Sincronización

Proceso 1	Proceso 2	Proceso 3

while(1) {
agregarTarea();
 	
tareasPendientes ++;
}
	while(1) {
	realizarTarea();
	
	tareasPendientes --;
	}
		while(1) {
		If (tareasPendientes == 0) {
		printf("No hay tareas
		pendientes");
		}
		}

```
registro = tareasPendientes
registro = registro + 1
tareasPendientes = registro
```

```
registro = tareasPendientes
registro = registro - 1
tareasPendientes = registro
```

```
registro = tareasPendientes
cmp registro 0
jz (ejecuta  printf)
jnz (sale del if)
```


RACE CONDITION YOU SAID?
NOOOOOOOOOOOOOOO

Es una situación en la que varios procesos/hilos manipulan datos compartidos concurrentemente, de forma que el resultado de la ejecución depende del orden particular en que se terminan ejecutando

Hay que asegurar que sólo un proceso/hilo pueda acceder a estos datos a la vez para garantizar la coherencia de mismos, hay que sincronizarlos

REGIÓN CRÍTICA

Siempre que más de un proceso acceda a datos compartidos tengo que sincronizarlos??

Independientes **Cooperativos** **Competitivos**

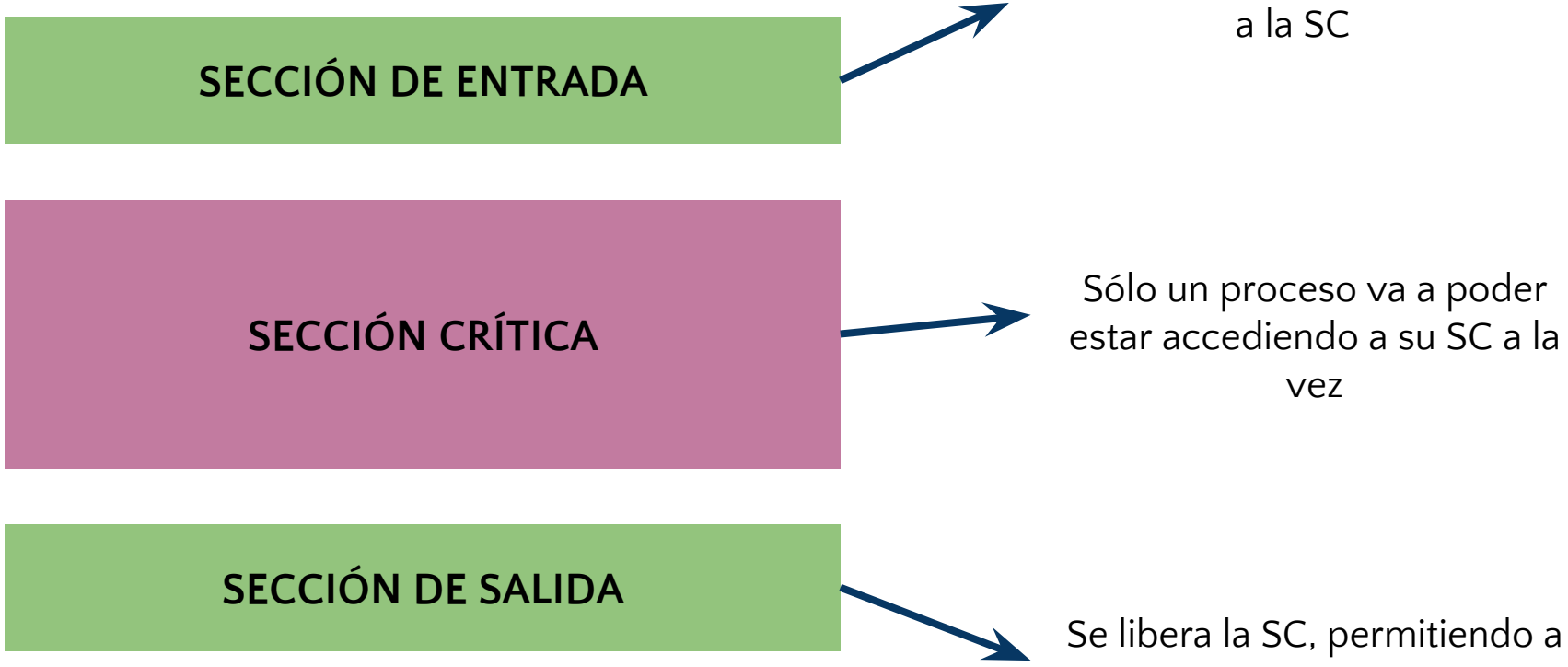
- Siendo W, conjunto de escritura y R conjunto de escritura, si se cumplen:

$$Wa \cap Wb = \{\}$$
$$Wa \cap Rb = \{\}$$
$$Ra \cap Wb = \{\}$$

Los procesos en forma concurrente no modifican los datos compartidos

No tenemos que preocuparnos por sincronizar dichos procesos

Protocolo para acceder a la sección crítica



REQUERIMIENTOS SOLUCIÓN A PROBLEMA SC

Mutua exclusión	Sólo un proceso puede acceder a su sección crítica a la vez
Progreso	Si ningún proceso está ejecutando su sección crítica y existen algunos que quieren entrar en la misma, sólo los procesos que no estén ejecutando su sección restante pueden participar en la decisión de qué proceso puede ingresar en su sección crítica, y esta selección no puede posponerse indefinidamente
Espera limitada	Tiene que haber un límite en la cantidad de veces que otros procesos pueden ingresar en sus secciones críticas luego de que un proceso pide ingresar en la suya, es decir, un proceso no debería esperar indefinidamente la autorización para ejecutar su SC.
Velocidad de los procesos	La solución debe funcionar sin importar cómo los procesos usen sus SCs, es decir, si las mismas son largas, cortas, si las utilizan muchas o pocas veces.

Se fueron desarrollando distintas soluciones de software para resolver la entrada y salida a la SC que cumplieran con las condiciones anteriores . . .

SOLUCIÓN 1

```
Proceso 0                          Proceso 1
while (true)                       while (true)
{                                  {
    while (turno != 0);                while (turno != 1);
    SECCIÓN CRÍTICA                    SECCIÓN CRÍTICA
    turno = 1;                         turno = 0;
    SECCIÓN RESTANTE                   SECCIÓN RESTANTE
}                                  }
```

SOLUCIÓN 2

```
Proceso 0                          Proceso 1
while (true)                       while (true)
{                                  {
    interesado[0] = TRUE;              interesado[1] = TRUE;
    while (interesado[1]);             while (interesado[0]);
    SECCIÓN CRÍTICA                    SECCIÓN CRÍTICA
    interesado[0] = FALSE;             interesado[1] = FALSE;
    SECCIÓN RESTANTE                   SECCIÓN RESTANTE
}                                  }
```

	Cumple mutua Exclusión?	Cumple progreso?	Problemas?
S1	✓	✗	Un proceso no puede entrar a su SC si el otro no entró antes (!= velocidades)
S2	✓	✗	Puede generar que ninguno de los dos pueda ingresar a la SC (quedan en el while)

SOLUCIÓN DE PETERSON

```
Proceso 0
while (true)
{
        interesado[0] = TRUE;
        turno = 1;
        while (interesado[1] && turno == 1);
        SECCIÓN CRÍTICA
        interesado[0] = FALSE;
        SECCIÓN RESTANTE
}
```

```
Proceso 1
while (true)
{
        interesado[1] = TRUE;
        turno = 0;
        while (interesado[0] && turno == 0);
        SECCIÓN CRÍTICA
        interesado[1] = FALSE;
        SECCIÓN RESTANTE
}
```

Cumple mutua Exclusión?	Cumple progreso?	Cumple espera limitada?	Problemas?
✓	✓	✓	Está limitado a dos procesos. Asume que LOAD y STORE son atómicas

El HW nos puede dar soporte para resolver el problema de la SC en forma sencilla y eficiente:

OBJETIVO → Evitar que las instrucciones de la SC sean interrumpidas con las de otros procesos

DESHABILITAR INTERRUPCIONES

deshabilitarInterrupciones();

SECCIÓN CRÍTICA

habilitarInterrupcioens

SECCIÖN RESTANTE

✓ No permite que se cambie de proceso una vez en la SC en forma sencilla

✗ No es bueno en sistemas multiprocesador:

- Costo de enviar mensaje a cada CPU
- Baja eficiencia uso CPU
- El proceso tarda más en ingresar a la SC

INSTRUCCIONES ATÓMICAS

TEST AND SET // SWAP AND EXCHANGE

Ejemplo usando TestAndSet

```
while ( TestAndSet (&lock ));
// nada
//    sección crítica
lock = FALSE;
//    sección restante
```

Funcionamiento TestAndSet

```
boolean TestAndSet (boolean *target) {
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

ATÓMICAMENTE

- ✓ Algunos sistemas proveen instrucciones que permiten validar el valor de una variable y modificarla en forma atómica
- ✓ Son sencillos de usar
- ✓ Son eficientes incluso en sistemas multiprocesador
- ✗ No todos los sistemas tienen este soporte

- Un semáforo es una variable entera que es accedida únicamente por dos funciones atómicas (syscalls) **wait** y **signal**

```
SEM = 1
...

WAIT(SEM);

SECCIÓN CRÍTICA

SIGNAL(SEM)

SECCIÓN RESTANTE
```

- A diferencia del resto de las estrategias vistas, se puede implementar con o sin espera activa

SEMÁFOROS: IMPLEMENTACIÓN

CON ESPERA ACTIVA

```
wait (sem) {
    while (sem == 0); // no-op
     sem--;
}

signal (sem) {
    sem++;
}
```

CON BLOQUEO

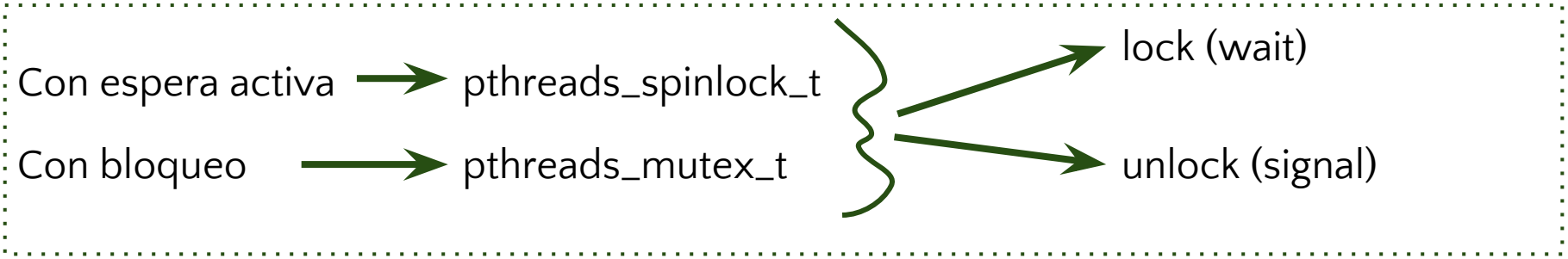
```
wait (S) {
    valor --;
    if (valor < 0) {
        block();
    }
}
```

```
signal (S) {
    valor ++;
    if (valor <= 0) {
        wakeup(pid);
    }
}
```

Cola de espera del semáforo

Despierta al primero que se bloqueo

PTHREADS



Como siempre priorizamos el buen uso de la CPU y no queremos desperdiciarlo con una espera activa, entonces ...

...Siempre es mejor usar la implementación con bloqueo?...

En ciertas situaciones puede ser más eficiente usar los spinlocks

- Cuando hay más de 1 CPU
- Cuando la SC es chica

El proceso en espera activa continúa su ejecución más rápido, nos ahorramos el bloqueo/desbloqueo + cambios de contexto

SEMÁFOROS: TIPOS

MUTEX

Permite solucionar el problema de la exclusión mutua
SIempre se inicializa en **1**

CONTADORES

Permite controlar el acceso a una cantidad de recursos.
Se inicializa en **N** (cantidad de instancias totales)

BINARIOS

Permite garantizar un orden de ejecución.
Representa dos estados, libre u ocupado

Si el valor de un semáforo es > 0 → Indica la cantidad de recursos disponibles de un semáforo contador

Si el valor de un semáforo es < 0 → Indica la cantidad de procesos bloqueados esperando

Puedo inicializar un semáforo con valor negativo?? ...

MUTUA EXCLUSIÓN

ORDENAR EJECUCIÓN

LIMITAR ACCESO A CANTIDAD DE INSTANCIAS

PRODUCTOR CONSUMIDOR

MUTUA EXCLUSIÓN

P1	P2

wait(semVar);
 	
var++;
 	
signal(semVar)
	wait(semVar)
	
	var--;
	
	signal(semVar);

semVar = 1	

LIMITAR ACCESO A CANTIDAD DE INSTANCIAS

P1	P2

wait(semContador);
 	
usarRecurso();
 	
signal(semContador)
	wait(semContador)
	
	usarRecurso();
	
	signal(semContador);

semContador = N -> cantidad total recursos	

ORDENAR EJECUCIÓN

P1	P2

while(1) {
wait(semP1);
 	
printf("NANANANANANA")
 	
signal(semP2);
}
	while(1) {
	wait(semP2);
	
	printf("BATMAN!")
	
	signal(semP1);
	}

semP1 = 1 // semP2 = 0	

PRODUCTOR CONSUMIDOR

P1(CONSUMIDOR)	P2(PRODUCTOR)

while(1) {
wait(tareasPendientes);
wait(mutexLista);
tarea = obtenerTarea(listaTareas);
signal(mutexLista);
signal(lugarEnLista);
 	
ejecutarTarea(tarea);
 	
}
	while(1) {
	
	nuevaTarea = crearTarea();
	
	wait(lugarEnLista);
	wait(mutexLista);
	agregarTarea(nuevaTarea, listaTareas)
	signal(mutexLista);
	signal(tareasPendientes)
	
	}

mutexLista = 1 tareasPendientes = 0 lugarEnLista = 20

Procesos P1, P2, P3 .. cuyas prioridades son: P1 < P2 < P3

T = 0 P1 adquiere un recurso R WAIT(semM)
T = 1 P3 ingresa el sistema y necesita el recurso R, se bloquea en la espera
T = 2 P2 ingresa desaloja a P1 ya que tiene mayor prioridad

P3, que es el proceso de mayor prioridad, no está pudiendo ejecutar porque espera un recurso retenido por P1, un proceso de menor prioridad

HERENCIA DE PRIORIDADES

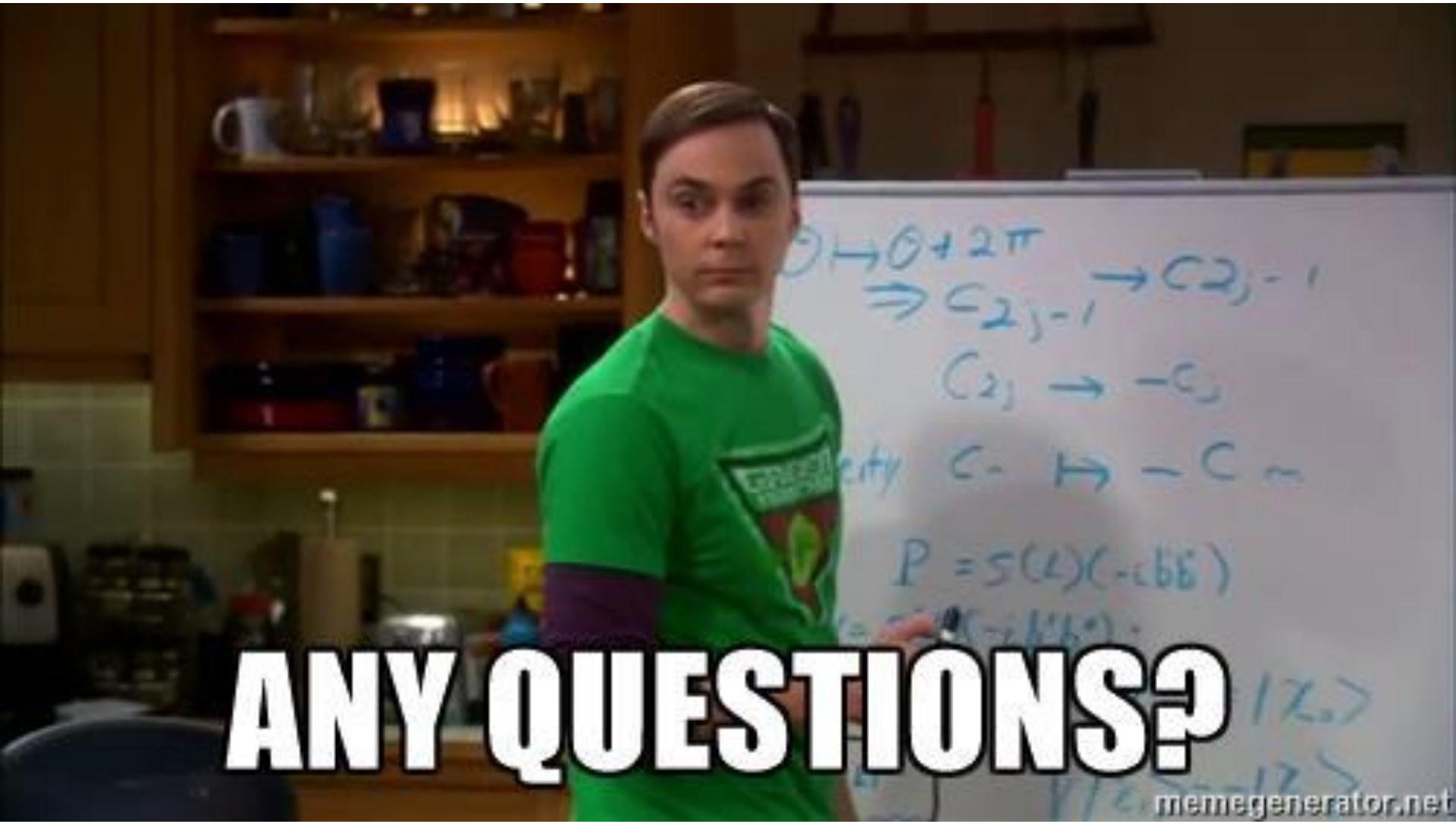
- Es un mecanismo que provee mutua exclusión
- Abstrae en una estructura el acceso a sus datos con ciertas operaciones específicas expuestas
- Un proceso a la vez está activo en el monitor

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
```

Want about to a race conditions hear joke?

"Knock knock". "Race condition". "Who's there?"

Master Yoda was just speaking Race Condition all along!


ANY QUESTIONS?
memegenerator.net