Programación concurrente

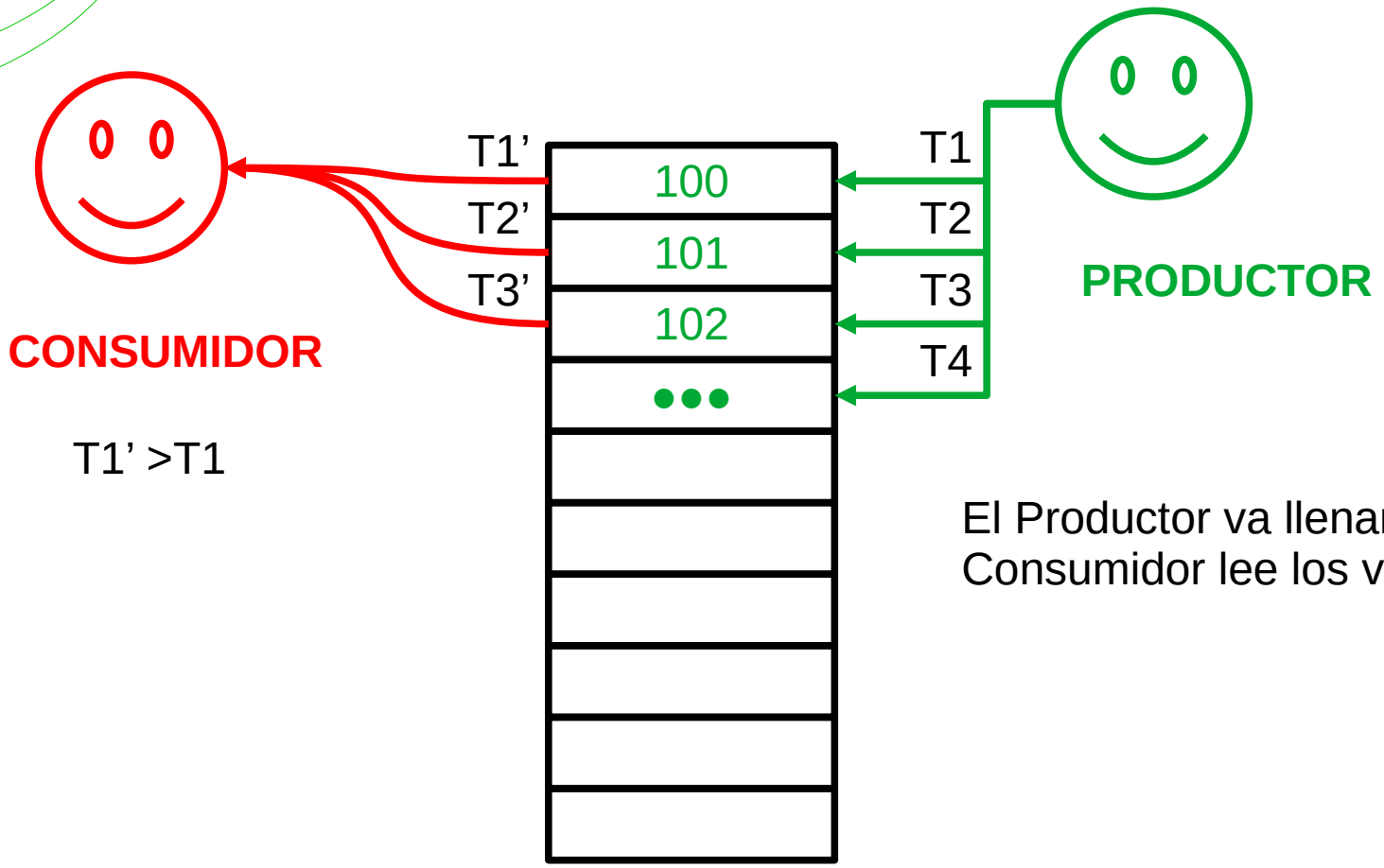
En esta clase:

- Programación Concurrente
- Condiciones de Carrera
- Región Crítica
- Operaciones Atómicas
- Semáforos
- Mutex
- Problemas comunes.

Programación concurrente

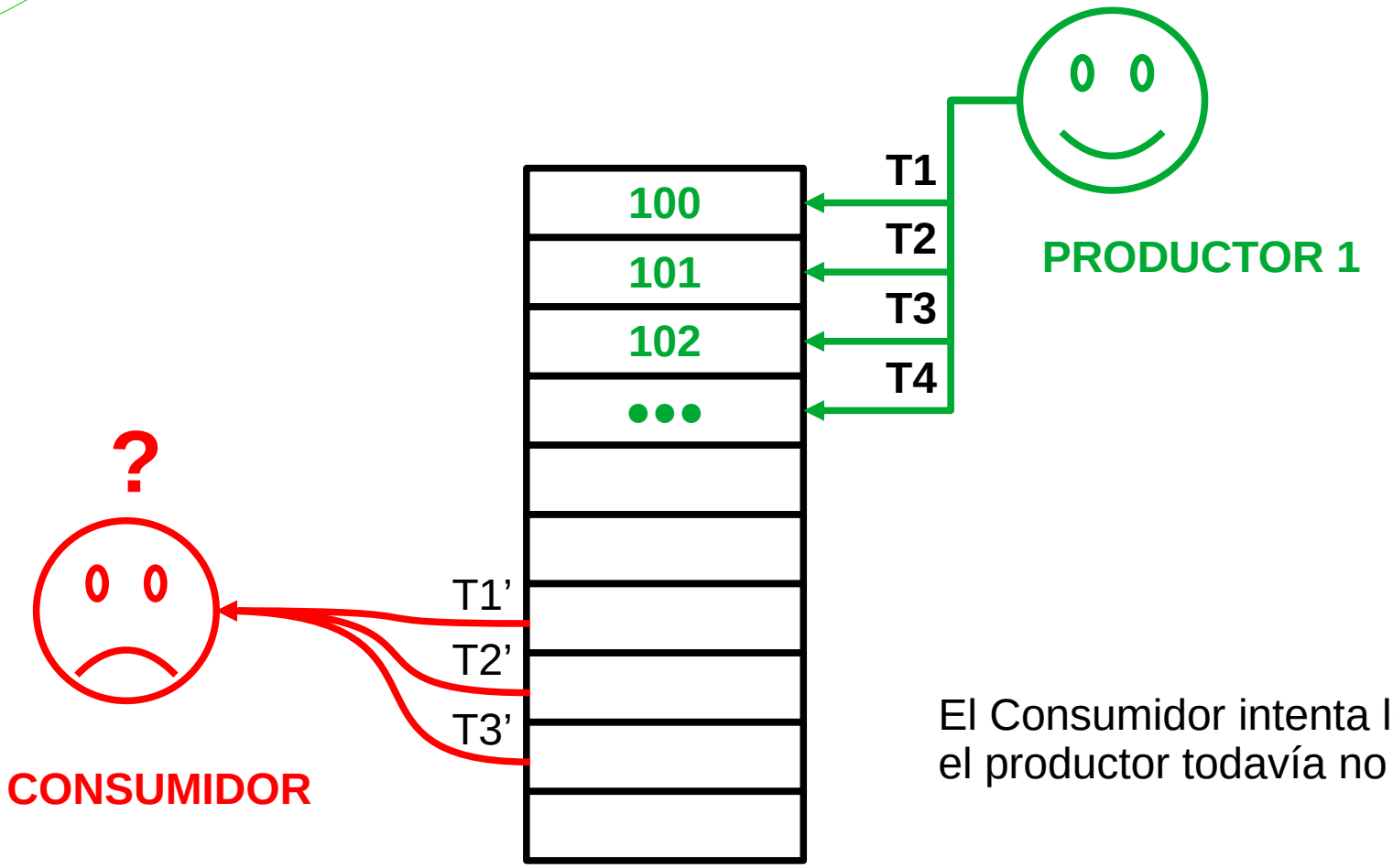
- Hasta ahora vimos como compartir variables (memoria) entre dos o más procesos
- Veamos algunos problemas que pueden suceder:
  - Si un proceso almacena datos en la memoria (Productor) y el otro los lee (Consumidor) Es necesario que el Consumidor no lea posiciones de memoria vacías, es decir que no consuma antes de que haya producción.
  - Si existen varios productores, puede darse que un proceso sobre escriba lo escrito por otro.

Un Productor y un Consumidor sincronizados



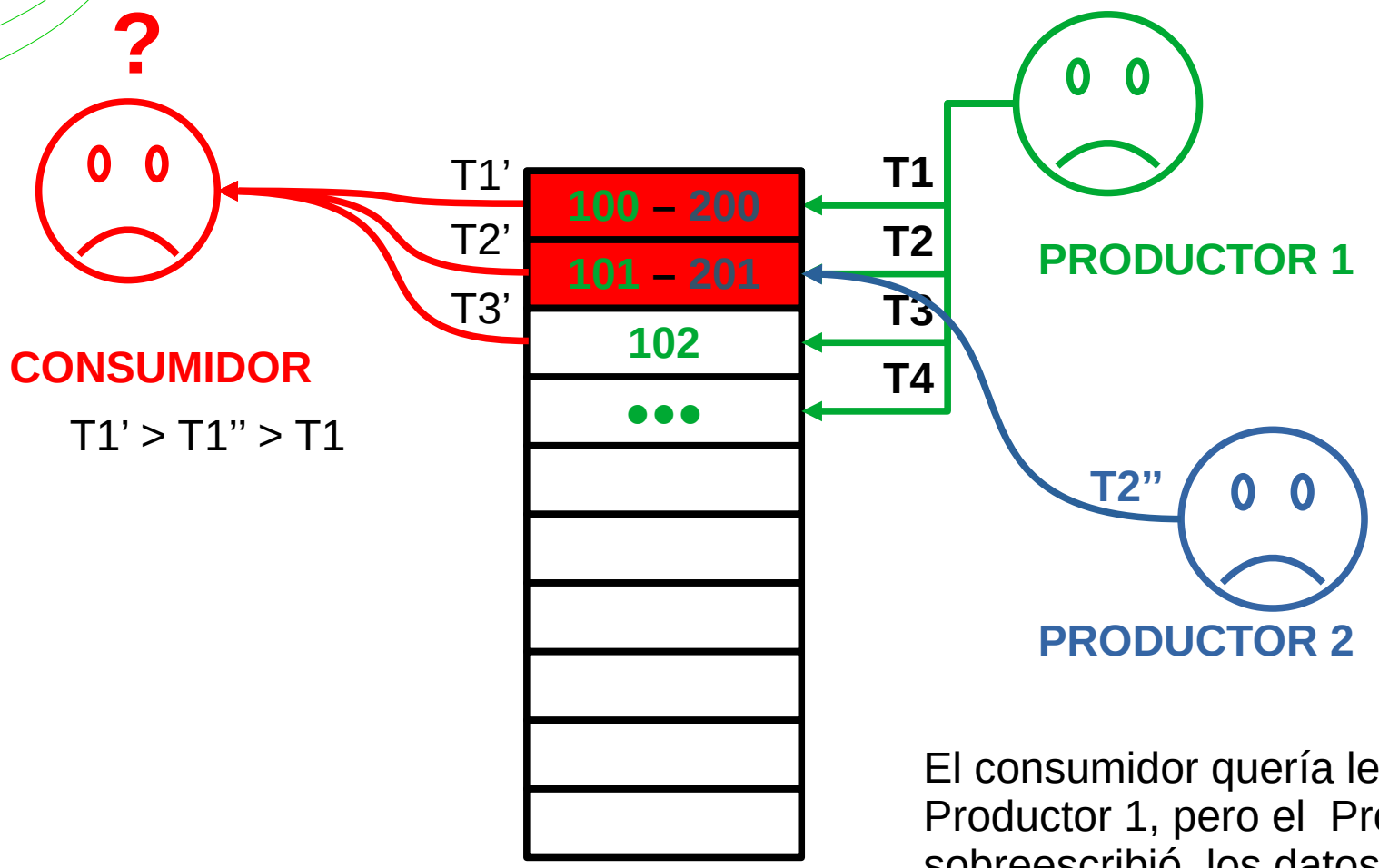
El Productor va llenando el buffer y el Consumidor lee los valores ya escritos.

Un Productor y un Consumidor no sincronizados



El Consumidor intenta leer valores que el productor todavía no produjo.

Dos Productores y un Consumidor



El consumidor quería leer los datos del Productor 1, pero el Productór 2 sobreescribió los datos, el Consumidor lee datos inconsistentes

Condiciones de Carrera

- El ejemplo de dos productores y un consumidor nos muestra una condición de carrera.
- Los dos productores intentan escribir en el mismo buffer, y sus datos se van a sobreescribir o en el mejor de los casos se van a mezclar.
- El resultado final que lee el consumidor depende del orden de llegada de los productores.
- El otro caso es que el Consumidor intente acceder a datos que todavía no fueron escritos.
- Esto se evitaría haciendo lo siguiente:
  - El Productor 1, se reserva el acceso al buffer impidiendo que lo accedan el Productor 2, o el Consumidor, hasta que termine de escribir todos sus datos.
  - Luego el Consumidor o el Productor 2, acceden y se reservan el acceso exclusivo hasta que terminen de realizar sus operaciones.
- Para realizar esto vamos a definir ***Región Critica***

Región Crítica

- Se denomina ***región crítica*** o ***sección crítica,*** a una porción de código de un programa en la que se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un proceso o hilo en ejecución a la vez.
- Al existir múltiples procesos que pueden acceder al mismo recurso, se da la posibilidad que el recurso no sea utilizado de la manera esperada.
- Otro ejemplo:
  - Dos procesos acceden a la misma impresora.
  - Cada proceso debería realizar la impresión completa antes de cederle el recurso a otro proceso. Si no se hace así las impresiones resultarían entremezcladas.
  - Para evitar esto, un proceso debería “bloquear” el acceso a otros procesos hasta que complete su uso.
  - El código que realiza la impresión es una ***“región crítica”***

Región Crítica

- Cada instrucción de lenguaje de alto nivel, (C, Python, Pascal, etc.) se traduce a una o varias instrucciones elementales ***(IE)*** del procesador.
- El sistema operativo puede interrumpir la ejecución de los procesos en cualquier momento, y se va a permitir finalizar la instrucción elemental que se este ejecutando en ese instante.
- Vemos por ejemplo la instrucción de lenguaje C

```
x=x+4; /*el valor original de x es 1*/
```

- Si lo pensamos, involucra varias instrucciones elementales
  - La primera trae el valor de x desde la memoria hasta el procesador y lo guarda en un registro A
  - La segunda realiza la operación de suma y la almacena en el registro A
  - La tercera almacena el contenido del registro A nuevamente en la memoria correspondiente a la variable x.

Región Crítica

- El sistema operativo puede interrumpir, en cualquier momento y no se va a completar la operación x=x+4; si no la instrucción elemental que se esté ejecutando.
- Si ahora otro proceso usa el valor x, lo que suceda va a depender del momento en que apareció la interrupción.
  1) Si aparece en la primera IE, x=1, El valor en el registro A es 1.
  2) Si aparece en la segunda IE, x=1. El valor en el registro A es 5.
  3) Si aparece en la tercera IE, x=5. El valor en el registro A es 5.
- El sistema operativo, va almacenar el contexto del proceso interrumpido, en este caso el registro A.

Región Crítica

- Supongamos que el nuevo proceso quiere realizar la operación x=x-1;
- Dependiendo del momento en que el SO interrumpió al proceso anterior, el nuevo valor de x será **0** o **4** considerando que la operación no es interrumpida para simplificar.
- Cuando retome el control el primer proceso, se restaura el registro A y se sigue desde ahi.
- El valor final de x será 5, sin importar que valor le haya asignado el segundo proceso.

Regiones Críticas y operaciones atómicas

- La asignación de una variable compartida es una región crítica.
- Se podría evitar si la asignación fuera una operación indivisible (atómica)
- Las Instrucciones Elementales son atómicas
- Si la operación compuesta de varias instrucciones elementales es una región crítica debo protegerla.

Protección de regiones críticas: Cerrojo o Mutex,

Proceso 1

```
while(cerrojo == 1)
{
;/* no hago nada*/
}
cerrojo = 1;← Aca interrumpe el SO
Instrucciones de Sección crítica
cerrojo = 0;
```

Proceso 2

```
while(cerrojo == 1)
{
;/* no hago nada*/
}
cerrojo = 1;
Instrucciones de Sección crítica
cerrojo = 0;
```

Se supone que `cerrojo` es una variable compartida entre ambos procesos y que comienza inicializada en 0. El SO puede pasar en cualquier momento de un proceso a otro. Veamos la situación “Normal”

- **Proceso 1** verifica la condición , como es falsa sale del `while` y asigna 1 a `cerrojo`, en ese momento el SO pasa el control al **Proceso 2**.
- El **Proceso 2** solo puede llegar hasta la instrucción `while` y no puede pasar de ahí, ya que `cerrojo == 1`. Queda en “espera activa".
- Cuando el SO vuelva a otorgarle la ejecución al Proceso 1, este va a cambiar el valor de `cerrojo` cuando finalice su sección critica.
- En ese momento, si el P**roceso 2** sigue ejecutándose, podrá salir del `while` y pasar a ejecutar su sección crítica, cambiando previamente el valor de `cerrojo` a “1” para bloquear el acceso a otros procesos.

Protección de regiones críticas: Cerrojo o Mutex,

Proceso 1

```
while(cerrojo == 1)
{
;/* no hago nada*/
} ← Aca interrumpe el SO
cerrojo = 1;
Instrucciones de Sección crítica
cerrojo = 0;
```

Proceso 2

```
while(cerrojo == 1)
{
;/* no hago nada*/
}
cerrojo = 1;
Instrucciones de Sección crítica → SO
cerrojo = 0;
```

Se supone que `cerrojo` es una variable compartida entre ambos procesos. El SO puede pasar en cualquier momento de un proceso a otro. Vemos otro caso

- **Proceso 1** verifica la condición que es falsa, sale del `while`, y en ese momento el SO pasa el control al **proceso 2**.
- El **proceso 2** también verifica la condición , sale del `while`, modifica `cerrojo` y comienza a ejecutar su sección critica.
- Si en ese momento el SO vuelve a otorgarle la ejecución al **Proceso 1**, este va comenzar ejecutando la instrucción en negro, ya que cuando ejecuto la ultima instrucción en verde, la condición era falsa.
- A partir de ahí ambos procesos están ejecutando la región crítica que es lo que queríamos evitar.

Protección de regiones críticas: Cerrojo o Mutex,

Proceso 1

```
int cerrojo = 0;
while(cerrojo == 1);
cerrojo = 1;
```

test & set (atómica)

```
Instrucciones de Sección crítica
cerrojo = 0;
```

Proceso 2

```
while(cerrojo == 1);
cerrojo = 1;
```

test & set (atómica)

```
Instrucciones de Sección crítica
cerrojo = 0;
```

El problema con lo que vimos es que el cambio de valor y la comprobación de la variable cerrojo están divididas, y el programa puede interrumpirse en el medio de la ejecución de ambas.

Para evitar esto muchos procesadores poseen una instrucción llamada **“test and set”** que realiza la comprobación del while y la asignación de la variable en una forma indivisible. (**instrucción atómica).**

Otro problema del mutex implementado de esta manera es que el proceso que no puede acceder a su región crítica, ocupa el tiempo que le asigna el SO solamente verificando que se haya liberado el cerrojo. Sería mejor que el SO operativo bloquee el proceso y lo saque de la cola de ejecución.

Para eso el SO debería intervenir en el proceso de protección de las regiones críticas.

Protección de regiones críticas: Semáforos.

Los semáforos son un tipo de datos que están compuestos por dos atributos: Un contador y una cola de procesos inicialmente vacía.
Y disponen de dos operaciones básicas que pasamos a describir en pseudocódigo:

```
down(semáforo s)
{
if (s.contador == 0)
   {
   añade proceso a
   s.cola_procesos;
   proceso pasa a estado
   bloqueado;
   }
else
   {
   s.contador--
   }
}
```

```
up(semáforo s)
{
if (hay procesos en
s.cola_procesos)
   {
   retira proceso de
   s.cola_procesos;
   proceso a estado preparado;
   }
else
   {
   s.contador++
   }
}
```

Protección de regiones críticas: Semáforos.

- Cuando hacemos la operación **`down(s)`** si el semáforo está “libre” **`(s.contador > 0)`**, entonces el semáforo se decrementa y el proceso sigue haciendo la tarea que haya a continuación.
- Pero si el semáforo estaba ocupado **`(s.contador == 0)`** entonces el proceso se añade a una cola de procesos en espera del semáforo y pasa al estado bloqueado.
- Cuando se realiza una operación **`up(s)`** sobre un semáforo en el que hay procesos en su cola resulta el primer proceso de la cola, es decir el lleva mas tiempo en ella, pase a estado “Listo para ejecutar”.
- Es un error frecuente pensar que una operación **`up(s)`** resulte en que el proceso retirado de la cola pase a estado activo. Recordar que las transiciones de estado activo a preparado y viceversa son siempre controladas por el planificador del sistema operativo.
- Las operaciones de verificación e incremento se implementan de forma atómica, para evitar los problemas ya vistos
- Las operaciones **`down(s)`** y **`up(s)`** también son conocidas como **`P(s)`** y **`V(s)`**, **`Wait(s)`** y **`Signal(s)`** o **`Tomar(s)`** y **`Soltar(s)`** respectivamente

Pthread_mutex

- pthread_mutex_t mutex1;
- pthread_mutex_init()
  - permite dar las condiciones iniciales a un candado mutex
- pthread_mutex_lock()
  - Permite solicitar acceso al mutex, el hilo se bloquea hasta su obtención
- pthread_mutex_trylock()
  - permite solicitar acceso al mutex, el hilo retorna inmediatamente. El valor retornado indica si otro hilo lo tiene.
- pthread_mutex_unlock()
  - Permite liberar un mutex.
- pthread_mutex_destroy()
  - Destruye la variable (de tipo pthread_mutex_t) usada para manejo de mutex.

Problemas Asociados a Semaforos y Mutex

- Desactivación Accidental.
- “ Deadlock” o Interbloqueo
  - Bloqueo recursivo
  - Interbloqueo por “muerte”
  - Enlace mortal” o “abrazo de la muerte”
- Inversión de Prioridad
- Uso de semáforo como una Señal

Problemas Asociados a Semaforos y Mutex

- Desactivación Accidental
  - Situación: Semáforo de conteo.
    - El valor del contador está en 0, o sea el semáforo debería bloquear cualquier proceso que lo intente utilizar
      Un proceso, por error, desactiva el semáforo cuando no debe.
    - El contador se incrementa,
    - El siguiente proceso no se bloquea y accede a la sección critica cuando no debe.

Problemas Asociados a Semaforos y Mutex

- Uso de semáforo como señal
  - Situación: Se usa el semáforo de una manera no habitual, se inicializa en 0 y una tarea utiliza P(s), bloquéandose a si misma.
  - Una tarea

Tarea 1

```
s.cont=0;

. . .

/*Wait*/

P(s);
//Tarea 1 se bloquea
```

Tarea 2

```
/*Signal*/

V(s);

//Habilito que Tarea 1
vuelva a estar lista para
ejecutar
```

Problemas Asociados a Semaforos y Mutex

- “ Deadlock” o Interbloqueo
  - Bloqueo recursivo
    - Una tarea llama más de una vez al mismo semáforo,
    - Suele pasar si una función se usa recursivamente y dentro de la función se llama al semáforo.
  - Interbloqueo por “muerte”
    - Una tarea termina sin liberar el semáforo que tomó.
    - Todas las demás tareas que lo usan, quedan bloqueadas.
    - Solución : timeout.
  - Enlace mortal” o “abrazo de la muerte”
    - Sucede cuando se necesitan dos recursos y se protegen con dos semáforos.

Problemas Asociados a Semaforos y Mutex

- “ Deadlock
  - Enlace mortal” o “abrazo de la muerte”
    - Sucede cuando se necesitan dos recursos y se protegen con dos semáforos.

Tarea 1

```
P(s1);
P(s2);
region critica 1
```

Tarea 2

```
P(s2);
P(s1);
region critica 2
```