Exclusión mutua

Gustavo Romero

Arquitectura y Tecnología de Computadores

22 de noviembre de 2010

Índice

Lecturas recomendadas

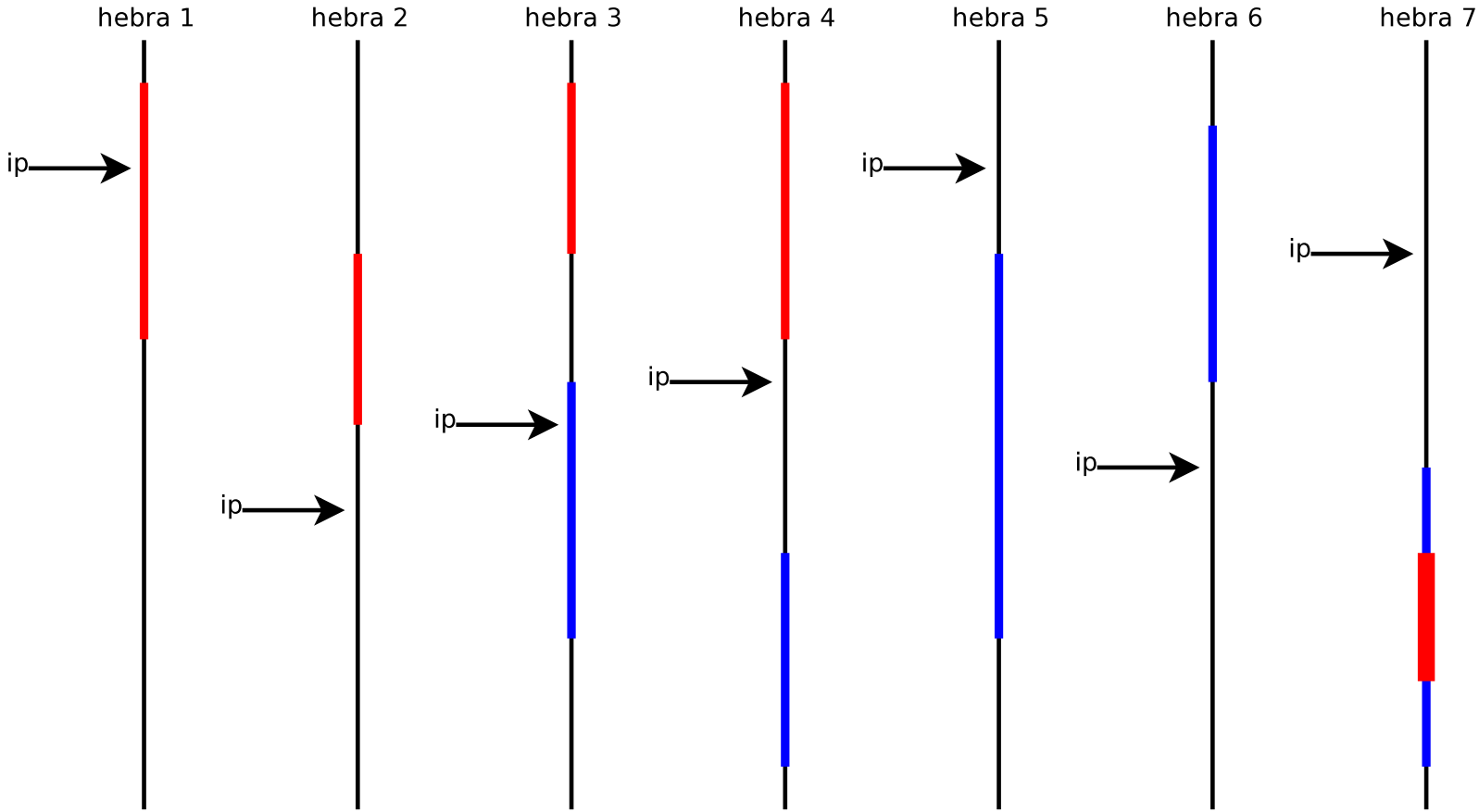
J.Bacon	Operating Systems (9, 10, 11)
A. Silberschatz	Fundamentos de Sistemas Operativos (4, 6, 7)
W. Stallings	Sistemas Operativos (5, 6)
A. Tanuenbaum	Sistemas Operativos Modernos (2)

Introducción

Sección crítica (1)

- Cuando una hebra/proceso accede a un recurso compartido diremos que está ejecutando una **sección crítica**.
- Una hebra puede tener **más de una** sección crítica.
- Las secciones críticas puede **anidarse**.
- La ejecución de una sección crítica debe ser **mutuamente excluyente**:
  - En cualquier instante de tiempo **sólo a una** hebra se le permite ejecutar su sección crítica.
  - Toda hebra debe **pedir permiso para entrar** en una sección crítica.
  - Al abandonar la sección crítica la hebra debe **asegurarse** de que otra hebra que lo necesite pueda entrar $\longrightarrow$ **avisar** de que ha **salido**.
  - Es necesario que toda hebra cumpla con este **protocolo**.

Sección crítica (2)



- Suponga que todas estas son hebras de un mismo proceso y que la hebra 1 ha conseguido entrar en la sección crítica roja.
- ¿Qué otros `IP` son válidos al mismo tiempo?

Uso de secciones críticas (1)

¿Cómo resolver el problema de las secciones críticas?

- Establecer un **protocolo** de **serialización** o **exclusión mutua**.
- El resultado de las hebras involucradas no dependerán del entrelazado no determinista de sus instrucciones.

Protocolo:

- El código que desee entrar en una sección_crítica debe solicitarlo mediante `entrar_sección_crítica`.
- Una vez ejecutada la sección crítica esta debe finalizarse con `salir_sección_crítica`.
- Al resto del código fuera de la sección crítica lo denominaremos `sección_no_crítica`.

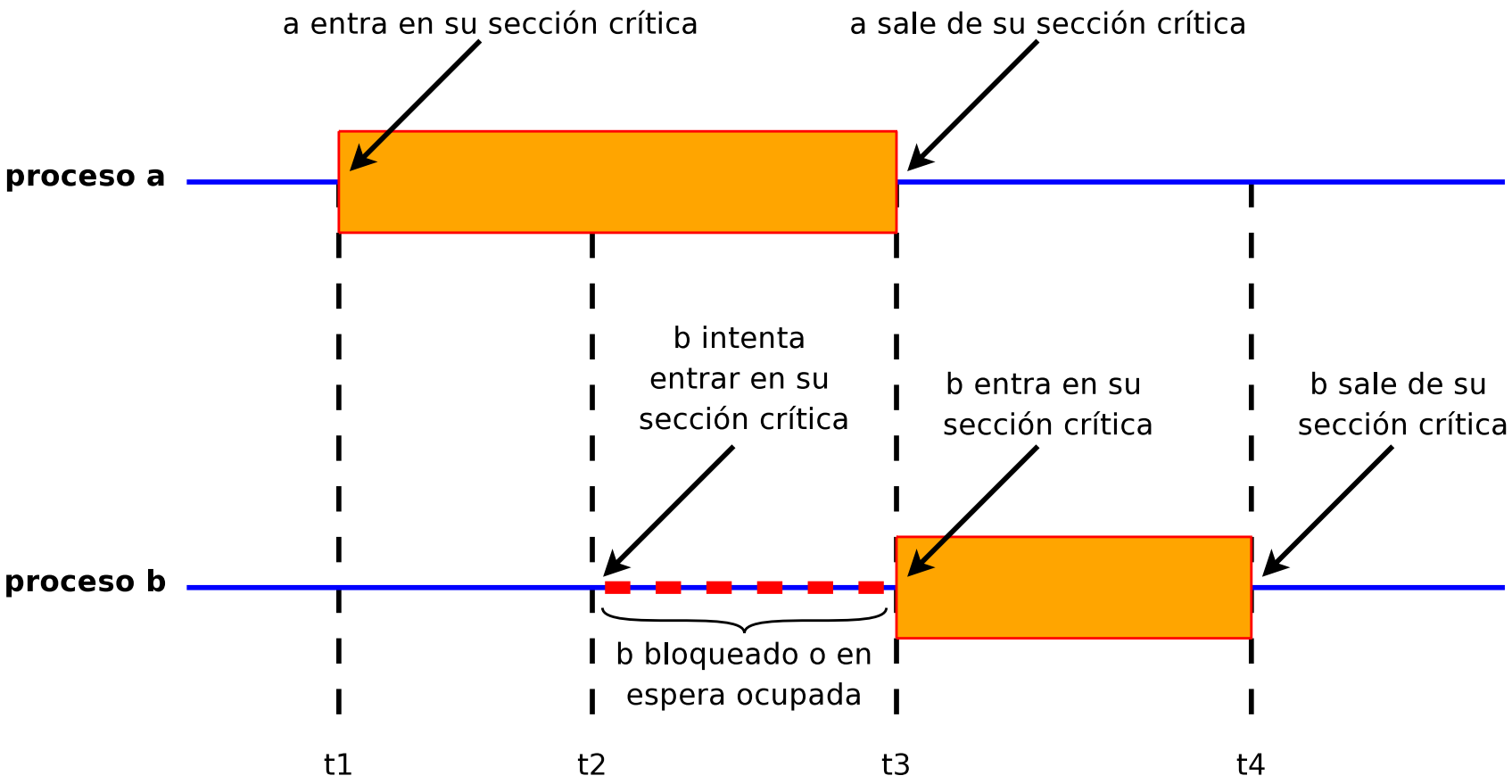
Uso de secciones críticas (2)

```
// patrón de comportamiento
// habitual de una hebra que
// ejecuta una sección crítica

while(true)
{
    entrar_sección_crítica
    sección_crítica
    salir_sección_crítica
    sección_no_crítica
}
```

- Cada hebra se ejecuta a una **velocidad no cero** pero no podemos hacer ninguna suposición acerca de su velocidad relativa de ejecución.
- Aunque dispongamos de más de un procesador la gestión de **memoria** previene el acceso simultáneo a la misma localización.

Protocolo de exclusión mutua



Ejecución de **secciones críticas** mediante **exclusión mutua**.

El problema de la exclusión mutua

1965
- Dijkstra define el problema.
- Primera solución por Dekker.
- Solución general por Dijkstra.

196x
- Soluciones por Fischer, Knuth, Lynch, Rabin, Rivest...

1974
- Algoritmo de la panadería de Lamport.

1981
- Algoritmo de Peterson.

2008
- Cientos de soluciones publicadas.
- Muchos problemas relacionados.

Soluciones para lograr exclusión mutua

Soluciones a nivel de **usuario**:

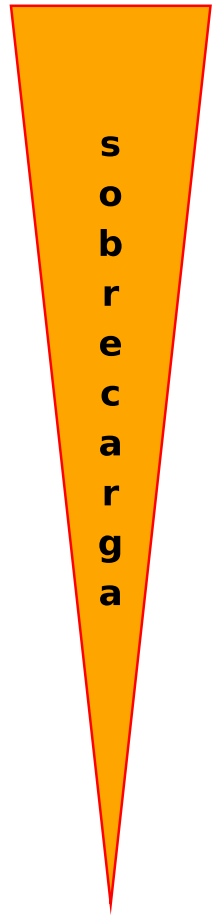
- Algoritmos no basados en el uso de instrucciones especiales del procesador ni de características del núcleo $\longrightarrow$ espera ocupada.

Soluciones **hardware**:

- Instrucciones especiales del procesador.

Soluciones a nivel del **núcleo**:

- Bajo nivel: cerrojos y semáforos binarios.
- Alto nivel: semáforos y monitores.



La solución adecuada dependerá del modelo de procesamiento.

Requisitos necesarios de una sección crítica (1)

1. Exclusividad:
   - Como máximo una hebra puede entrar en su sección crítica.
2. Progreso:
   - Ninguna hebra fuera de su sección crítica puede impedir que otra entre en la suya.
3. Espera acotada (no inanición):
   - El tiempo de espera ante una sección crítica debe ser limitado.
4. Portabilidad:
   - El funcionamiento correcto no puede basarse en...
     - la velocidad de ejecución.
     - el número o tipo de los procesadores.
     - la política de planificación del SO.
   - No todos los autores dan importancia a este requisito.

Requisitos necesarios de una sección crítica (2)

1. Exclusividad:
   - Si no se cumple la solución es **incorrecta**.
2. Progreso:
   - Los protocolos más sencillos a veces violan este requisito.
3. Espera acotada (no inanición):
   - La espera ocupada y las políticas de prioridad estáticas son típicos candidatos para violar este requisito.
4. Portabilidad:
   - Algunas soluciones dependen de...
     - el número o tipo de los procesadores.
     - la política de planificación del SO.

Propiedades **deseables** de las secciones críticas

- Eficiencia:
  - La sobrecarga de entrar y salir de la sección crítica debería ser pequeña en comparación con el trabajo útil realizado en su interior.
  - En general suele ser buena idea evitar la espera ocupada.
- Justicia:
  - Si varias hebras esperan para entrar en su sección crítica deberíamos permitir que todas entrasen con parecida frecuencia.
- Escalabilidad:
  - Debe funcionar igual de bien para cualquier número de hebras.
- Simplicidad:
  - Deben ser fáciles de usar $\longrightarrow$ el esfuerzo de programación debe ser lo más pequeño posible.

Usuario

Soluciones a nivel usuario

- La sincronización se consigue a través de **variables globales**.
- Estas soluciones funcionan mediante **espera ocupada**:

```
while(condición != cierto); // sólo salta
```

- Simplificación:
  - Empezaremos resolviendo el problema para 2 procesos ó 2 hebras: H_0 y H_1.
  - Cuando hablemos de más de dos hebras H_i y H_j denotarán a dos hebras diferentes ($i \neq j$).
- Exclusión mutua para 2 hebras:
  - Algoritmos de Dekker.
  - Algoritmo de Petterson.
- Exclusión mutua para más de 2 hebras:
  - Algoritmo de la panadería (Lamport).

Algoritmo 1: exclusión mutua mediante espera ocupada

variables compartidas

```
int turno = 0;
```

Hebra 0

```
while(true)
{
    while (turno != 0);
    sección_crítica();
    turno = 1;
    sección_no_crítica();
}
```

Hebra 1

```
while(true)
{
    while (turno != 1);
    sección_crítica();
    turno = 0;
    sección_no_crítica();
}
```

- Primera solución propuesta para el problema de la exclusión mutua en secciones críticas.
- Versión 1 del algoritmo de Dekker... ¡hay 5!.

Análisis del algoritmo 1: ¿funciona? (1)

variables compartidas

```
turno = 0; // compartida
```

Hebra i

```
while (true)
{
    while (turno != i);
    sección_crítica();
    turno = j;
    sección_no_crítica();
}
```

- Debemos inicializar la variable `turno`.
- La hebra H_i ejecuta su sección crítica si y sólo si `turno == i`.
- H_i se mantiene en **espera ocupada** mientras H_j está en su sección crítica $\longrightarrow$ **exclusividad satisfecha**.
- El requisito de **progreso no es satisfecho** porque estamos suponiendo la alternancia estricta de las 2 hebras.

progreso no satisfecho $\longrightarrow$ solución no válida

Análisis del algoritmo 1: ¿es correcto? (2)

- Supongamos SNC_0 larga y SNC_1 corta.
- Si `turno == 0`, H_0 entrará en su SC_0, la abandonará haciendo `turno = 1` y comenzará a ejecutar su larga SNC_0.
- Ahora H_1 entrará en su SC_1, la abandona haciendo `turno = 0` y ejecuta su corta SNC_1.
- Si de nuevo H_1 intenta entrar en su SC_1, debe esperar hasta que H_0 finalice su larga SNC_0 y pase de nuevo por SC_0.

Análisis del algoritmo 1

requisito	**cumple**	**motivo**
exclusión mutua	si	debido a turno sólo una hebra entra en su sección crítica
progreso	no	alternancia estricta, además el tamaño de la sección_no_crítica afecta a la otra hebra
espera acotada	si	espera igual al tiempo de ejecución de la otra sección crítica
portabilidad	no	1 procesador + prioridad estática = circulo vicioso
eficiencia	no	espera ocupada
escalabilidad	no	el problema se acentúa al aumentar el número de hebras

Analice cuidadosamente y compare su análisis con la bibliografía.

Algoritmo 2

¿Cuál es el principal problema del primer algoritmo?

- Almacenamos el identificador de hebra en la variable compartida usada para sincronizar y el identificador de cada hebra ha de establecerlo la otra.

¿Podemos hacerlo mejor?

- Hagamos que cada hebra sea responsable de modificar la variable compartida para permitirse el uso de la sección crítica.
- Ahora las hebras competirán por el uso de la sección crítica en lugar de cooperar.

Algoritmo 2

variables compartidas

```
bool bandera[2] = {false, false};
```

$hebra_i$

```
while (true)
{
    while (bandera[j]);
    bandera[i] = true;
    sección_crítica();
    bandera[i] = false;
    sección_no_crítica();
}
```

- Necesitamos una variable lógica por cada hebra.
- H_i muestra su deseo de entrar en su sección crítica haciendo `bandera[i] = true`
- No satisface el requisito de exclusión mutua.
- Satisface el requisito de progreso.

Análisis del algoritmo 2

requisito	**cumple**	**motivo**
exclusión mutua	no	condición de carrera en el protocolo de acceso a la sección crítica
progreso	si	
espera acotada	si	
portabilidad	no	
eficiencia	no	usa espera ocupada
escalabilidad	no	más hebras requiere recodificación

Analice cuidadosamente y compare su análisis con la bibliografía.

Algoritmo 3

variables compartidas

```
bool bandera[2] = {false, false};
```

$hebra_i$

```
while (true)
{
    bandera[i] = true;
    while (bandera[j]);
    sección_crítica();
    bandera[i] = false;
    sección_no_crítica();
}
```

- Necesitamos una variable lógica por cada hebra.
- H_i muestra su deseo de entrar en su sección crítica haciendo `bandera[i] = true`
- Satisface el requisito de exclusión mutua.
- Satisface el requisito de progreso.

Algoritmo 3: ¿es correcto?

Imaginemos la siguiente secuencia de ejecución:

- H_0 : `bandera[0] = true;`
- H_1 : `bandera[1] = true;`

¿Qué pasaría?

Ambas hebras esperarían para siempre y ninguna podría entrar en su sección crítica $\Longrightarrow$ **interbloqueo**.

Análisis del algoritmo 3

requisito	**cumple**	**motivo**
exclusión mutua	si	
progreso	si	
espera acotada	no	interbloqueo en el protocolo de entrada
portabilidad	no	
eficiencia	no	utiliza espera ocupada
escalabilidad	no	más hebras requiere recodificación

Analice cuidadosamente y compare su análisis con la bibliografía.

Algoritmo 4

¿Cuál es el principal problema del algoritmo 3?

- Cada hebra establece su estado sin preocuparse del estado de la otra hebra.
- Si las dos hebras insisten en entrar simultáneamente se produce un interbloqueo.

¿Podemos hacerlo mejor?

- Cada hebra activa su bandera indicando que desea entrar pero la desactiva si ve que la otra también quiere entrar.

Algoritmo 4

variables compartidas

```
bool bandera[2] = {false, false};
```

$hebra_i$

```
while (true)
{
   bandera[i] = true;
   while (bandera[j]) {
     bandera[i] = false;
     // retardo
     bandera[i] = true;
   };
   sección_crítica();
   bandera[i] = false;
   sección_no_crítica();
}
```

- bandera expresa el deseo de entrar en la sección crítica.
- H_i hace `bandera[i] = true` pero está dispuesta a dar marcha atrás.
- Satisface el requisito de exclusión mutua.
- Satisface el requisito de progreso.
- No satisface el requisito de espera acotada.

Análisis del algoritmo 4

requisito	**cumple**	**motivo**
exclusión mutua	si	
progreso	si	
espera acotada	no	
portabilidad	no	
eficiencia	no	utiliza espera ocupada
escalabilidad	no	empeora con el número de hebras y requiere recodificación

Analice cuidadosamente y compare su análisis con la bibliografía.

Algoritmo de Dekker

variables compartidas

```
bool bandera[2] = {false, false};
int turno = 0; // desempate
```

- bandera expresa la intención de entrar en la sección crítica.
- turno indica cuál es la hebra preferida en caso de empate.
- H_i hace bandera[i] = true pero está dispuesta a dar marcha atrás con la ayuda de turno.

$hebra_i$

```
while (true)
{
    bandera[i] = true;
    while (bandera[j]) {
      if (turno == j) {
        bandera[i] = false;
        while (turno == j);
        bandera[i] = true;
      }
    }
    sección_crítica();
    turno = j;
    bandera[i] = false;
    sección_no_crítica();
}
```

Análisis del algoritmo de Dekker (5)

requisito	**cumple**	**motivo**
exclusión mutua	si	
progreso	si	
espera acotada	si	
portabilidad	no	
eficiencia	no	utiliza espera ocupada
escalabilidad	no	empeora con el número de hebras y requiere recodificación

Analice cuidadosamente y compare su análisis con la bibliografía.

Algoritmo de Peterson

variables compartidas

```
bool bandera[2] = {false, false};
int turno = 0; // desempate
```

hebra$_i$

```
while (true)
{
  bandera[i] = true;
  turno = j;
  while (bandera[j] && turno == j);
  sección_crítica();
  bandera[i] = false;
  sección_no_crítica();
}
```

- La hebra *i* muestra su deseo de entrar en su sección crítica haciendo `bandera[i] = true`.
- Si ambas hebras intentan entrar a la vez en su sección crítica la variable `turno` decide quien va a pasar.
- En comparación con el algoritmo de Dekker es más eficiente, más simple y extensible a más de 2 hebras.

Algoritmo de Peterson: ¿es correcto?

Exclusión mutua:

- H_0 y H_1 sólo pueden alcanzar su sección crítica si `bandera[0] == bandera[1] == true` y sólo si `turno == i` para cada hebra H_i, lo que es imposible.

Progreso y espera acotada:

- H_i podría evitar que H_j entre en su sección crítica sólo si está atrapado en `while (bandera[j] && turno == j);`.
- Si H_j no está intentando entrar en su sección crítica `bandera[j] == false` y H_i puede entrar en su sección crítica.
- Si H_j ha hecho `bandera[j] == true` y está atrapado en el bucle `while` entonces `turno == i` o `turno = j`.
  - Si `turno == i` H_i podrá entrar en su sección crítica.
  - Si `turno == j` H_j entrará en su sección crítica y hará `bandera[j] = false` a la salida y permitirá a H_i entrar.
- Si a H_j le da tiempo a hacer `bandera[j] = true`, hará `turno = i`.
- Como H_i no cambia el valor de `turno` mientras está esperando en el `while`, H_i podrá entrar en su sección crítica tras como mucho una ejecución de la sección crítica de H_j.

Análisis del algoritmo de Peterson

requisito	**cumple**	**motivo**
exclusión mutua	si	
progreso	si	
espera acotada	si	
portabilidad	no	
eficiencia	no	espera ocupada
escalabilidad	si	pero requiere recodificación

Analice cuidadosamente y compare su análisis con la bibliografía.

El problema de las hebras defectuosas

- Si una hebra falla fuera de su sección crítica no afectará a otra:
  - Si una solución cumple los 3 requisitos (exclusión mutua, progreso y espera acotada) entonces proporcionará robustez frente a fallos en el exterior de la sección crítica.
- Sin embargo, ninguna solución proporcionará robustez frente a una hebra que falle en su sección crítica... ¿Por qué?
  - Una hebra que falle en su sección crítica no podrá ejecutar `salir_de_sección_crítica()` con lo que impedirá a cualquier otra hebra entrar en su sección crítica porque no podrán finalizar `entrar_en_sección_crítica()`.
- ¿Qué puede hacerse en este caso?
- ¿Deberíamos hacer algo?

El algoritmo de la panadería de Lamport (1)

- Antes de entrar en su sección crítica cada hebra recibe un número.
- El propietario del menor número entra en su sección crítica.
- En caso de que dos hebras H_i y H_j reciban el mismo número si $i < j$ entonces H_i puede entrar primero.
- H_i pone su número a 0 al abandonar la sección crítica.
- Notación: $(a, b) < (c, d)$ si $(a < c)$ o $((a = c)$ y $(b < d))$.
- La corrección se basa en que si H_i está en su sección crítica y H_k ha escogido su número $(\neq 0)$, entonces $(numero[i], i) < (numero[k], k)$.

variables compartidas

```
const int N; // número de hebras
bool escoger[N] = {false, false,... false};
int número[N] = {0, 0,... 0};
```

El algoritmo de la panadería de Lamport (2)

hebra$_i$

```
while (true)
{
  escoger[i] = true;
  número[i] = max(número[0],...número[N - 1]) + 1;
  escoger[i] = false;
  for (int j = 0; j < N; ++j)
  {
    while (escoger[j]);
    while (número[j] != 0 && (número[j],j) < (número[i], i));
  }
  sección_crítica();
  número[i] = 0;
  sección_no_crítica();
}
```

Inconvenientes de las soluciones a nivel de usuario

- Las hebras intentando entrar en una **sección crítica bloqueada** se mantienen en **espera ocupada** $\Longrightarrow$ se desperdicia tiempo del procesador.
- Si la sección crítica tarda mucho tiempo en ejecutarse puede ser más **eficiente** bloquear a la hebra.
- Con un único procesador y una política de prioridad estática, la espera ocupada puede llevar a la **inanición**.

Hardware

Primera solución: un simple cerrojo (1)

- Un `cerrojo` es un **objeto en memoria principal** sobre el que podemos realizar dos operaciones:
  - `adquirir()`: antes de entrar en la sección crítica.
    - Puede requerir algún tiempo de espera a la entrada de la sección crítica.
  - `liberar()`: tras abandonar la sección crítica.
    - Permite a otra hebra acceder a la sección crítica.
- Tras cada `adquirir()` debe haber un `liberar()`:
  - Entre `adquirir()` y `liberar()` la hebra **posee** el cerrojo.
  - `adquirir()` sólo retorna cuando el llamador se ha convertido en el dueño del cerrojo... ¿Por qué?
- ¿Qué podría suceder si existieran llamadas a `adquirir()` sin su correspondiente `liberar()` emparejado?
- ¿Qué pasa si el propietario de un cerrojo trata de adquirirlo por segunda vez?

Usando un cerrojo (1)

retirar fondos de una cuenta bancaria

```
while (true)
{
    cerrojo.adquirir();
    saldo = conseguir_saldo(cuenta);
    saldo = saldo - cantidad;
    almacenar_saldo(cuenta, saldo);
    cerrojo.liberar();
    return saldo;
}
```

Usando un cerrojo (2)

Hebra 1

```
cerrojo.adquirir();
```

```
saldo = conseguir_saldo(cuenta);
saldo = saldo - cantidad;
```

```
almacenar_saldo(cuenta, saldo);
```

```
cerrojo.liberar();
```

```
return saldo;
```

Hebra 2

```
cerrojo.adquirir();
```

```
saldo = conseguir_saldo(cuenta);
saldo = saldo - cantidad;
almacenar_saldo(cuenta, saldo);
```

```
cerrojo.liberar();
```

```
return saldo;
```

Implementación de un cerrojo "giratorio" ("*spinlock*") (1)

```
class cerrojo {
public:
  cerrojo(): cerrado(false) {} // inicialmente abierto
  void adquirir()
    {
      while (cerrado); // espera ocupada [1]
      cerrado = true;  // cerrar
    }
  void liberar()
    {
      cerrado = false; // abrir
    }
private:
  volatile bool cerrado;
};
```

[1] condición de carrera

Implementación de un cerrojo "giratorio" ("*spinlock*") (2)

Problema:

- Las operaciones `adquirir()` y `liberar()` tienen en su interior **secciones críticas**.
- Estas operaciones deben ser **atómicas**.

Aparece un **dilema** de difícil solución:

- Hemos creado los cerrojos para implementar un protocolo de exclusión mutua que resolviese el problemas de las secciones críticas pero la solución contiene otro problema de sección crítica.
- ¿Qué podemos hacer?
  - Utilizar algún algoritmo que solucione el problema (panadería).
  - Utilizar la arquitectura del procesador para poner fin a este problema recursivo.

Implementación de operaciones atómicas

El soporte hardware puede facilitar la tarea:

- **Deshabilitar las interrupciones**: `cli/sti`
  - ¿Por qué funciona?
  - ¿Por qué puede esta acción evitar el cambio de hebra?
  - ¿Funcionaría en cualquier tipo de sistema? → **no**, válido sólo en sistemas con un **único procesador**.

- **Instrucciones atómicas**:
  - Existen instrucciones para las que el **procesador** y el **bus** garantizan su ejecución atómica.

`TAS`	Test And Set:	`lock xchg%al,(%edx)`
`FAA`	Fetch And Add:	`lock xadd%eax,(%edx)`
`CAS`	Compare And Swap:	`lock cmpxchg%ecx,(%edx)`
`LL/SC`	Load Linked/Store Conditional:	`ldrex/strex`

Bloqueo mediante interrupciones

Mecanismo primitivo y sólo válido en sistemas monoprocesador.

- No cumple con el requisito de la portabilidad.
- Deshabilitar las interrupciones en un procesador no evita que en otro se ejecute la sección crítica.
- Sólo en el caso de secciones críticas **extremadamente cortas** dentro del **núcleo** y en sistemas **monoprocesador** será una solución válida.
- Si no podemos atender la interrupción del reloj no podremos cambiar de hebra tal y como hacen la mayoría de los sistemas de tiempo compartido.

Protocolo:

- Antes de que una hebra entre en su sección crítica deshabilita las interrupciones (`cli`).
- Al salir de la sección crítica se han de volver a habilitar las interrupciones (`sti`).

Deshabilitar las interrupciones a nivel de usuario (1)

monoprocesador

hebra¡

```
while(true)
{
  deshabilitar_interrupciones();
  sección_crítica();
  habilitar_interrupciones();
  sección_no_crítica();
}
```

Se preserva la exclusión mutua pero se degrada la eficiencia del sistema: mientras está en la sección crítica no se atenderán interrupciones.

- No habrá tiempo compartido.
- El retraso en la atención de las interrupciones podría afectar al sistema completo. $\Longleftrightarrow$ ¿Por qué?
- Las aplicaciones podrían abusar o contener fallos $\Longrightarrow$ cuelgue del sistema.

Deshabilitar las interrupciones a nivel de usuario (2)

multiprocesador

Multiprocesador:

- No es efectivo en absoluto porque...
  - Aunque las deshabilitásemos en todos los procesadores no arregla nada porque...
  - Varios procesadores podría acceder a la vez a un mismo recurso $\longrightarrow$ no garantiza la exclusión mutua.

En resumen:

- Los efectos laterales son inaceptables en general.
- La buena noticia es que esta operación es **privilegiada** :)

Deshabilitar las interrupciones a nivel del núcleo (1)

variable compartida

```
cerrojo_t cerrojo;
```

hebra_i

```
while(true)
{
  cerrojo.adquirir();
  sección_crítica();
  cerrojo.liberar();
  sección_no_crítica();
}
```

- Vamos a crear un objeto **cerrojo** que permita implementar el protocolo de exclusión mutua necesario para las secciones críticas.
- Protocolo:
  - `adquirir()`: la primera hebra que ejecuta este método puede continuar ejecutando su sección crítica. Las siguientes deben esperar a que la primera la abandone.
  - `liberar()`: la hebra que abandona la sección crítica lo ejecuta para comunicar que queda libre y dejar pasar a otra hebra.

Deshabilitar las interrupciones a nivel del núcleo (2)

```
class cerrojo {
public:
  cerrojo(): cerrado(false) {}
  void adquirir() {
    deshabilitar_interrupciones();
*   while (cerrado);
    cerrado = true;
    habilitar_interrupciones();
  }
  void liberar() {
    deshabilitar_interrupciones();
    cerrado = false;
    habilitar_interrupciones();
  }
private:
  volatile bool cerrado;
};
```

Permite implementar **cerrojos** (*"spin lock"*) de forma atómica:

- ¿Existe alguna condición de carrera? $\longrightarrow$ no.
- ¿Utilizarías esta solución para resolver un problema de sección crítica? $\longrightarrow$ no.
- ¿Cuál es su principal desventaja? $\rightarrow$ espera ocupada con **efectos laterales** sobre el sistema.
- ¿Es capaz de detectar un **grave error**? *

Deshabilitar las interrupciones a nivel del núcleo (3)

```
class cerrojo
{
private:
  volatile bool cerrado;
public:
  cerrojo(): cerrado(false) {}
  void adquirir()
  {
    deshabilitar_interrupciones();
    while (cerrado)
    {
      habilitar_interrupciones();
      deshabilitar_interrupciones();
    }
    cerrado = true;
    habilitar_interrupciones();
  }
  void liberar()
  {
    deshabilitar_interrupciones();
    cerrado = false;
    habilitar_interrupciones();
  }
};
```

- Con esta solución es posible el cambio de hebra.
- Todavía no es una solución válida porque no es portable y depende del número de procesadores.

Instrucciones atómicas

- Lectura/modificación/escritura sobre una posición de memoria:
  - `lock`[2] `xchg %eax,(%edx)`: intercambia de forma atómica el valor del registro con la dirección de memoria.
  - `lock xadd %eax,(%edx)`: intercambia y suma, `%eax = (%edx)`, `(%edx) = %eax + (%edx)`.
  - `lock cmpxchg %ecx, (%edx)`: compara el acumulador con `(%edx)`. Si son iguales activa `ZF = 1` y hace `(%edx) = %ecx`. En otro caso limpia `ZF = 0` y lleva `(%edx)` al acumulador.
- Carga enlazada y almacenamiento condicional:
  - `ldrex r0,[mem]`: carga el contenido de la dirección de memoria `mem` y el procesador se vuelve sensible a dicha dirección de memoria.
  - `strex [mem],r0`: el almacenamiento fallará si otro procesador ha modificado el contenido de `[mem]`.

[2] prefijo de bloqueo del bus, necesario con más de 1 procesador.

La instrucción "Test And Set"

- La instrucción TAS se ejecuta siempre de forma atómica independientemente del número de cachés que se tengan.

Test And Set (v1)

```
bool test_and_set(bool *cerrojo)
{
  bool viejo = *cerrojo;
  *cerrojo = true;
  return viejo;
}
```

Test And Set (v2)

```
bool test_and_set(int *c)
{
  if (*c == 0)
  {
    *c = 1;
    return true;
  }
  return false;
}
```

- cerrojo es un parámetro tanto de entrada como de salida.

La instrucción "Compare And Swap"

Compare And Swap (v1)

```
bool compare_and_swap(long *p, long viejo, long nuevo)
{
  if (*p == viejo)
  {
    *p = nuevo;
    return true;
  }
  return false;
}
```

Compare And Swap (v2)

```
long compare_and_swap(long *p, long viejo, long nuevo)
{
  long valor = *p;
  if (*p == viejo)
  {
    *p = nuevo;
  }
  return valor;
}
```

Ejemplo de implementación: i386/{futex,cmpxchg}.h

```
long int testandset (int *spinlock)
{
  long int ret;
  __asm__ __volatile__("xchgl %0, %1"
                       : "=r"(ret), "=m"(*spinlock)
                       : "0"(1), "m"(*spinlock)
                       : "memory");
  return ret;
}

int compare_and_swap (long int *p, long int oldval, long int newval)
{
  char ret;
  long int readval;
  __asm__ __volatile__ ("lock; cmpxchgl %3, %1; sete %0"
                        : "=q" (ret), "=m" (*p), "=a" (readval)
                        : "r" (newval), "m" (*p), "a" (oldval)
                        : "memory");
  return ret;
}
```

Implementación de un cerrojo mediante TAS

```
class cerrojo
{
public:
 cerrojo(): ocupado(false)

adquirir()
{
  while (test_and_set(&ocupado));
}

liberar()
{
  ocupado = false;
}

private:
 volatile bool ocupado;
};
```

- ¿Existe todavía alguna condición de carrera? → no.
- ¿Es portable? → no.
- ¿Es eficiente? → no.
- ¿Qué sucede cuando muchas hebras intentan adquirir el cerrojo? → baja la tasa de aprovechamiento del procesador debido a la espera ocupada.

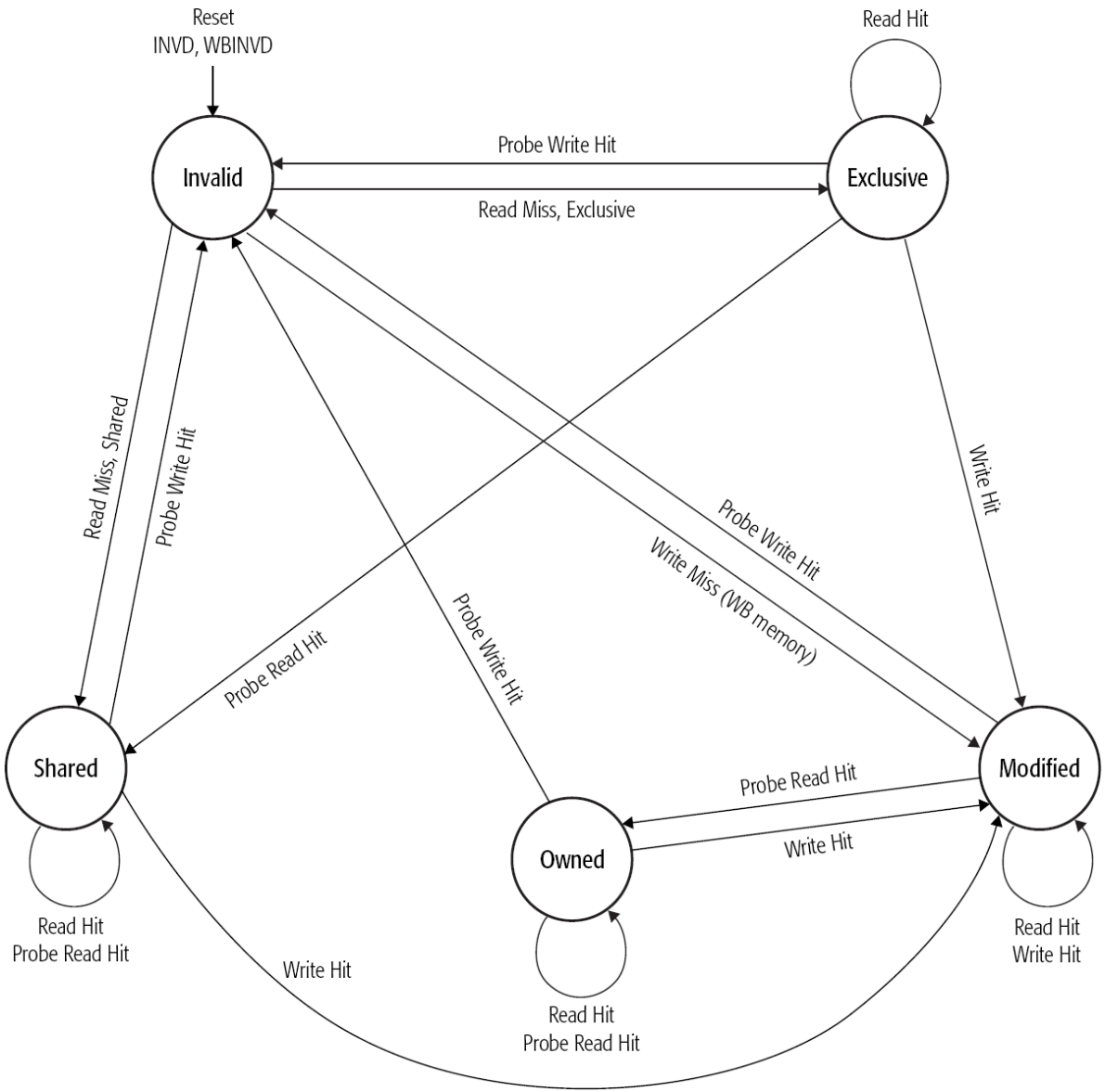
Análisis del cerrojo (1)

- La exclusión mutua se preserva.
- Si H_i entra en su sección crítica las demás hebras H_j realizan **espera ocupada** $\Longrightarrow$ **problema de eficiencia**.
- Cuando H_i sale de su sección crítica, la elección de la siguiente hebra H_j es arbitraria lo que viola el requisito de **espera acotada** $\Longrightarrow$ posible **inanición**.
- Algunos procesadores (como el Pentium) proporcionan instrucciones atómicas como `xchg` o `cmpxchg` que sufren los mismo inconvenientes que `TAS`.
- Sin embargo, ¿Cuál es el mayor problema con este tipo de cerrojos?

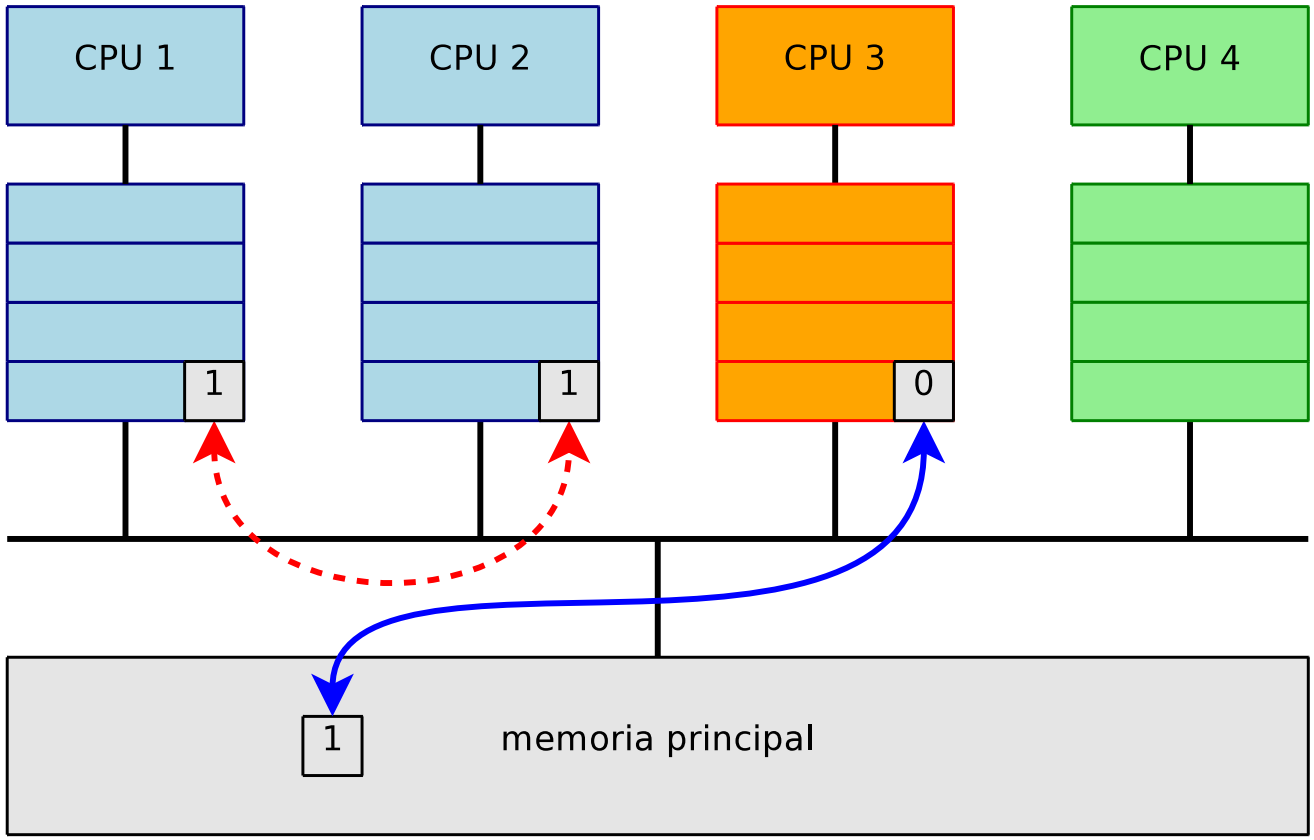
Análisis del cerrojo (2)

- Repetidas llamadas a `test_and_set()` pueden **monopolizar el bus del sistema** afectando a otras tareas independientemente de si están relacionadas con la sección crítica o no.
- ¿Qué pasará con la coherencia de caché? → garantizada.
  - `write-through`: Intel 486.
  - `write-back`: MESI(pentium), MOESI(k8), MESIF(nehalem). **M**odified/**O**wned/**E**xclusive/**S**hared/**I**nvalid/**F**orwarding.
- Además hay un **grave peligro** de otra clase de **inanición** en sistemas con un sólo procesador. Recordar lo que pasaba al utilizar espera ocupada a nivel de usuario y existían diferentes tamaños de `sección_no_crítica`.

Protocolos de coherencia de caché MOESI



Efecto ping-pong



- El efecto ping-pong entre la caché 1 y la caché 2 monopoliza, y desperdicia, el uso del bus del sistema.
- Es necesario implementar cerrojos más eficientes.

Conclusiones sobre TAS

Ventajas:

- El número de hebras involucradas en la sección crítica no está limitado.
- Puede utilizarse para controlar varias secciones críticas sin más que utilizar un cerrojo diferente para cada una ellas.
- Solución válida en sistemas multiprocesador.
- Solución simple y fácil de comprender.

Inconvenientes:

- Las espera ocupada es ineficiente.
- Una proceso o hebra tipo núcleo puede padecer inanición si tiene baja prioridad.
- Susceptible a interbloqueo e inversión de prioridad en secciones críticas anidadas.

La instrucción de intercambio

- Disponible en la inmensa mayoría de los procesadores.
- ¿Permite la creación de un protocolo de sección crítica?

$$\Downarrow$$

- ¿Soluciona el problema de la portabilidad? $\longrightarrow$ si.

instrucción de intercambio

```
void intercambiar(bool *a, bool *b)
{
  bool t = *a;
  *a = *b;
  *b = *t;
}
```

Núcleo

Semáforos con contador

Valor de un semáforo con contador:

positivo: máximo número de hebras que pueden entrar en una sección crítica. Si la exclusión mutua es necesaria ≤ 1.

negativo: número de hebras que han intentado entrar en su sección crítica y han sido colocadas en cola de espera.

cero: ninguna hebra está esperando para entrar y dentro de la sección crítica hay tantas hebras como el valor al que se inicializó el semáforo.

¿Cómo conseguir que las operaciones sobre el semáforo sean **atómicas**?

Semáforos con contador: implementación

```
class semáforo
{
public:                    // ¿0?
  semáforo(int _valor = 1):
    valor(_valor),
    bloq(vacía)
```

```
void p() // esperar()
 {
   valor = valor - 1;
   if (valor < 0)
   {
     bloq.meter(esta);
     bloquear(esta);
   }
 }
```

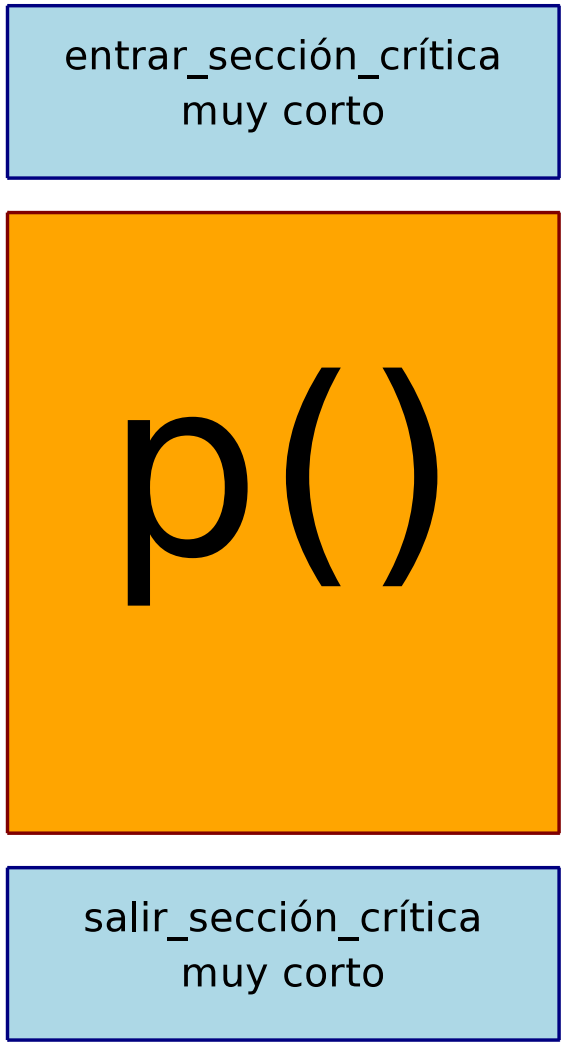
```
void v() // señalar()
 {
   valor = valor + 1;
   if (valor <= 0)
   {
     desbloquear(bloq.sacar());
   }
 }
```

```
private:
  int valor;
  lista<hebra> bloq;
};
```

Operaciones atómicas sobre semáforos (1)

Problema: `p()` y `v()` constan de múltiples instrucciones máquina que deben ejecutarse de forma atómica.

Solución: Utilizar otro tipo de secciones críticas con la esperanza de que tenga tiempos de ejecución más cortos y que permitan la atomicidad y exclusividad de las operaciones sobre el semáforo.



Operaciones atómicas sobre semáforos (2)

- Son deseables secciones críticas cortas al implementar `p()` y `v()` atómicos y mutuamente excluyentes.
- Posibles soluciones (ya vistas pero con renovada validez en este nuevo contexto):

  Monoprocesador:
  - **Deshabilitar las interrupciones** mientras se ejecutan las operaciones sobre semáforos.
  - Contradice la recomendación general de no manipular las interrupciones desde el nivel de usuario.
  - Solución válida porque sabemos que se van a deshabilitar durante **muy poco tiempo** $\Longleftrightarrow$ conocemos el tamaño de la sección crítica.

  Multiprocesador:
  - Emplear **instrucciones atómicas**: `TAS`, `CAS`, `LL/SC`,...

Semáforos con contador: implementación monoprocesador

```
class semáforo {
public:
  semáforo(int _valor = 0):
    valor(_valor), bloq(vacía) {}
  void p() // esperar()
  {
    deshabilitar_interrupciones();
    valor = valor - 1;
    if (valor < 0)
    {
      bloq.meter(esta);
      bloquear'(esta); [3]
    }
    habilitar_interrupciones();
  }
  void v() // señalar()
  {
    deshabilitar_interrupciones();
    valor = valor + 1;
    if (valor <= 0)
    {
      desbloquear'(bloq.sacar()); [4]
    }
    habilitar_interrupciones();
  }
private:
  volatile int valor;
  volatile lista<hebra> bloq;
};
```

¿Qué sucede al cambiar a otra hebra? ¿Siguen las interrupciones deshabilitadas?

[3] `bloquear'() = bloquear() + sti`

[4] `desbloquear'() = cli + desbloquear()`

Semáforos con contador: implementación multiprocesador

```
class semáforo {
public:
  semáforo(int _valor = 0):
    ocupado(false),
    valor(_valor),
    bloq(vacía) {}

  void p() // esperar()
  {
    while(TAS(ocupado) == true);

    valor = valor - 1;
    if (valor < 0)
    {
      bloq.meter(esta);
      bloquear'(esta); [5]
    }

    ocupado = false;
  }

  void v() // señalar()
  {
    while(TAS(ocupado) == true);

    valor = valor + 1;
    if (valor <= 0)
    {
      desbloquear'(bloq.sacar()); [6]
    }

    ocupado = false;
  }

private:
  volatile bool ocupado;
  volatile int valor;
  volatile lista<hebra> bloq;
};
```

[5] bloquear'() = bloquear() + ocupado = false

[6] desbloquear'() = ocupado = true + desbloquear()

Semáforo débil

```
class semáforo_débil
{
public:
  semáforo_débil(int _valor = 0):
    valor(_valor),
    bloq(vacía)

  void p() // esperar()
  {
    valor = valor - 1;
    if (valor < 0)
    {
      bloq.meter(esta);
      bloquear(esta);
    }
  }

  void v() // señalar()
  {
    valor = valor + 1;
    if (valor <= 0)
    {
      desbloquear(bloq.sacar());
    }
  }

private:
  int valor;
  lista<hebra> bloq;
};
```

¡No se respeta el orden de llegada!

Semáforo fuerte (estricto)

```
class semáforo_fuerte
{
public:
  semáforo_fuerte(int _valor = 0):
    valor(_valor),
    bloq(vacía)

  void p() // esperar()
  {
    valor = valor - 1;
    if (valor < 0)
    {
      bloq.añadir(esta);
      bloquear(esta);
    }
  }

  void v() // señalar()
  {
    valor = valor + 1;
    if (valor <= 0)
    {
      desbloquear(bloq.primero());
    }
  }

private:
  int valor;
  cola<hebra> bloq;
};
```

Respeta el orden primero en llegar, primero en ser servido $\Longrightarrow$ **espera acotada.**

Aplicaciones de los semáforo con contador

- Supongamos **n hebras** concurrentes:
- Si inicializamos ***s.valor* = 1** sólo una hebra podrá entrar en su sección crítica $\Longrightarrow$ **exclusión mutua**.
- Si inicializamos ***s.valor* = *k*** con ***k* > 1** entonces ***k*** hebras podrán acceder a su sección crítica $\Longleftrightarrow$ ¿Cuándo utilizar así un semáforo? $\Longrightarrow$ **sincronización**, ejemplo: gestión de recursos divisibles.

variable global

```
semáforo s = 1;
```

hebra_i

```
while(true)
{
    s.esperar();
    sección_crítica();
    s.señalar();
    sección_no_crítica();
}
```

Productor/Consumidor (búfer ilimitado) (1)

- Necesitamos un semáforo, `s`, para conseguir la exclusión mutua en el acceso al búfer.
- Necesitamos otro semáforo, `n`, para sincronizar el número de elementos del búfer por parte del productor y el consumidor: sólo podemos consumir después de haber producido.
- El productor es libre para añadir nuevos elementos al búfer en cualquier instante, pero antes debe hacer `s.esperar()`. Tras añadir un elemento debe hacer `s.señalar()` para garantizar la exclusión mutua.
- El productor debe hacer `n.señalar()` después de cada ejecución de su sección crítica para avisar al consumidor que hay elementos para consumir.
- El consumidor debe hacer `n.esperar()` antes de ejecutar su sección crítica para asegurarse de que el búfer contiene elementos.

Productor/Consumidor (búfer ilimitado) (2)

```
semáforo s = 1;                    // exclusión mutua búfer
semáforo n = 0;                    // número de elementos
volatile elemento búfer[];         // búfer ilimitado
volatile int entra = 0, sale = 0; // posiciones del búfer

void* productor(void*)
{
  while(true)
  {
    elemento e = producir();
    s.esperar();
    búfer[entra++] = e;
    s.señalar();
    n.señalar();
  }
}
```

```
void* consumidor(void*)
{
  while(true)
  {
    n.esperar(); // orden
    s.esperar(); // orden
    elemento e = búfer[sale++];
    s.señalar();
    consumir(e);
  }
}
```

[7] ¿Por qué es importante el orden de las 2 llamadas a esperar()?

Productor/Consumidor (búfer ilimitado) (3)

- ¿Necesitamos exclusión mutua para el búfer? → si.
- ¿Por qué hace `n.esperar()` el productor? → permite al consumidor acceder a su sección crítica.
- ¿Por qué es importante el orden de las llamadas a `esperar()` en el consumidor? → evita interbloqueo.
- ¿Es el orden de las llamadas a `señalar()` en el productor importante? → si, por eficiencia.
- ¿Es esta solución extensible a más de dos productores y/o consumidores? → si.
- ¿Cómo modificaría esta implementación para el caso del búfer limitado? → añadir un nuevo semáforo para evitar producir sobre un búfer lleno.

Productores/Consumidores (búfer limitado)

```
const unsigned N = 100;              // tamaño del búfer
volatile elemento búfer[];           // búfer ilimitado
volatile int entra = 0, sale = 0; // posiciones del búfer
semáforo exclusión = 1;              // exclusión mutua búfer
semáforo elementos = 0;              // número de elementos introducidos
semáforo huecos = N;                 // número de huecos libres

void* productor(void*)
{
  while(true)
  {
    elemento e = producir();

    huecos.esperar();
    exclusión.esperar();

    búfer[entra] = e;
    entra = (entra + 1) % N;

    exclusión.señalar();
    elementos.señalar();
  }
}
```

```
void* consumidor(void*)
{
  while(true)
  {
    elementos.esperar();
    exclusión.esperar();

    elemento e = búfer[sale];
    sale = (sale + 1) % N;

    exclusión.señalar();
    huecos.señalar()

    consumir(e);
  }
}
```

Semáforos binarios, cerrojos o "*mutex*"

- En lugar de implementar `adquirir()` mediante espera ocupada podríamos utilizar el API del núcleo: `bloquear()` y `desbloquear()`.
- Cuando el cerrojo ya está en posesión de una hebra, cualquier otra que llame a `adquirir()` será bloqueada.
- La hebra debería permanecer bloqueada hasta que pueda adquirir el cerrojo.
- El procesador queda libre para ejecutar otras tareas.
- También es necesario modificar `liberar()` para tener en cuenta las hebras bloqueadas.
- Cada semáforo binario tiene asociada una cola de hebras bloqueadas.

Semáforo binario

```
class semáforo_binario
{
public:
  semáforo_binario():
    ocupado(false),
    bloq(vacía)

  void esperar()
  {
    if (!ocupado)
      ocupado = true;
    else
    {
      cola.añadir(esta);
      bloquear(esta);
    }
  }

  void señalar()
  {
    if (cola.vacía())
      ocupado = false;
    else
      desbloquear(cola.primero());
  }

private:
  volatile bool ocupado;
  volatile cola<hebra> bloq;
};
```

Esta solución tiene efectos adversos muy serios sobre la arquitectura del sistema.

Semáforo binario: conflicto de depuración

```
void esperar()
{
  if (!ocupado)
    ocupado = true;
  else
  {
    cola.añadir(esta);
    bloquear(esta);
  }
}

void señalar()
{
  if (cola.vacía())
    ocupado = false;
  else
    desbloquear(cola.primero());
}
```

```
void depurador(hebra h)
{
  bloquear(h);
  operaciones de depuración sobre h;
  desbloquear(h);
}
```

Conflicto entre depurador y depurado:

- Doble `bloquear()` $\Longrightarrow$ uno es "olvidado".
- `desbloquear()` en el `depurador()` **viola la exclusión mutua** al permitir el acceso a su sección crítica a la hebra que ejecutó `esperar()`.
- `desbloquear()` en `señalar()` viola el funcionamiento del depurador.

Independencia conceptual (1)

- `bloquear()` y `desbloquear()` son operaciones de **planificación pura**...
  - sin embargo las estamos utilizando para **transferir información** entre hebras.
  - el simple hecho de que alguien **desbloquee** una hebra le **indica** que **posee** acceso a la sección crítica.
- Consecuencias:
  - la planificación y los semáforos son **operaciones fuertemente acopladas**.
  - Cualquier método de sincronización que utilice `bloquear()` y `desbloquear()` debería tener en cuenta sus consecuencias sobre los semáforos.
- Todo método de sincronización, aunque lógicamente independiente de la planificación, debe ser implementado en un único módulo.

Independencia conceptual (2)

¿Cómo arreglarlo?

Intento 1:

- **Contar las llamadas** a `bloquear()` y `desbloquear()`:
  - n × `bloquear()` → n × `desbloquear()`.
- `esperar()`, `señalar()`, depuración y planificación deben coexistir.
- Sin embargo el sistema podría dejar de funcionar si no emparejamos de forma correcta las llamadas a `bloquear()` y `desbloquear()` → solución no válida.

Intento 2:

- **No transferir información** a través de `desbloquear()`.
- Transferir la información acerca de quien **posee** el semáforo de forma **explícita**.
- Utilizar `bloquear()` y `desbloquear()` tan sólo como optimización.

Semáforo binario

```
class semáforo_binario {
public:
  semáforo_binario():
    propietario(nadie),
    bloq(vacía)

void esperar()
{
  if (propietario == nadie)
    propietario = esta;
  else
  {
    cola.añadir(esta);
    do
      bloquear(esta);
    while (propietario != esta);
  }
}

void señalar()
{
  if (cola.vacía())
    propietario = nadie;
  else
  {
    propietario = cola.primero();
    desbloquear(propietario);
  }
}

private:
  volatile hebra propietario;
  volatile cola<hebra> bloq;
};
```

Independencia conceptual: conclusiones

Como reglas de diseño...

- Mantener independientes las cosas independientes.
- Evitar cualquier técnica que induzca efectos laterales.

Consecuencias:

- El diseño y la implementación de semáforos requiere habilidad y visión de futuro.
- Modificar las operaciones del núcleo es justo lo contrario.

En resumen...

- Los semáforos son una herramienta de coordinación primitiva:
  - para garantizar la **exclusión mutua**.
  - para **sincronizar** hebras.
- Las llamadas a `esperar()` y `señalar()` suelen estar dispersas por varias hebras con lo que es difícil comprender todos sus efectos.
- El uso debe ser correcto en todas estas las hebras.
- Una hebra defectuosa o maliciosa puede estropear el funcionamiento del resto.

Algunos autores recomiendan evitar su uso $\Longrightarrow$ ¿Qué utilizar en su lugar? ¿Existen mejores métodos de sincronización?

Monitores

- Construcciones de alto nivel de algunos lenguajes de programación.
  - Funcionamiento parecido al de los semáforos binarios.
  - Gestión automática por parte del lenguaje.
- Ofrecido en lenguajes de programación concurrente como Java, Modula-3, Pascal concurrente,...
- Internamente puede ser implementado mediante semáforos u otros mecanismos de sincronización.

Definición de monitor

- Un módulo software (clase) que contiene:
  - Una interfaz con uno o más **procedimientos**.
  - Una secuencia de **inicialización**.
  - **Variables** de datos locales.
- Características:
  - Las variables locales sólo son accesibles mediante los procedimientos del monitor.
  - Una hebra entra en el monitor mediante la invocación de uno de sus procedimientos.
  - Sólo una hebra puede estar ejecutando algún procedimiento del monitor $\longrightarrow$ los monitores proporcionan **exclusión mutua** de forma automática.

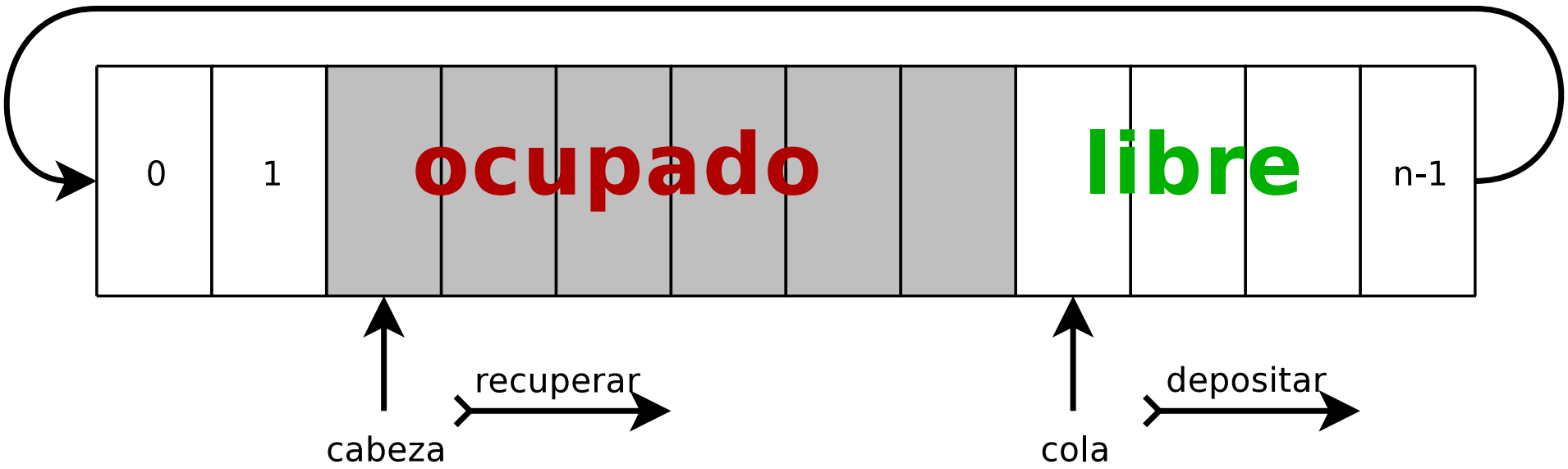
Características de los monitores

- Los monitores aseguran la **exclusión mutua** $\Longrightarrow$ **no** es necesario programar esta restricción **explícitamente**.
- Los **datos compartidos** son **protegidos automáticamente** si los colocamos dentro de un monitor.
- Los monitores **bloquean** sus datos cuando una hebra entra.
- **Sincronización** adicional puede conseguirse **dentro** de un monitor mediante el uso de **variables condición**.
- Una variable de condición representa una condición (evento) que tiene que cumplirse antes de que una hebra pueda **continuar** la ejecución de alguno de los procedimientos del monitor.

Ejemplo: Búfer circular (1)

Ejemplo de monitor **sin** variables condición

- Utilización de un vector contiguo de `N` posiciones como búfer circular.
- Interfaz con operaciones: `recuperar()` y `depositar()`



Ejemplo: búfer circular (2)

```
monitor búfer_circular
{
public:
  búfer_circular():
    búfer(vacio),
    cabeza(0),
    cola(0),
    contador(0) {}

  void depositar(elemento e)
  {
    búfer[cola] = e;
    cola = (cola + 1) % N;
    contador = contador + 1;
  }

  void recuperar(elemento e)
  {
    e = búfer[cabeza];
    cabeza = (cabeza + 1) % N;
    contador = contador - 1;
  }

private:
  elemento búfer[N];
  int cabeza, cola, contador;
};
```

- Exclusión mutua automática entre hebras.
- Operaciones **serializadas** pero aun podemos `depositar()` sobre un **búfer lleno** o `recuperar()` desde un **búfer vacio** $\longrightarrow$ 2 restricciones más.

Ejemplo: búfer circular (3)

```
monitor búfer_circular {
public:
  búfer_circular():
    búfer(vacio), cabeza(0),
    cola(0), contador(0),
    dep(vacía), rec(vacía) {}

  void depositar(elemento e)
  {
    while (contador == N)
    * bloquear(dep.meter(esta));
    búfer[cola] = e;
    cola = (cola + 1) % N;
    contador = contador + 1;
    if (!rec.vacía())
      desbloquear(rec.sacar());
  }

  void recuperar(elemento e)
  {
    while (contador == 0)
    * bloquear(rec.meter(esta));
    e = búfer[cabeza];
    cabeza = (cabeza + 1) % N;
    contador = contador - 1;
    if (!dep.vacía())
      desbloquear(dep.sacar());
  }

private:
  elemento búfer[N];
  int cabeza, cola, contador;
  cola<hebra> dep, rec;
};
```

Ya no podemos depositar sobre un búfer lleno o recuperar desde un búfer vacio pero... **bloqueamos el monitor** (*).

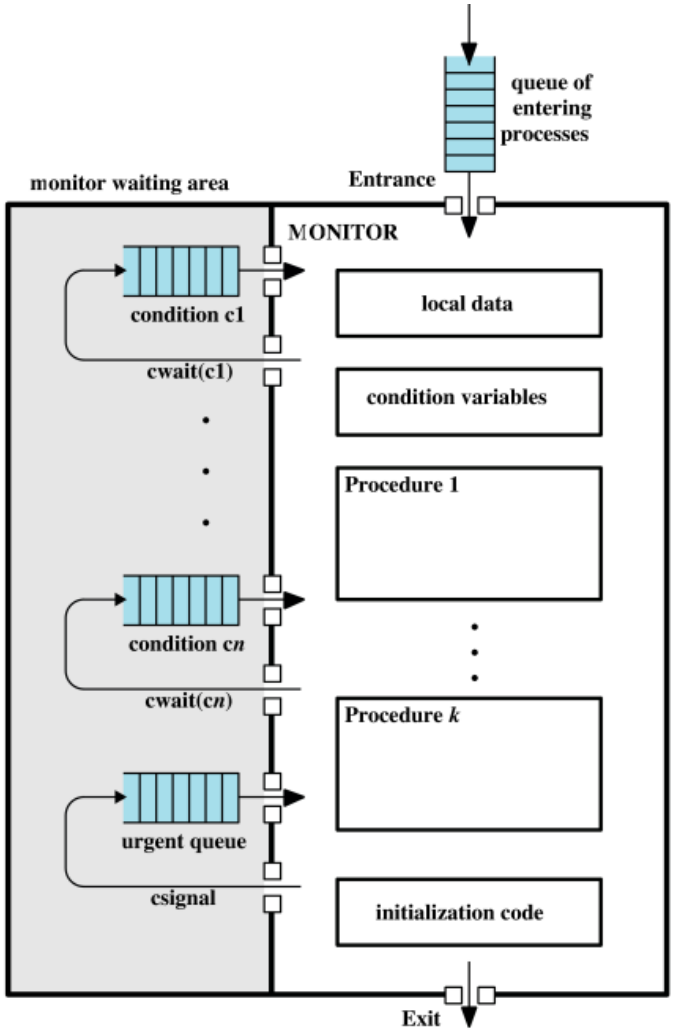
Variables condición

- Variables locales al monitor y sólo **accesibles** desde su **interior** mediante las operaciones:

`esperar()`: Bloquea la ejecución de la hebra llamadora sobre la variable condición. La hebra bloqueada sólo puede continuar su ejecución si otra ejecuta `señalar()`.

`señalar()`: Reactiva la ejecución de alguna hebra bloqueada en la variable condición. Si hay varias podemos escoger una cualquiera. Si no hay ninguna no hace nada.

Monitores y variables condición



- Las hebras pueden estar esperando en la cola de entrada o en la de alguna variable de condición.
- Una hebra se pone en la cola de espera al invocar `esperar()` sobre una variable de condición.
- `señalar()` permite a una hebra que estuviese bloqueada en la cola de una condición continuar.
- `señalar()` bloquea a la hebra llamadora y la pone en la cola urgente a menos que sea la última operación del procedimiento (¿?).

Ejemplo: búfer circular (v2)

Ejemplo de monitor **con** variables condición

```
monitor búfer_circular
{
public:
  búfer_circular():
    búfer(no_vacio), cabeza(0),
    cola(0), contador(0) {}

  void depositar(elemento e)
  {
    while (contador == N)
      no_lleno.esperar();

    búfer[cola] = e;
    cola = (cola + 1) % N;
    contador = contador + 1;

    no_vacio.señalar();
  }

  void recuperar(elemento e)
  {
    while (contador == 0)
      no_vacio.esperar();

    e = búfer[cabeza];
    cabeza = (cabeza + 1) % N;
    contador = contador - 1;

    no_lleno.señalar();
  }

private:
  elemento búfer[N];
  int cabeza, cola, contador;
  variable_condición no_lleno, no_vacio;
};
```

No bloquea el monitor, sólo hebras.

Productor/consumidor con monitores

- Dos tipos de hebras:
  1. productor
  2. consumidor
- La sincronización queda confinada en el interior del monitor.
- `depositar()` y `recuperar()` son métodos del monitor.
- Si estos métodos se implementan correctamente cualquier hebra podrá utilizarlos para sincronizarse de forma correcta.

$productor_i$

```
while (true)
{
  producir(e);
  depositar(e);
}
```

$consumidor_i$

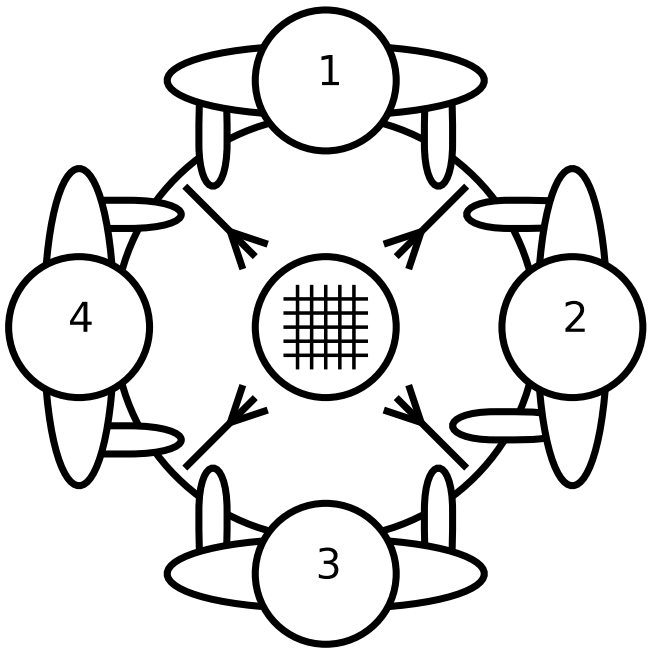
```
while (true)
{
  recuperar(e);
  consumir(e);
}
```

Comentarios y problemas sin resolver

- Si una hebra H_i ejecuta `señalar()` mientras que otra, H_j, estaba bloqueada debido a una ejecución previa de `esperar()`, ¿Cuál de las dos debería continuar ejecutándose? $\longrightarrow$ estilos Hoare/Mesa.
- Un monitor debería permanecer cerrado si algún evento iniciado externamente ocurriese, por ejemplo el fin del quantum asignado. De otra forma no podríamos garantizar la exclusión mutua $\longleftrightarrow$ otra hebra podría ejecutar el monitor.
- ¿Qué hacer si un método de un monitor M_i invoca un método de otro monitor M_j?

Problemas

El problema de la cena de los filósofos (1)



vida de un filósofo

```
repetir
{
    pensar
    estar hambriento
    coger tenedores
    comer
    soltar tenedores
}
```

El problema de la cena de los filósofos (2)

Posibles soluciones:

Turno cíclico Se empieza por un filósofo, si quiere comer lo hace y, en cualquier caso, pasa el turno al siguiente.

Colas de tenedores Cada tenedor tiene asociada una cola de peticiones. Si el filósofo no puede coger los dos tenedores devuelve el que tiene y espera una cantidad de tiempo aleatorio antes de volver a intentarlo.

Camarero Deja que como máximo se sienten a comer $n-1$ filósofos con lo que se garantiza que al menos uno de ellos pueda comer.

Jerarquía de recursos Numerar los tenedores y coger siempre el de menor valor antes de coger el de mayor. Devolverlos en orden contrario.

El problema de la cena de los filósofos (3)

Solución de Chandy/Misra (1984)

Notas:

- Cada tenedor puede estar limpio o sucio. Inicialmente todos los tenedores están sucios.
- El sistema debe inicializarse de forma no perfectamente simétrica, ej: todos los filósofos poseen el tenedor izquierdo.

Algoritmo:

1. Para cada par de filósofos que compiten por un tenedor dárselo al de menor identificador.
2. Cuando un filósofo desea quiere comer debe obtener los dos tenedores de sus vecinos. Por cada tenedor que no tenga debe enviar un mensaje para solicitarlo.
3. Cuando un filósofo posee un tenedor y recibe una petición pueden pasar dos cosas:
   - Si está limpio lo conserva.
   - Si está sucio lo limpia y lo pasa.
4. Después de comer los dos tenedores de un filósofo están sucios. Si otro filósofo solicita alguno de ellos lo limpia y lo pasa.

El problema de la cena de los filósofos (4)

¿Libre de interbloqueos?

- Supongamos un interbloqueo: todos los filósofos tienen el tenedor izquierdo limpio.
- El F_1 obtuvo su tenedor de F_2 porque...
  - el tenedor estaba sucio después de que F_2 comiera.
  - los tenedores sólo se limpian al pasarse.
  - los tenedores limpios nunca se pasan.
- F_2 comió después de F_1, F_3 comió después de F_2, F_4 comió después de F_3 y F_1 comió después de $F_4 \Longrightarrow F_1$ comió después de $F_1 \Longrightarrow$ contradicción.
- El interbloqueo es imposible

El problema de la cena de los filósofos (5)

¿Libre de inanición?

- Un filósofo pasa hambre si...
  - tiene un tenedor limpio.
  - no tiene ningún tenedor.
- Imaginemos que el F_1 acaba sin ningún tenedor, entonces...
  - el F_2 pasa hambre con el tenedor derecho limpio.
  - el F_4 pasa hambre con el tenedor izquierdo limpio.
  - el F_3 también pasaría hambre porque no puede estar comiendo eternamente $\Longrightarrow$ contradicción con interbloqueo.
- Supongamos entonces que el F_1 tiene el tenedor izquierdo limpio, entonces...
  - el F_2 pasa hambre con el tenedor izquierdo limpio.
  - los F_3 y F_4 también pasarían hambre por la misma causa $\Longrightarrow$ contradicción con interbloqueo.
- La inanición es imposible.

El problema de los lectores/escritores

Los lectores tienen prioridad (W. Stallings)

```
unsigned numero_lectores = 0; // numero de lectores
semaforo no_lectores = 1;     // exclusion mutua para numero_lectores
semaforo no_escritor = 1;     // exclusion mutua datos compartidos
```

```
void* lector(void*)
{
  while(true)
  {
    no_lectores.esperar();
    ++numero_lectores;
    if (numero_lectores == 1)
      no_escritor.esperar();
    no_lectores.señalar();
    leer();
    no_lectores.esperar();
    --numero_lectores;
    if (numero_lectores == 0)
      no_escritor.señalar();
    no_lectores.señalar();
  }
}
```

```
void* escritor(void*)
{
  while(true)
  {
    no_escritor.esperar();
    escribir();
    no_escritor.señalar();
  }
}
```

El problema de los lectores/escritores (2)

Los lectores tienen prioridad (W. Stallings)

```
void* lector(void*) {
  while(true) {
    no_lectores.esperar();
    ++numero_lectores;
    if (numero_lectores == 1)
      no_escritor.esperar();
    no_lectores.señalar();
    leer();
    no_lectores.esperar();
    --numero_lectores;
    if (numero_lectores == 0)
      no_escritor.señalar();
    no_lectores.señalar();
  }
}
```

```
void* escritor(void*)
{
  while(true)
  {
    no_escritor.esperar();
    escribir();
    no_escritor.señalar();
  }
}
```

El problema de los lectores/escritores (3)

Los escritores tienen prioridad (J. Bacon)

```
class RWProtocol
{
private:
  semáforo
    R,      // lecturas pendientes
    W,      // escrituras pendientes
    WGUARD, // escritura exclusiva
    CGUARD; // exclusión contadores
  int
    ar, // lectores activos
    rr, // lectores leyendo
    aw, // escritores activos
    ww; // escritores escribiendo

public:
  RWProtocol()
    : ar(0), rr(0), aw(0), ww(0)
  {
    sem_init(R     , 0, 0);
    sem_init(W     , 0, 0);
    sem_init(WGUARD, 0, 1);
    sem_init(CGUARD, 0, 1);
  }
```

```
void adquirir_lector()
{
  sem_wait(CGUARD);
  ar = ar + 1;
  if (aw == 0)
  {
    rr = rr + 1;
    sem_post(R);
  }
  sem_post(CGUARD);
  sem_wait(R);
}

void liberar_lector()
{
  sem_wait(CGUARD);
  rr = rr - 1;
  ar = ar - 1;
  if (rr == 0)
    while (ww < aw)
    {
      ww = ww + 1;
      sem_post(W);
    }
  sem_post(CGUARD);
}
```

```
void adquirir_escritor()
{
  sem_wait(CGUARD);
  aw = aw + 1;
  if (rr == 0)
  {
    ww = ww + 1;
    sem_post(W);
  }
  sem_post(CGUARD);
  sem_wait(W);
  sem_wait(WGUARD);
}

void liberar_escritor()
{
  sem_post(WGUARD);
  sem_wait(CGUARD);
  ww = ww - 1;
  aw = aw - 1;
  if (aw == 0)
    while (rr < ar)
    {
      rr = rr + 1;
      sem_post(R);
    }
  sem_post(CGUARD);
}
};
```

Pthreads

Semáforos binarios: **mutex** en Pthreads

`#include <pthreads.h>` Cabecera C/C++.

`pthread_mutex_t` Tipo mutex, inicializable a PTHREAD_MUTEX_INITIALIZER.

`pthread_mutex_init(mutex, attr)` Crea e inicializa `mutex` con los atributos `attr`.

`pthread_mutex_destroy(mutex)` Destruye `mutex`.

`pthread_mutex_lock(mutex)` Adquiere `mutex` en caso de estar libre. En otro caso bloquea la hebra.

`pthread_mutex_unlock(mutex)` Desbloquea `mutex`.

`pthread_mutex_trylock(mutex)` Intenta bloquear `mutex` y en caso de no poder continua sin bloquear la hebra.

Semáforos: semáforos POSIX

`#include <semaphore.h>` Cabecera C/C++.

`sem_t` Tipo semáforo.

`sem_init(sem, attr, valor)` Inicializa el semáforo `sem` al valor `valor` con los atributos `attr`.

`sem_destroy(sem)` Destruye el semáforo `sem`.

`sem_wait(sem)` Si el valor del semáforo `sem` es positivo lo decrementa y retorna inmediatamente. En otro caso se bloquea hasta poder hacerlo.

`sem_trywait(sem)` Versión no bloqueante de `sem_wait(sem)`. En cualquier caso retorna inmediatamente. Es necesario comprobar la salida antes de continuar.

`sem_post(sem)` Incrementa el valor del semáforo `sem`. En caso de cambiar a un valor positivo desbloquea a alguno de los llamadores bloqueados en `sem_wait(sem)`.

Variables condición: Pthreads

`pthread_cond_t` Tipo variable de condición. Inicializable a PTHREAD_COND_INITIALIZER.

`pthread_cond_init(cond,attr)` Inicializa la variable de condición `cond` con los atributos `attr`.

`pthread_cond_destroy(cond)` Destruye la variable de condición `cond`.

`pthread_cond_wait(cond, mutex)` Bloque a la hebra llamadora hasta que se señale `cond`. Esta función debe llamarse mientras `mutex` está ocupado y ella se encargará de liberarlo automáticamente mientas espera. Después de la señal la hebra es despertada y el cerrojo es ocupado de nuevo. El programador es responsable de desocupar `mutex` al finalizar la sección crítica para la que se emplea.

`pthread_cond_signal(cond)` Función para despertar a otra hebra que espera que se señale sobre la variable de condición `cond`. Debe llamarse después de que `mutex` esté ocupado y se encarga de liberarlo en `pthread_cond_wait(cond, mutex)`.

`pthread_cond_broadcast(cond)` Igual que la función anterior para el caso de que queramos desbloquear a varias hebras que esperan.