma  
todos los caramelos
    -- comer caramelo
  }
}
```

Libera la SC y
nunca más
asegura la EM

```
int cant = 0;
sem mutex = 1;
```

Process Chico[id: 0..C-1]

```
{ P(mutex);
  while (cant < N)
  { -- tomar caramelo
    cant = cant + 1;
    V(mutex);
    -- comer caramelo
    P(mutex); repite  
instrucción de  
antes del while
  }
}
```

Sólo un proceso
termina, el resto
se bloquea.

EJERCICIO 2

Al salir del *while* se debe liberar la SC para que otro proceso pueda acceder a ella y darse cuenta de que debe terminar su procesamiento.

```
int cant = 0;
sem mutex = 1;

Process Chico[id: 0..C-1]
{ P(mutex);
  while (cant < N)
  { -- tomar caramelo
    cant = cant + 1;
    V(mutex);
    -- comer caramelo
    P(mutex);
  }
  V(mutex);
}
```



EJERCICIO 3

Hay C chicos y hay una bolsa con caramelos limitada a N caramelos **administrada por UNA abuela**. Cuando todos los chicos han llegado llaman a la abuela, y a partir de ese momento ella N veces selecciona un chico aleatoriamente y lo deja pasar a tomar un caramelo.

Primero hay que hacer la Barrera entre los chicos. Para eso se puede usar un Contador compartido y todos esperan a que llegue a C

```
int contador = 0;
```

```
Process Chico[id: 0.. $C$ -1]
```

```
{ contador = contador + 1;
```

```
  if (contador ==  $C$ ) {
```

```
    despertar a los demorados
```

```
    despertar a la Abuela
```

```
  }
```

```
  else demorarse
```

```
    .....
```

```
}
```

Proteger el uso de *contador* en una SC que incluya el incremento y el chequeo del IF.

Usar un semáforo privado donde se duerme la abuela y acá se la despierta.

Cómo se demoran los procesos en ese punto → usar un semáforo para señalización de eventos (inicializado en 0)



EJERCICIO 3

```
int contador = 0;  
sem mutex = 1; exclusion mutua  
sem espera_abuela = 0; la abuela espera a q lleguen todos  
sem barrera = 0; esperan los chicos hasta que llegan todos
```

Process Chico[id: 0..C-1]

```
{ int i;  
  P(mutex);  
  contador = contador + 1; llega 1  
  if (contador == C) { llegaron todos  
    for i = 1..C → V(barrera); despierta a cada chico que estaba esperando en el semaforo barrera  
    V(espera_abuela); despierto abuela  
  }  
  V(mutex); libero SC para otro chico  
  P(barrera);  
  ..... los C-1 chicos q llegaron primero se duermen  
}
```

Se debe hacer un
V por cada
proceso demorado
en ese semáforo

PRIMERO SE HACE SIEMPRE EL V(mutex) Y
LUEGO EL P(barrera) porque se genera
bloqueo permanente de todos los procesos

Recordar siempre liberar la SC antes
de demorarse en *barrera*, sino se
bloquean todos los procesos.

EJERCICIO 3

Con la barrera completa se debe comenzar el proceso de tomar los caramelos.

```
int contador = 0;
sem mutex = 1;
sem espera_abuela = 0;
sem barrera = 0;
```

Process Chico[id: 0..C-1]

```
{ int i;
  P(mutex);
  contador = contador + 1;
  if (contador == C)
    { for i = 1..C → V(barrera);
      V(espera_abuela); }
  V(mutex);
  P(barrera);
  while (haya caramelos)
    esperar a que la abuela lo llame
    --tomar caramelo
    --comer caramelo
```

Process Abuela

```
{ int i;
  P(espera_abuela);
  for i = 1..N
    { selecciona chico ID
      despierta al chico ID
    }
  avisa que no hay mas caramelos
}
```

si pasa este semaforo es
porque llegaron todos los
chicos

Usar variable
booleana
seguir

Como espera a que lo despierten a él en
particular → usar semáforos privados
espera_chico[C]

}

EJERCICIO 3

```
int contador = 0;
bool seguir = true;
sem mutex = 1, espera_abuela = 0, barrera = 0, espera_chicos[C] = ([C] 0);
```

Process Chico[id: 0..C-1]

```
{ int i;
  P(mutex);
  contador = contador + 1;
  if (contador == C)
    { for i = 1..C → V(barrera);
      V(espera_abuela); }
  V(mutex);
  P(barrera); se duermen hasta q la abuela lo llame
  P(espera_chicos[id]); recibe el llamado
  while (seguir)
  { la abuela llama a uno específico, entonces hacemos semáforos privados para c/ chico
    --tomar caramelo
    --comer caramelo
    P(espera_chicos[id]); espera otro caramelo
  }
}
```

Process Abuela

```
{ int i, aux;
  P(espera_abuela);
  for i = 1..N
    { aux = (rand mod C); elige uno random
      V(espera_chicos[aux]); despierta a un chico particular, poniendo en 1 su semáforo
    }
  seguir = false; se terminan los caramelos, avisa a los chicos
}
```

Como se asegura la abuela que no hay más de dos chicos a la vez *tomando caramelo*. Puede despertar a uno cuando el anterior aún no ha tomado el caramelo → Se debe sincronizar tanto el inicio como el final de la interacción con otro semáforo.

EJERCICIO 3

```
int contador = 0;    bool seguir = true;
sem mutex = 1, espera_abuela = 0, barrera = 0, espera_chicos[C] = ([C] 0), listo = 0;
```

Process Chico[id: 0..C-1]

```
{ int i;
  P(mutex);
  contador = contador + 1;
  if (contador == C)
    { for i = 1..C → V(barrera);
      V(espera_abuela); }
  V(mutex);
  P(barrera);
  P(espera_chicos[id]);
  while (seguir)
    { --tomar caramelo
      V(listo); avisa a la abuela que ya tomo el caramelo
      --comer caramelo
      P(espera_chicos[id]);
    }
```

Process Abuela

```
{ int i, aux;
  P(espera_abuela);
  for i = 1..N
    { aux = (rand mod C);
      V(espera_chicos[aux]);
      P(listo); la abuela espera a que el chico le avise que ya tomo el caramelo
    }
  seguir = false;
}
```

Como se enteran los chicos que se modificó el valor de *seguir* → después de modificar el valor de *seguir* la abuela debe volver a despertar a cada chico para que entren a su SC y detecten este valor.

EJERCICIO 3

```
int contador = 0;    bool seguir = true;
sem mutex = 1, espera_abuela = 0, barrera = 0, espera_chicos[C] = ([C] 0), listo = 0;
```

Process Chico[id: 0..C-1]

```
{  int i;
   P(mutex);
   contador = contador + 1;
   if (contador == C)
       { for i = 1..C → V(barrera);
         V(espera_abuela); }
   V(mutex);
   P(barrera);
   P(espera_chicos[id]);
   while (seguir)
       { --tomar caramelo
         V(listo);
         --comer caramelo
         P(espera_chicos[id]);
       }
}
```

Process Abuela

```
{  int i, aux;
   P(espera_abuela);
   for i = 1..N
       { aux = (rand mod C);
         V(espera_chicos[aux]);
         P(listo);
       }
   seguir = false;
   for aux = 0..C-1 → V(espera_chicos[aux]);
   } despierta a los C chicos y les avisa que ya se terminaron
```



EJERCICIO 4

En una empresa de genética hay N clientes que envían secuencias de ADN para que sean analizadas y esperan los resultados para poder continuar. Para resolver estos análisis la empresa cuenta con 1 servidores que resuelve los pedidos de acuerdo al orden de llegada de los mismos.

Se necesitan los N procesos *Cliente* para enviar los pedidos y recibir los resultados, y el proceso *Servidor* para resolverlos

Process Cliente[id: 0.. N -1]

```
{ while (true)
  { --generar secuencia  $S$ 
    encolar ( $id$ ,  $S$ )
    esperar resultado
  }
}
```

¿Cómo?

Process Servidor

```
{ while (true)
  { recibir pedido ( $id$ ,  $S$ )
    resolver solicitud  $S$ 
    retornar el resultado a  $id$ 
  }
}
```



EJERCICIO 4

Se debe usar una *cola* C compartida donde se encolan los pedidos para mantener el orden. Al ser compartida el *push* y el *pop* se deben hacer con Exclusión Mutua → para eso usaremos el semáforo *mutex*

```
sem mutex = 1;  
cola C;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
  { --generar secuencia S  
    P(mutex);  
    push(C, (id, S));  
    V(mutex);  
    esperar resultado  
  }  
}
```

Process Servidor

```
{ secuencia sec;  int aux;  
  while (true)  
  { P(mutex);  
    pop(C, (aux, sec));  
    V(mutex);  
    resolver solicitud sec  
    retornar el resultado a aux  
  }  
}
```

¿Y si la cola está vacía?



EJERCICIO 4

No podemos hacer el *pop* sin estar seguros de que hay algo en la cola, sino se puede producir un error → ¿consultamos por el estado de la cola?

```
sem mutex = 1;  
cola C;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
    { --generar secuencia S  
      P(mutex);  
      push(C, (id, S));  
      V(mutex);  
      esperar resultado  
    }  
}
```

Process Servidor

```
{ secuencia sec; int aux;  
  while (true)  
    { P(mutex);  
      if not (empty(C)) → pop(C, (aux, sec));  
      V(mutex);  
      resolver solicitud sec  
      retornar el resultado a aux  
    }  
}
```

¿Y si está vacía? → se produce **BUSY WAITING**



EJERCICIO 4

Debe quedarse demorado en un semáforo hasta que seguro haya algo en la cola; cuando un cliente se encoló debe avisar por medio de ese semáforo, usado como *contador de recursos*.

```
sem mutex = 1, pedidos = 0;  
cola C;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
  { --generar secuencia S  
    P(mutex);  
    push(C, (id, S));  
    V(mutex);  
    V(pedidos);  
    esperar resultado  
  }  
}
```

Process Servidor

```
{ secuencia sec; int aux;  
  while (true)  
  { P(pedidos);  
    P(mutex);  
    pop(C, (aux, sec));  
    V(mutex);  
    --resolver solicitud sec  
    retornar el resultado a aux  
  }  
}
```

¿Cómo devolver el resultado al cliente?



EJERCICIO 4

Usaremos un vector *resultados* para poner el resultado para cada cliente, y un semáforo privado *espera* para cada uno con el cual se le avisa que ya está la respuesta en su posición del vector.

```
sem mutex = 1, pedidos = 0, espera[N] = ([N] 0);  
int resultados[N];  
cola C;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
  { --generar secuencia S  
    P(mutex);  
    push(C, (id, S));  
    V(mutex);  
    V(pedidos);  
    P(espera[id]);  
    --ver resultado de resultados[id]  
  }  
}
```

Process Servidor

```
{ secuencia sec; int aux;  
  while (true)  
  { P(pedidos);  
    P(mutex);  
    pop(C, (aux, sec));  
    V(mutex);  
    resultados[aux] = resolver(sec);  
    V(espera[aux]);  
  }  
}
```



EJERCICIO 5

En una empresa de genética hay N clientes que envían secuencias de ADN para que sean analizadas y esperan los resultados para poder continuar. Para resolver estos análisis la empresa cuenta con 2 servidores que van alternando su uso para no exigirlos de más (en todo momento uno está trabajando y el otro descansando); cada 5 horas cambia en servidor con el que se trabaja. El servidor que está trabajando, toma un pedido (de a uno de acuerdo al orden de llegada de los mismos), lo resuelve y devuelve el resultado al cliente correspondiente. Cuando terminan las 5 horas se intercambian los servidores que atienden los pedidos. Si al terminar las 5 horas el servidor se encuentre atendiendo un pedido, lo termina y luego se intercambian los servidores.

Nos basamos en la solución del ejercicio 4 para empezar. Los clientes no deberán modificarse, a ellos no le importa quien lo atiende. Hay que modificar el servidor y agregar un proceso *reloj* para que cuente las 5 horas de cada servidor.



EJERCICIO 5

¿Cómo resolvemos
el reloj?

```
sem mutex = 1, pedidos = 0, espera[N] = ([N] 0);  
int resultados[N]; cola C;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
    { --generar secuencia S  
      P(mutex);  
      push(C, (id, S));  
      V(mutex);  
      V(pedidos);  
      P(espera[id]);  
      --ver resultado de resultados[id]  
    }  
}
```

Process Servidor[id: 0..1]

```
{ secuencia sec; int aux;  
  while (true)  
    { espera su turno  
      inicia reloj  
      while (no termine el tiempo)  
        { P(pedidos);  
          P(mutex);  
          pop(C, (aux, sec));  
          V(mutex);  
          resultados[aux] = resolver(sec);  
          V(espera[aux]);  
        }  
    }  
}
```

Process Reloj

```
{ while (true)  
  { espera inicio  
    delay(5 hs);  
    avisa final del tiempo  
  }  
}
```



EJERCICIO 5

Usaremos un semáforo *inicio* para avisar al reloj que debe comenzar a correr las 5 horas. Una variable booleana *FinTiempo* para indicar que el tiempo termino.

```
sem mutex = 1, pedidos = 0, espera[N] = ([N] 0), inicio = 0;  
int resultados[N]; cola C; bool finTiempo = false;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
  { --generar secuencia S  
    P(mutex);  
    push(C, (id, S));  
    V(mutex);  
    V(pedidos);  
    P(espera[id]);  
    --ver resultado de resultados[id]  
  }  
}
```

Process Servidor[id: 0..1]

```
{ secuencia sec; int aux;  
  while (true)  
  { espera su turno  
    inicia reloj  
    while (no termine el tiempo)  
    { P(pedidos);  
      P(mutex);  
      pop(C, (aux, sec));  
      V(mutex);  
      resultados[aux] = resolver(sec);  
      V(espera[aux]);  
    }  
  }  
}
```

Como
manejamos
el turno de
cada servidor

Process Reloj

```
{ while (true)  
  { P(inicio);  
    delay(5 hs);  
    finTiempo = true;  
    V(pedidos);  
  }  
}
```

El servidor actual puede esperar en un **ÚNICO** semáforo tanto el pedido de un cliente como el fin del reloj → se le avisa por medio del semáforo *pedidos*

EJERCICIO 5

Cada servidor tendrá un semáforo *turno* donde se demora hasta que deba trabajar, uno inicializado en 1 (el que inicia trabajando) y el otro en 0 (el que inicia dormido).

```
sem mutex = 1, pedidos = 0, espera[N] = ([N] 0), inicio = 0, turno[2] = (1, 0);  
int resultados[N]; cola C; bool finTiempo = false;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
  { --generar secuencia S  
    P(mutex);  
    push(C, (id, S));  
    V(mutex);  
    V(pedidos);  
    P(espera[id]);  
    --ver resultado de resultados[id]  
  }  
}
```

Process Servidor[id: 0..1]

```
{ secuencia sec; int aux;  
  while (true)  
  { P(turno[id]);  
    finTiempo = false;  
    V(inicio);  
    while (no termine el tiempo)  
    { P(pedidos);  
      P(mutex);  
      pop(C, (aux, sec));  
      V(mutex);  
      resultados[aux] = resolver(sec);  
      V(espera[aux]);  
    }  
  }  
}
```

¿Cómo sabe
cuando hasta
cuando iterar?

Process Reloj

```
{ while (true)  
  { P(inicio);  
    delay(5 hs);  
    finTiempo = true;  
    V(pedidos);  
  }  
}
```

EJERCICIO 5

Cuando pasa el $P(\text{pedidos})$ es porque el reloj avisó que termino el tiempo ($\text{finTiempo} = \text{true}$) y/o hay pedidos en la cola \rightarrow en base a eso despierta al otro o atiende pedido.

```
sem mutex = 1, pedidos = 0, espera[N] = ([N] 0), inicio = 0, turno[2] = (1, 0);  
int resultados[N]; cola C; bool finTiempo = false;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
  { --generar secuencia S  
    P(mutex); push(C, (id, S)); V(mutex);  
    V(pedidos);  
    P(espera[id]);  
    --ver resultado de resultados[id]  
  }  
}
```

Process Reloj

```
{ while (true)  
  { P(inicio); delay(5 hs); finTiempo = true;  
    V(pedidos);  
  }  
}
```

Process Servidor[id: 0..1]

```
{ secuencia sec; int aux; bool ok;  
  while (true)  
  { P(turno[id]); finTiempo = false; V(inicio);  
    ok = true;  
    while (ok)  
    { P(pedidos);  
      if (finTiempo) { ok = false;  
                       V(turno[1-id]);  
                       }  
      else { P(mutex); pop(C, (aux, sec)); V(mutex);  
             resultados[aux] = resolver(sec);  
             V(espera[aux]);  
           }  
    }  
  }  
}
```

Si termino el tiempo entonces marca la salida del *while* interno y despierta al otro servidor

EJERCICIO 5

```
sem mutex = 1, pedidos = 0, espera[N] = ([N] 0), inicio = 0, turno[2] = (1, 0);  
int resultados[N]; cola C; bool finTiempo = false;
```

Process Cliente[id: 0..N-1]

```
{ secuencia S;  
  while (true)  
  { --generar secuencia S  
    P(mutex);  
    push(C, (id, S));  
    V(mutex);  
    V(pedidos);  
    P(espera[id]);  
    --ver resultado de resultados[id]  
  }  
}
```

Process Reloj

```
{ while (true)  
  { P(inicio);  
    delay(5 hs);  
    finTiempo = true;  
    V(pedidos);  
  }  
}
```

Process Servidor[id: 0..1]

```
{ secuencia sec; int aux; bool ok;  
  while (true)  
  { P(turno[id]);  
    finTiempo = false;  
    V(inicio);  
    ok = true;  
    while (ok)  
    { P(pedidos);  
      if (finTiempo) { ok = false;  
                      V(turno[1-id]); }  
      else { P(mutex);  
             pop(C, (aux, sec));  
             V(mutex);  
             resultados[aux] = resolver(sec);  
             V(espera[aux]);  
           }  
    }  
  }  
}
```


EJERCICIO 6

En una montaña hay **30 escaladores** que en una parte de la subida deben utilizar un único paso de a uno a la vez y de acuerdo al orden de llegada al mismo.

En este caso sólo se deben usar los procesos que representes a los *escaladores*, y entre ellos administrarán el uso del Recurso Compartido (el paso).

```
Process Escalador[id: 0..29]
```

```
{ -- llega al paso
```

```
  -- Solicita acceso al paso
```

```
  // Usa el paso con Exclusión Mutua
```

```
  -- Libera el paso
```

```
}
```

Si está libre pasa y
sino debe esperar

Si hay alguien esperando le cede el
acceso al primero y sino lo libera



EJERCICIO 6

Usamos una cola para mantener el orden en que van llegando los escaladores. Si la cola está vacía el paso está libre, y sino debo esperar en esa cola.

```
cola c;  
sem espera[30] = ([30] 0);
```

Process Escalador[id: 0..29]

```
{ -- llega al paso  
  if (not empty(C))  
    { push (C, id);  
      P (espera[id]);  
    };  
  
  // Usa el paso con Exclusión Mutua  
  
  -- Libera el paso  
}
```

Siempre es falso. Por lo que usa
el paso sin Exclusión Mutua.

EJERCICIO 6

Que la cola esté vacía no implica que nadie la esté usando, sino que no hay nadie esperando. Se requiere tener el “estado” del paso en una variable booleana *libre*, y consultar por esa variable para saber si se puede acceder o hay que esperar.

```
cola c;  
sem espera[30] = ([30] 0);  
boolean libre = true;
```

Process Escalador[id: 0..29]

```
{ -- llega al paso  
  if (libre) libre = false  
  else { push (C, id);  
        P (espera[id]);  
        };
```

```
  // Usa el paso con Exclusión Mutua
```

```
  -- Libera el paso
```

```
}
```

Puede haber inconsistencia y más de uno encontrar el recurso libre a la vez.

EJERCICIO 6

Se debe proteger el uso de las variables compartidas *libre* y *C*, pero como ambas están relacionadas se deben usar protegidas por un mismo semáforo *mutex*.

```
cola c;  
sem mutex = 1, espera[30] = ([30] 0);  
boolean libre = true;
```

Process Escalador[id: 0..29]

```
{ -- llega al paso  
  P (mutex);  
  if (libre) libre = false  
  else { push (C, id);  
        P (espera[id]);  
      };  
  V (mutex);
```

// Usa el paso con Exclusión Mutua

-- Libera el paso

```
}
```

El que se
demore acá deja
la SC ocupada.

Liberar el Recurso
Compartido.

```
cola c;  
sem mutex = 1, espera[30] = ([30] 0);  
boolean libre = true;
```

Process Escalador[id: 0..29]

```
{ -- llega al paso  
  P (mutex);  
  if (libre) { libre = false;  
              V (mutex); }  
  else { push (C, id);  
        V (mutex);  
        P (espera[id]);  
      };
```

// Usa el paso con Exclusión Mutua

-- Libera el paso

```
}
```

EJERCICIO 6

Ahora se implementa la “liberación” del Recurso Compartido (el paso). Si hay alguien esperando se le pasa el control del RC y sino se libera.

No usa las variables compartidas con EM

```
cola c;
sem mutex = 1, espera[30] = ([30] 0);
boolean libre = true;

Process Escalador[id: 0..29]
{ int aux;
  -- llega al paso
  P (mutex);
  if (libre) { libre = false; V (mutex); }
  else { push (C, id); V (mutex);
        P (espera[id]);
      };
  // Usa el paso con Exclusión Mutua
  if (empty (C)) libre = true
  else { pop (C, aux);
        V (espera[aux]);
      };
}
```

EJERCICIO 6

Se debe proteger con el mismo semáforo que en el acceso al RC (*mutex*).

```
cola c;  
sem mutex = 1, espera[30] = ([30] 0);  
boolean libre = true;  
  
Process Escalador[id: 0..29]  
{ int aux;  
  -- llega al paso  
  P (mutex);  
  if (libre) { libre = false; V (mutex); }  
  else { push (C, id); V (mutex); P (espera[id]); };  
  // Usa el paso con Exclusión Mutua  
  P (mutex);  
  if (empty (C)) libre = true  
  else { pop (C, aux); V (espera[aux]); };  
  V (mutex);  
}
```

EJERCICIO 6

Diferencia con la solución usando “passing the baton”

```
cola c;  
sem mutex = 1, espera[30] = ([30] 0);  
boolean libre = true;
```

Process Escalador[id: 0..29]

```
{ int aux;  
  -- llega al paso  
  P (mutex);  
  if (libre) { libre = false; V (mutex); }  
  else { push (C, id);  
        V (mutex);  
        P (espera[id]); };  
  // Usa el paso con Exclusión Mutua  
  P (mutex);  
  if (empty (C)) libre = true  
  else { pop (C, aux); V (espera[aux]); };  
  V (mutex);  
}
```

```
cola c;  
sem mutex = 1, esp[30] = ([30] 0);  
boolean libre = true;
```

Process Escalador[id: 0..29]

```
{ int aux;  
  -- llega al paso  
  P (mutex);  
  if (not libre) { push (C, id);  
                  V (mutex); P (esp[id]); };  
  libre = false;  
  V (mutex);  
  // Usa el paso con Exclusión Mutua  
  P (mutex);  
  libre := true;  
  if (not empty (C)) { pop (C, aux);  
                      V (esp[aux]); }  
  else V(mutex);  
}
```

EJERCICIO 6

Solución del estilo del algoritmo
TICKET

```
sem mutex = 1, espera[30] = ([30] 0);  
int actual = 0, ticket = 0;
```

Process Escalador[id: 0..29]

```
{ int miTurno;  
  -- llega al paso  
  P (mutex);  
  miTurno = ticket;  
  ticket++;  
  V (mutex);  
  P (espera[miTurno]);  
  // Usa el paso con Exclusión Mutua  
  P (mutex);  
  actual++;  
  V (espera[actual]);  
  V (mutex);  
}
```

Y al primero en llegar (ticket 0)
quien lo despierta?

EJERCICIO 6

Inicializo la posición 0 de espera en 1 o lo controlo en el código:

```
sem mutex = 1, espera[30] = ([30] 0);  
int actual = 0, ticket = 0;
```

Process Escalador[id: 0..29]

```
{ int miTurno;  
  -- llega al paso  
  P (mutex);  
  miTurno = ticket;  
  ticket++;  
  if (miTurno == 0) V(espera[0]);  
  V (mutex);  
  P (espera[miTurno]);  
  //Usa el paso con Exclusión Mutua  
  P (mutex);  
  actual++;  
  V (espera[actual]);  
  V (mutex);  
}
```