


Planificación de CPU

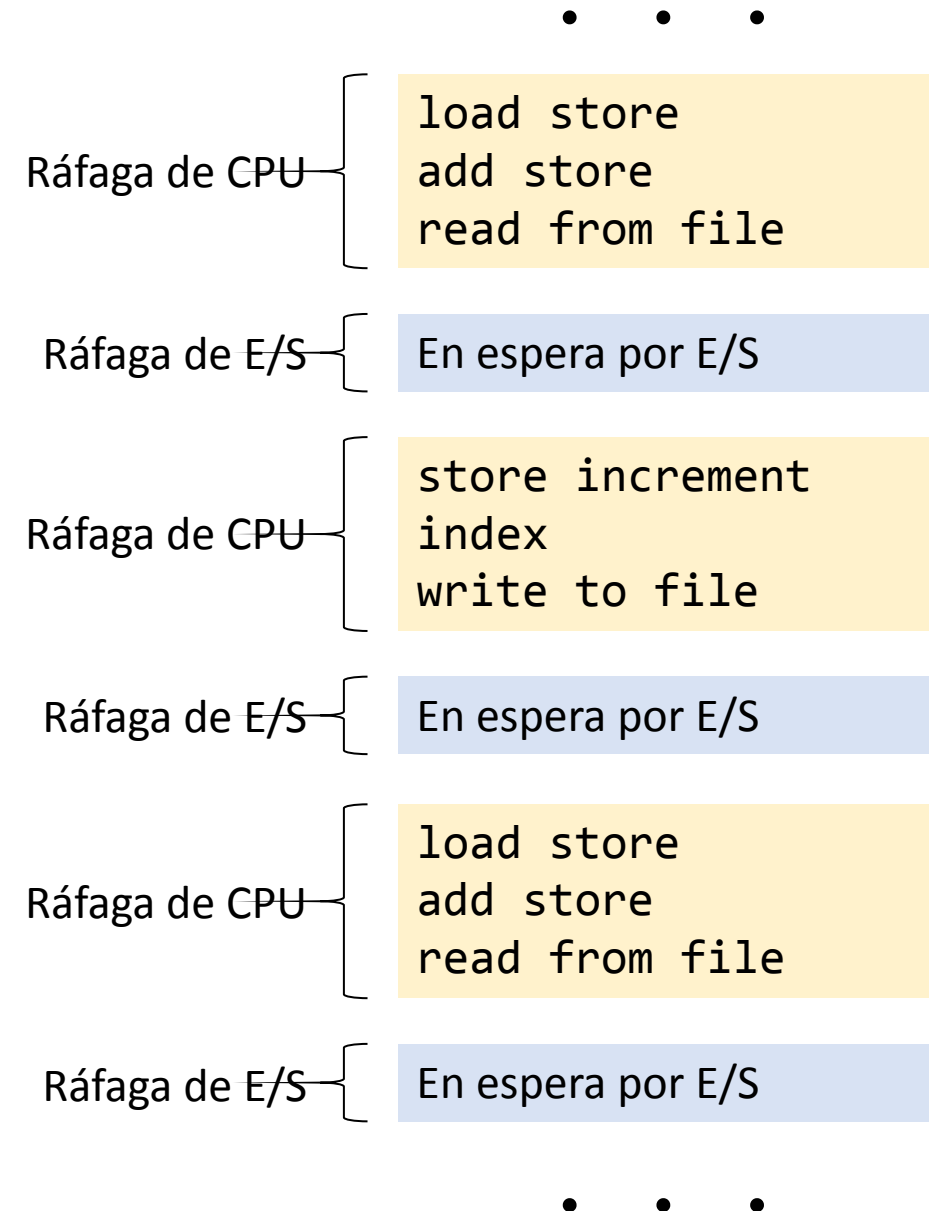
Adaptación de diferentes referencias (ver al final)

Conceptos

- Un solo procesador (o núcleo) significa un solo proceso a la vez en ejecución.
- Grado de multiprogramación  # de procesos activos (Es decir que están en RAM)
 - Número de procesos activos (**en memoria principal**) que mantiene un sistema
 - A más procesos activos mayor probabilidad de encontrar un proceso listo para ejecutar
 - Afecta de forma importante el rendimiento de un computador
- **Objetivo** de la **multiprogramación**
 - Tener **algún proceso** ejecutándose en todo momento para **maximizar** el uso de CPU

Ráfagas de CPU y de E/S

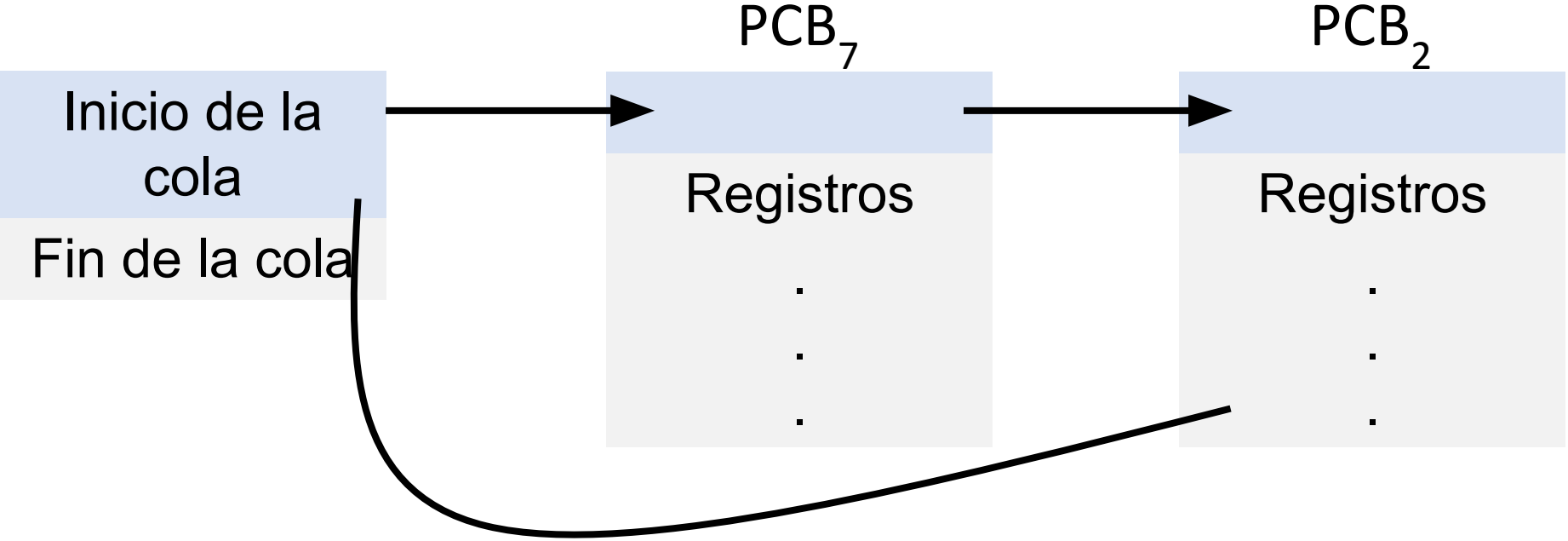
- La ejecución de un proceso consiste en **alternar entre dos estados**
 - En ejecución (**ráfaga de CPU**)
 - En espera por E/S (**ráfaga de E/S**)
- Patrón cíclico de ejecución de un proceso
- **Objetivo** de multiprogramación
 - Usar los tiempos de espera de E/S de un proceso para darle la CPU a otro proceso



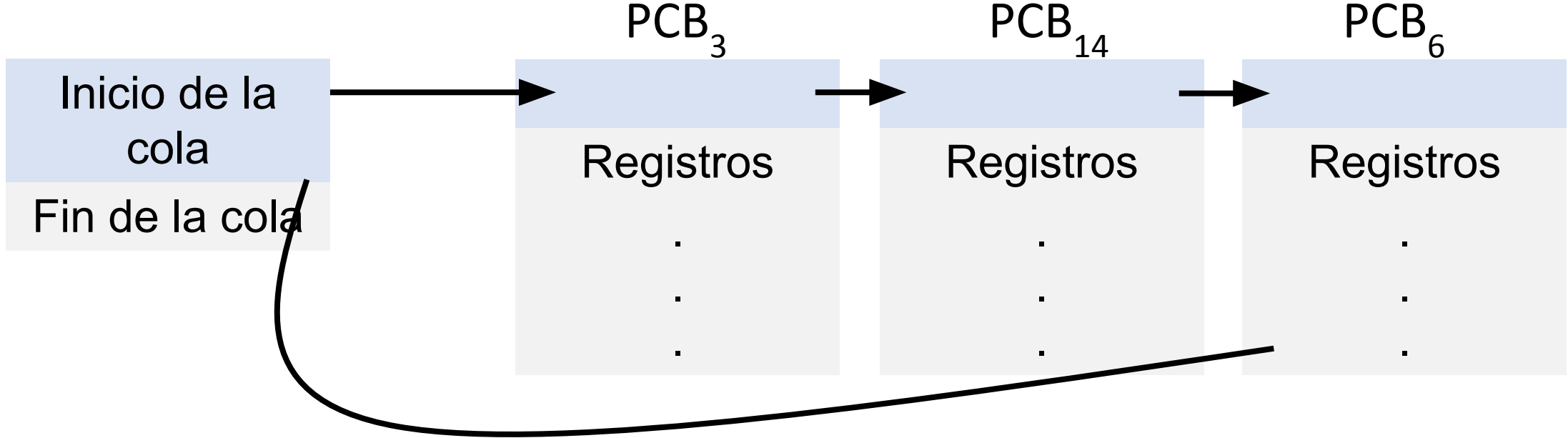
Planificador de CPU

- Forma parte del núcleo del S.O
- Entra en ejecución cada vez que se activa el S.O
- Selecciona el siguiente proceso a ejecutar
 - Lo selecciona de la cola de **listo**.
 - La cola de listo puede atenderse con diferentes disciplinas.
 - **No** necesariamente es FIFO.

Cola listo



Cola espera



Planificación apropiativa y no apropiativa

• Circunstancias en las que entra el planificador a tomar decisiones

1. Proceso pasa de **ejecución** a **espera** (E/S o llamada a wait)
2. Proceso pasa de **ejecución** a **listo** (ocurre una interrupción)
3. Proceso pasa de **espera** a **listo** (se completa una operación de E/S)
4. Proceso termina

hay decisiones
de planificación

Apropiativas

No apropiativas

No hay decisiones
de planificación



Planificación apropiativa y no apropiativa

• Circunstancias 1 y 4

- No hay decisiones en términos de planificación
- Siempre que haya un proceso en estado de listo, él es el que sigue en CPU
- **Planificación NO apropiativa**
- El proceso permanece en CPU hasta que termine o hasta que pase a espera

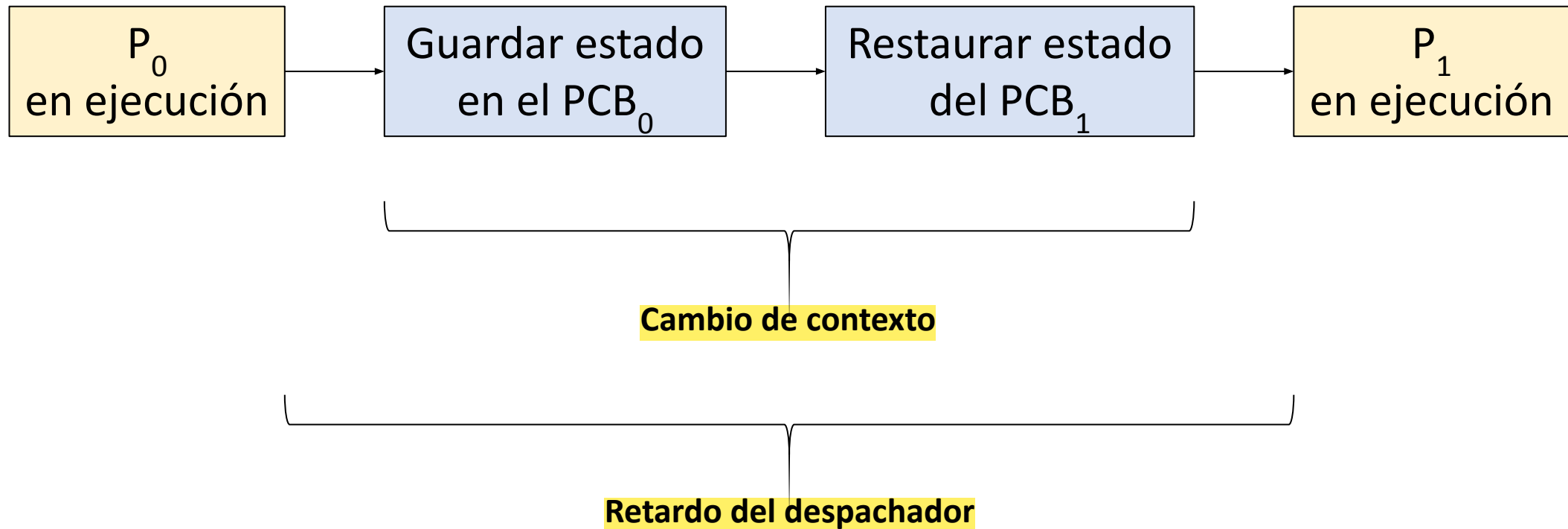
• Circunstancias 2 y 3

- Si hay decisiones en términos de planificación
- ¿A quién se le asigna la CPU? ¿Al proceso interrumpido? ¿A otro? ¿Se interrumpe el que ya está en ejecución?
- **Planificación apropiativa**
- Se dan condiciones de carrera: un proceso actualizando datos (compartidos) y entra otro

Despachador

- Forma parte del núcleo del S.O
- Responsable de entregarle a la CPU el proceso seleccionado por el planificador
 - Realiza el cambio de contexto
 - Cambia el modo de ejecución: pasa de modo kernel (**ring 0**) a modo usuario (**ring 3**)
 - Apunta el registro IP a la siguiente instrucción del proceso que será ejecutado
- Estas operaciones introducen un retardo
 - Retardo del despachador
 - Se desea que sea lo más rápido posible

Despachador



Criterios (objetivos) de planificación

- **Utilización CPU**

- Mantener la CPU lo más ocupada posible: ejecutando algún proceso.

- **Throughput**

- Número de procesos que se completan por unidad de tiempo.

- **Tiempo de ida y vuelta**

- Suma de tiempos: de espera en cola de listo + tiempo en CPU + tiempo en E/S

- **Tiempo de espera**

- Suma de tiempos en los que proceso pasa esperando en la cola de listo.

- **Tiempo de respuesta**

- Tiempo que le toma al proceso comenzar a responder después de recibir una solicitud

Criterios (objetivos) de planificación

- **Deseable maximizar**

- Uso de CPU
- Throughput

- **Deseable minimizar**

- Tiempo de ida y vuelta
- Tiempo de espera
- Tiempo de respuesta

POSIX

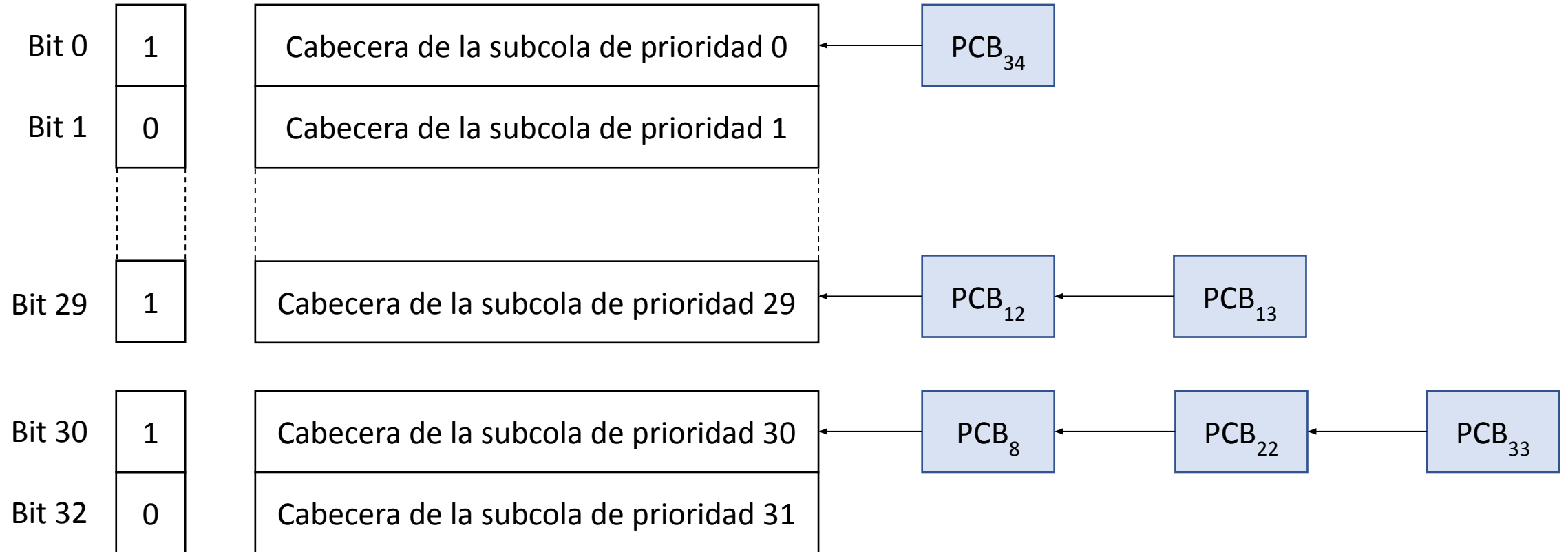
- POSIX es una estándar de IEEE
 - Recomendación de una interfaz estándar del S.O
 - También como Single Unix Specification SUS (hoy SUSv4 POSIX.1-2008)
- Incluye
 - Interfaz estándar del S.O y el entorno
 - Interpretes de comandos
 - Programas y utilidades comunes
- Propósito
 - Apoyar la portabilidad de aplicaciones a nivel de código fuente
- P. Ej.: POSIX.1
 - Biblioteca estándar de C, creación y control de procesos, etc.

Planificación en POSIX

- Cada proceso (o hilo) lleva asociada
 - Una política de planificación
 - Una prioridad
- Cada política de planificación lleva asociada un rango de prioridades
 - Al menos 32 niveles de prioridad según estándar POSIX
 - El planificador selecciona siempre el proceso con la prioridad más alta
- Linux/Unix
 - **40** niveles de prioridad: desde **-20** (la más alta) a **19** (la más baja)
 - Procesos iniciados por usuarios se les asigna prioridad 0
 - El comando **nice** sirve para modificar la prioridad de un proceso en ejecución

Colas de planificación

Palabra resumen



Políticas de planificación en POSIX

- **FIFO**

1. Procesos se agregan al final de la cola de su prioridad asociada
2. Proceso se expulsa de CPU cuando ejecute llamada bloqueante
3. Proceso se expulsa de CPU cuando aparezca un proceso con mayor prioridad

- **Reglas del planificador para FIFO**

1. Proceso expulsado por causa No. 3: proceso expulsado pasa a ser el primero de la cola de su prioridad
2. Proceso pasa de bloqueado a listo: proceso se agrega al final de la cola de su prioridad
3. Cambio de prioridad o política: se realiza replanificación, si resulta expulsado, se agrega al final de la cola

Políticas de planificación en POSIX

- Cíclica

- Se asigna rodaja de tiempo (*quantum*) a procesos en colas de prioridad
- Proceso que acaba su *quantum* se agrega al final de la cola de su prioridad
- Proceso expulsado por otro de mayor prioridad: expulsado se agrega al principio de la cola pero sin restaurar su *quantum*.

Planificación en Windows

- La unidad fundamental de planificación en Windows es el hilo
- Se usan 32 niveles de prioridades en planificación cíclica
 1. Dieciséis niveles para procesos en tiempo real: del 16 al 31
 2. Quince niveles variables: del 1 al 15
 3. Un nivel para el sistema: 0
- Procesos en mismo nivel reciben el mismo *quantum*.
- Procesos en nivel dos:
 - Inician con una prioridad determinada y ésta va cambiando pero sin llegar a nivel 16.
 - Prioridad disminuye si acaba su *quantum*
 - Prioridad aumenta si proceso se bloquea por E/S

Planificación en Windows

- Lectura complementaria
 - Processes, Threads, and Jobs in the Windows Operating System
 - <https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=7>

Referencias

- Carretero Pérez, J., García Carballeira, F., de Miguel Anasagasti, P., & Pérez Costoya, F. (2001). Planificación. In *Sistemas operativos. Una Visión Aplicada* (pp. 102–109). McGraw Hill.
- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). CPU Scheduling. In *Operating Systems Concepts* (10th ed., pp. 199–205). John Wiley & Sons, Inc.

First-Come First-Served

Adaptación (ver referencias)

FCFS

- Primer proceso que solicita la CPU es el primer proceso que la recibe
- Se implementa mediante una cola FIFO
- Proceso que entra a la cola de listo, su PCB se enlaza al final de la cola
- Cuando CPU queda libre, primer proceso en cola de listo se le asigna la CPU
- Desventaja
 - El promedio de tiempo de espera en la cola de listo suele ser alto

Procesos	Tiempo necesario en CPU
P_1	24 ms
P_2	3 ms
P_3	3 ms

Orden de Llegada: P_1, P_2, P_3

Tiempo espera $P_1 = 0$

Tiempo espera $P_2 = 24$

Tiempo espera $P_3 = 27$

Tiempo espera promedio = $(24+27)/3 = 17\text{ms}$



Procesos	Tiempo necesario en CPU
P_1	24 ms
P_2	3 ms
P_3	3 ms

Orden de Llegada: P_2, P_3, P_1

Tiempo espera $P_1 = 6$

Tiempo espera $P_2 = 0$

Tiempo espera $P_3 = 3$

Tiempo espera promedio = $(6+0+3)/3 = 3\text{ms}$



FCFS

- Los tiempos de espera pueden variar en función de los tiempos de ráfagas de CPU.
- Un proceso intensivo de CPU y con ráfagas largas de CPU haría que proceso intensivos de E/S permanezcan mucho tiempo en cola de listo esperando por la CPU.
 - Situación que se podría resolver dando el turno a los procesos de menos necesidad de CPU.
- FCFS es NO apropiativo
 - Proceso en CPU se mantiene hasta que termina o hasta que se bloquea por E/S
 - Mantener la CPU para un solo proceso no conviene en sistemas interactivos.

Referencias

- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). CPU Scheduling. In *Operating Systems Concepts* (10th ed., pp. 205–206). John Wiley & Sons, Inc.

Shortest-Job-First (SJF)

Primero el trabajo más corto

Adaptación (ver referencias)

Shortest-Job-First (SJF)

- Se asocia a cada proceso la longitud de la siguiente ráfaga de CPU
 - En lugar de su longitud total como si sucede en FCFS
- Cuando la CPU está disponible, se le entrega al proceso con la ráfaga de CPU más corta
 - Situaciones de empate se resuelven mediante FCFS

Procesos	Tiempo siguiente ráfaga de CPU
P_1	6 ms
P_2	8 ms
P_3	7 ms
P_4	3 ms

Tiempo espera $P_1 = 3$

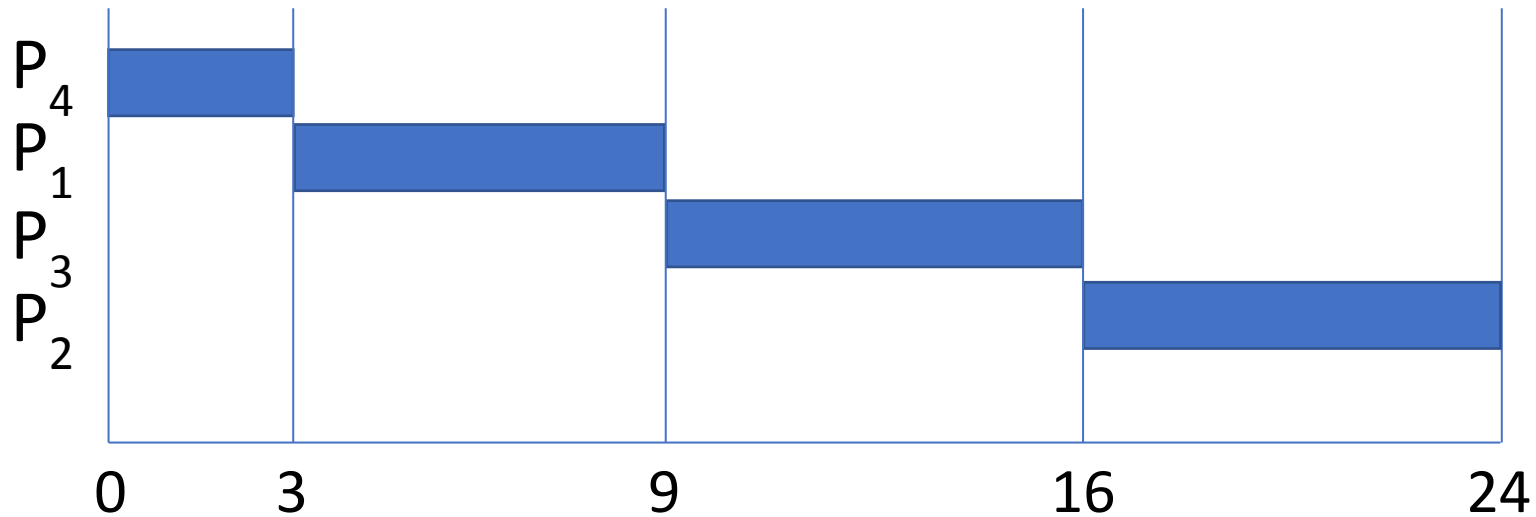
Tiempo espera $P_2 = 16$

Tiempo espera $P_3 = 9$

Tiempo espera $P_4 = 0$

Tiempo espera promedio = 7 ms

Con FCFS
promedio = 10.25 ms



Shortest-Job-First (SJF)

- En la práctica resulta difícil saber (a priori) las necesidades de ráfagas de CPU de un proceso.
- Se puede predecir la duración de la siguiente ráfaga de CPU asumiendo que son similares a las previas.
 - Se usa promedio exponencial para predecir la duración de la siguiente ráfaga de CPU de un proceso.
- Puede ser apropiativo y no apropiativo
 - Decisión tiene lugar cuando llega un nuevo proceso con una ráfaga de CPU más corta de lo que le queda al proceso que actualmente se está ejecutando.

Procesos	Tiempo siguiente ráfaga de CPU	Tiempo de llegada
P ₁	8 ms	0
P ₂	4 ms	1
P ₃	9 ms	2
P ₄	5 ms	3

Tiempo espera P₁ = 10 - 1

Tiempo espera P₂ = 1 - 1

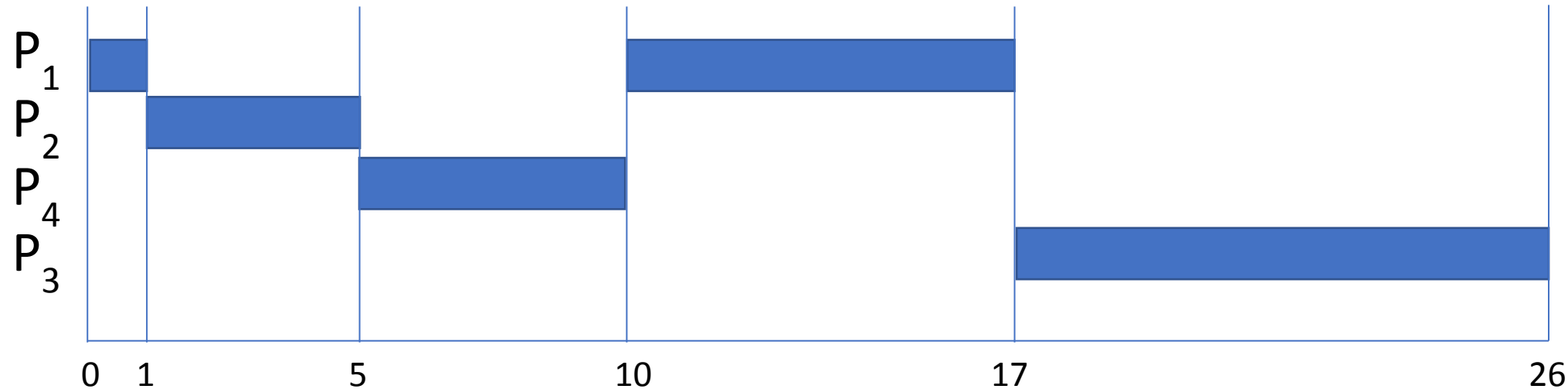
Tiempo espera P₃ = 17 - 2

Tiempo espera P₄ = 5 - 3

Tiempo espera promedio = 6.5 ms

Ejecución no apropiativa
promedio = 7.75 ms

SJF ejecución apropiativa



SJF ejecución apropiativa

- t_0 : P_1 es el único proceso en cola, se ejecuta
- t_1 : Llega P_2 con menos tiempo de CPU
 - Sale P_1 y le quedan 7ms
 - Se ejecuta P_2
- t_2 : Llega P_3 pero requiere 9ms, espera
 - Se sigue ejecutando P_2 , sigue siendo el trabajo más corto
- t_3 : Llega P_4 pero requiere 5 ms, espera
 - P_2 está ejecutando y seguirá siendo el trabajo más corto
- t_5 : Termina P_2 , sale de CPU
 - Entra a ejecutar P_4
 - Aún esperan P_1 con 7ms y P_3 con 9ms
- t_{10} : Se ejecuta todo P_4 y
 - Sale de CPU
 - Entra P_1
- t_{17} : Termina P_1
 - Sale de CPU
 - Entra P_3 y se ejecuta
- t_{26} : Termina P_3

Referencias

- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). CPU Scheduling. In *Operating Systems Concepts* (10th ed., pp. 207–209). John Wiley & Sons, Inc.

Round Robin

(planificación cíclica)

Adaptación (ver referencias)

Planificación cíclica: Round Robin

- Similar a FCFS pero con planificación apropiativa
 - Proceso no permanece todo su *quantum* en CPU.
- Se define una unidad de tiempo (*quantum*)
 - Usualmente está entre 10 y 100 ms
- La cola de listo es una cola circular
- Planificador le asigna la CPU al proceso por un intervalo de 1 quantum.
- Algoritmo con modelo de planificación apropiativa

Planificación cíclica: Round Robin

- Planificador siempre toma el primer proceso de la cola de listo e inicia un temporizador (hasta 1 *quantum*).
 - Proceso que tenga una ráfaga de CPU inferior a 1 *quantum* abandona CPU voluntariamente cuando termina.
 - Proceso con ráfaga de CPU superior a 1 *quantum*, se le asigna la CPU, pero cuando temporizador alcanza límite, se expulsa de CPU
- Para asignar CPU a nuevo proceso se da cambio de contexto.
- Proceso que sale de CPU por vencimiento de *quantum*, se agrega al final de la cola de listo.

Procesos	Tiempo requerido en CPU
P_1	24 ms
P_2	3 ms
P_3	3 ms

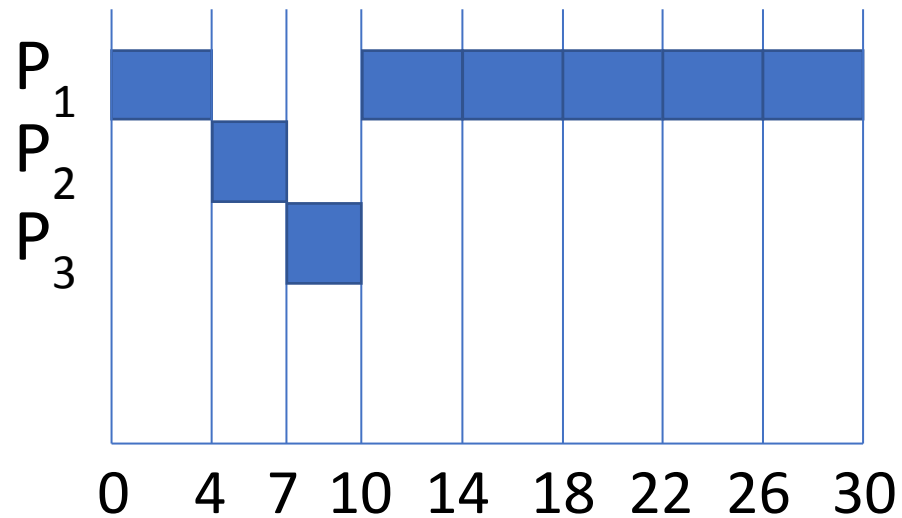
Tiempo espera $P_1 = 10 - 4 = 6$

Tiempo espera $P_2 = 4$

Tiempo espera $P_3 = 7$

Tiempo espera promedio = 5.6 ms

- Todos llegan en t_0
- Quantum = 4ms



Planificación cíclica: Round Robin

- El desempeño del algoritmo depende del tamaño del *quantum*.
- El tamaño del *quantum* se desea que sea lo suficientemente amplio con relación al tiempo de cambio de contexto.
 - Promedio de cambio de contexto 10 micro segundos en computadores modernos.
 - Pero no tan amplio porque termina siendo FCFS.
- Tiempo de ida y vuelta también depende del *quantum*
 - Suma de tiempos: de espera en cola de listo + tiempo en CPU + tiempo en E/S

Referencias

- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). CPU Scheduling. In *Operating Systems Concepts* (10th ed., pp. 207–209). John Wiley & Sons, Inc.

Planificación por prioridad

Adaptación (ver referencias)

Planificación por prioridad

- En planificación apropiativa
 - Si prioridad de proceso en CPU es mayor a proceso que llega a cola de listo, se expulsa proceso en CPU y se ejecuta proceso recién llegado.
 - Proceso expulsado pasa a la cola de listo.
- En planificación no apropiativa
 - Proceso recién llegado (de menor prioridad) se pone de primero en la cola de listo pero no se expulsa al que está en CPU.

Procesos	Tiempo requerido en CPU	Prioridad
P_1	10 ms	3
P_2	1 ms	1
P_3	2 ms	4
P_4	1 ms	5
P_5	5 ms	2

Tiempo espera $P_1 = 6$ ms
 Tiempo espera $P_2 = 0$ ms
 Tiempo espera $P_3 = 16$ ms
 Tiempo espera $P_4 = 18$ ms
 Tiempo espera $P_5 = 1$ ms
 Tiempo espera promedio = 8.2 ms

- Procesos llegan todos en t_0 en orden: P_1, P_2 , etc.



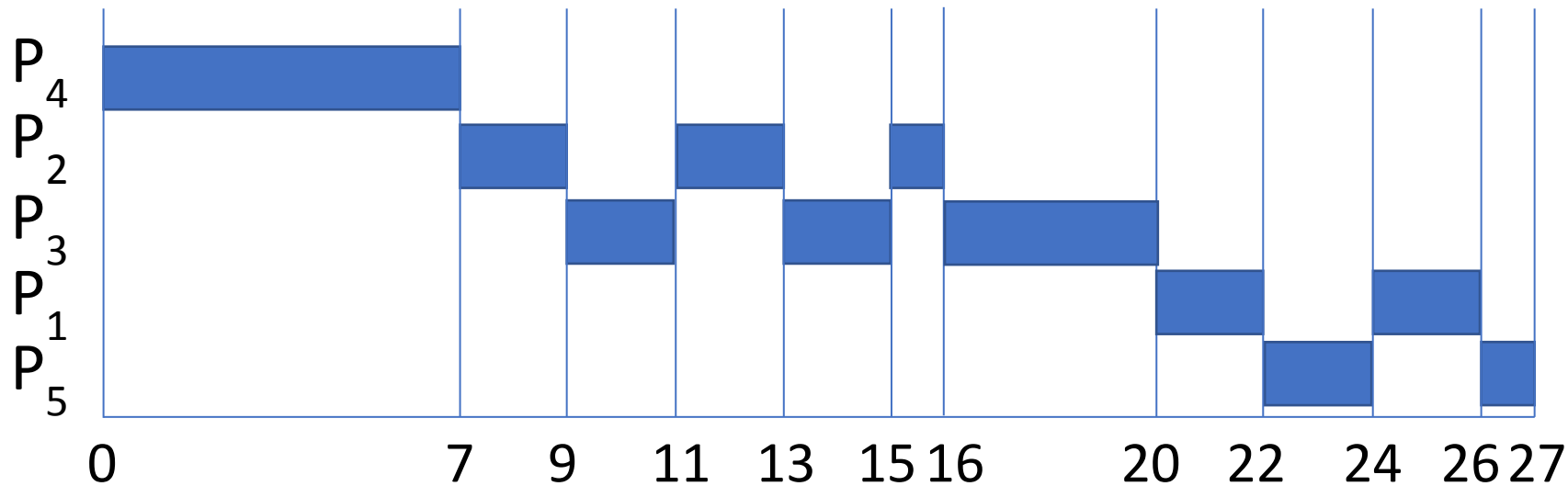
Planificación por prioridad

- Suceden problemas de inanición
 - Procesos de menor prioridad se pueden quedar esperando indefinidamente la CPU si siempre están llegando procesos de mayor prioridad
- Posible solución
 - Aumentar la prioridad de un proceso conforme pasa el tiempo en cola de listo.
 - Por ejemplo, aumentar la prioridad en uno cada segundo que el proceso pasa en espera en la cola de listo.
- Otra solución: cíclica + prioridad
 - Se ejecutan primero procesos de mayor prioridad
 - Procesos de misma prioridad se ejecutan de manera cíclica

Procesos	Tiempo requerido en CPU	Prioridad
P_1	4 ms	3
P_2	5 ms	2
P_3	8 ms	2
P_4	7 ms	1
P_5	3 ms	3

Tiempo espera $P_1 = 20 + 2 = 22$ ms
 Tiempo espera $P_2 = 7 + 2 + 2 = 11$ ms
 Tiempo espera $P_3 = 9 + 2 + 1 = 12$ ms
 Tiempo espera $P_4 = 0$ ms
 Tiempo espera $P_5 = 22 + 2 = 24$ ms
 Tiempo espera promedio = 13.8 ms

- Procesos llegan todos en t_0 en orden: P_1, P_2 , etc.
- Quantum = 2 ms



Referencias

- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). CPU Scheduling. In *Operating Systems Concepts* (10th ed., pp. 207–209). John Wiley & Sons, Inc.

Planificación en colas de múltiples niveles

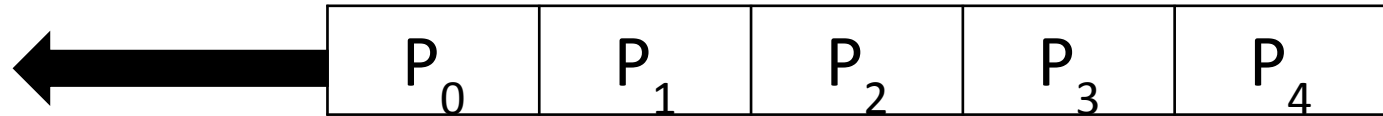
Adaptación (ver referencias)

Planificación en colas de múltiples niveles

- En planificación cíclica + prioridad hay una sola cola.
 - Hay que buscar siempre el proceso de menor prioridad en la cola
 - Implica algoritmo de búsqueda
- Una solución es tener varias colas para cada prioridad (POSIX)
 - Planificador siempre planifica proceso de la cola de prioridad más alta
- Se puede combinar con planificación cíclica
 - Procesos en la misma cola de prioridad se atienden mediante RR
 - Proceso se mantiene en la misma cola de prioridad
 - Las prioridades se asignan de manera estática: no hay manera de mover procesos otras colas de prioridad diferente
- Primero se atienden todos los procesos de la cola de mayor prioridad.

Planificación en colas de múltiples niveles

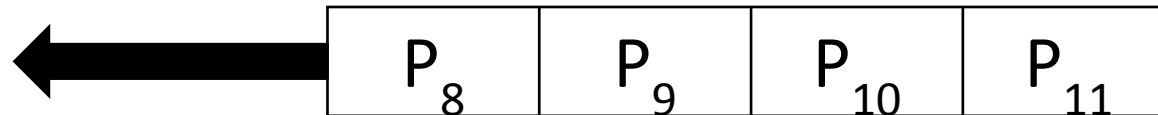
Cola de prioridad **0**



Cola de prioridad **1**



Cola de prioridad **2**

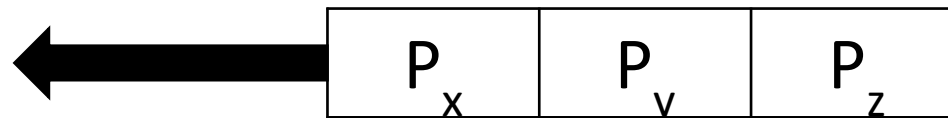


.

.

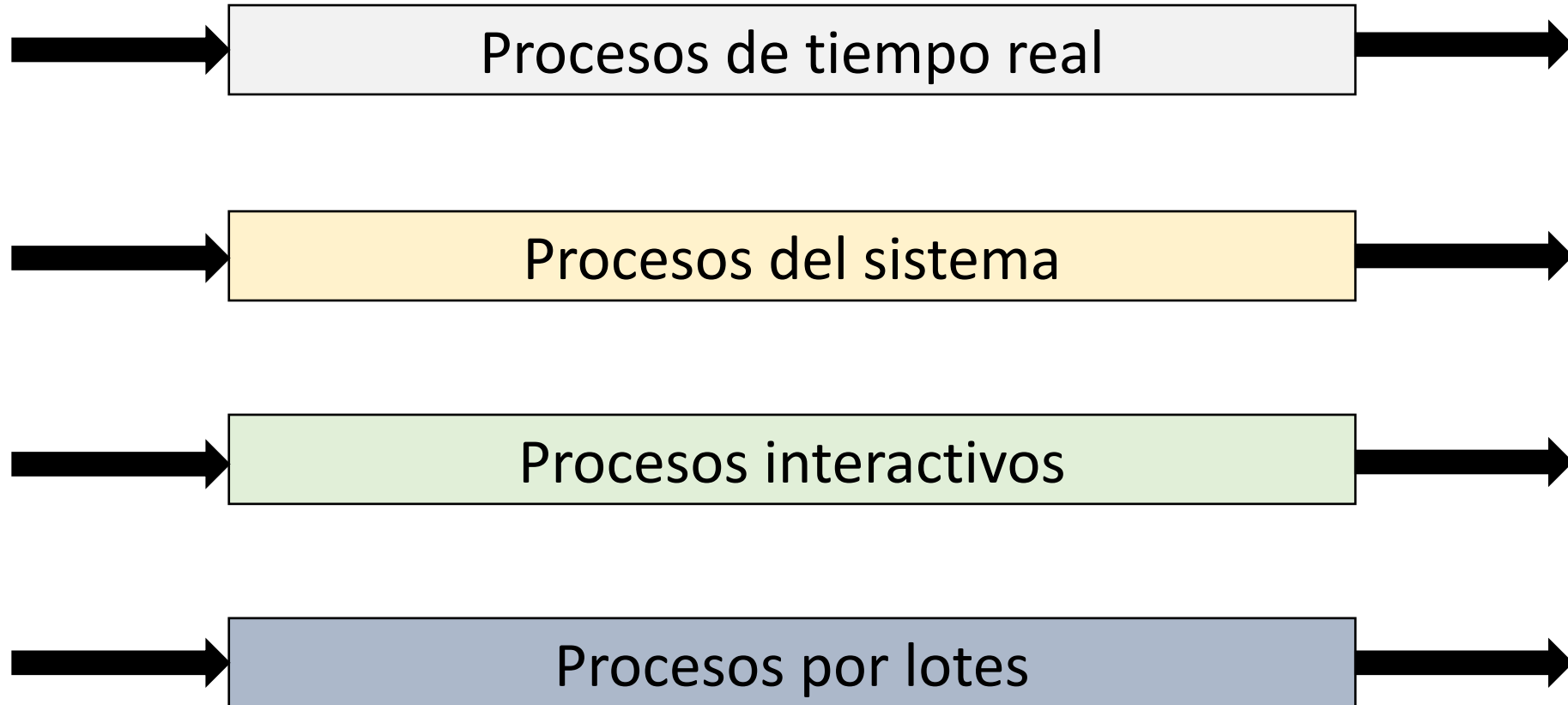
.

Cola de prioridad **n**



Planificación en colas de múltiples niveles

Prioridad más alta



Prioridad más baja

Planificación en colas de múltiples niveles

- Múltiples colas permiten organizar los procesos según sus demandas de CPU y necesidades de tiempos de respuesta.
 - Procesos del usuario: procesos interactivos con ráfagas cortas de CPU
 - Procesos en lote: procesos no interactivos
- Cada cola puede tener su propia disciplina (algoritmo)
 - P. Ej.: Procesos interactivos se atienden con planificación cíclica
 - P. Ej.: Procesos en lote se atienden con planificación FCFS

Planificación en colas de múltiples niveles

- Cada cola tiene un número de prioridad absoluta
 - Procesos en lote no se pueden ejecutar hasta que cola de proceso interactiva no esté vacía.
 - Proceso en lote que se esté ejecutando es expulsado de CPU si llega un proceso interactivo.
- También se puede asignar una franja de tiempo entre las diferentes colas
 - P. Ej.: cola de procesos interactivos tiene 80% del tiempo de CPU en planificación cíclica.
 - P. Ej.: cola de procesos en lote tiene 20% del tiempo de CPU en planificación FCFS.

Referencias

- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). CPU Scheduling. In *Operating Systems Concepts* (10th ed., pp. 207–209). John Wiley & Sons, Inc.

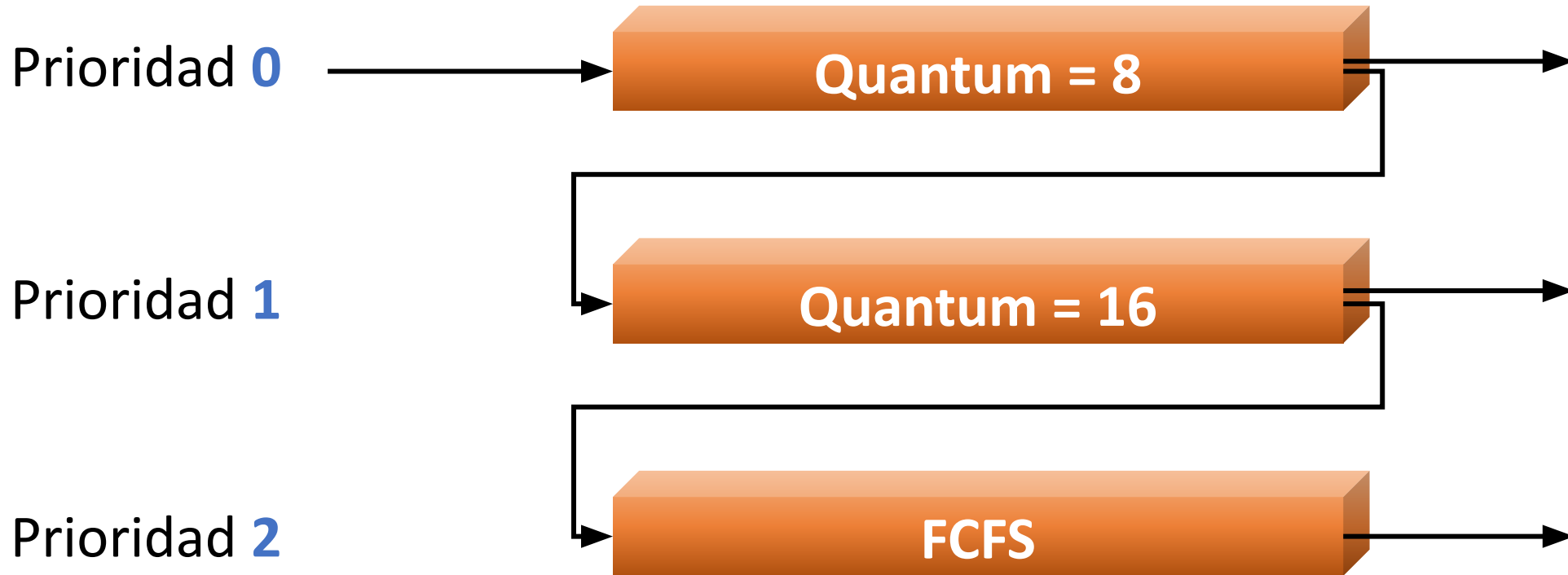
Planificación en colas de múltiples niveles con *feedback*

Adaptación (ver referencias)

Planificación en colas de múltiples niveles con *feedback*

- En planificación en colas de múltiples niveles procesos permanecen en la misma cola.
 - Procesos no se mueven a otras colas bajo ninguna circunstancia
- En planificación con *feedback* los procesos se pueden mover a otras colas
 - P. Ej.: si un proceso usa la CPU mucho tiempo, se mueve a cola de menor prioridad
 - Prioriza procesos interactivos: ráfagas cortas de CPU y ráfagas cortas de E/S
 - Procesos que esperan mucho en colas de baja prioridad se mueven a colas de mayor prioridad

Planificación en colas de múltiples niveles con *feedback*



Planificación en colas de múltiples niveles con *feedback*

- Planificador atiende a todos los procesos de cola de prioridad 0
- Cuando la cola de prioridad 0 está vacía, pasa a la cola de prioridad 1
- Cuando la cola de prioridad 1 está vacía, pasa a la cola de prioridad 2
- Un proceso de prioridad 2 que tenga la CPU puede ser expulsado de la CPU si llega un proceso de prioridad 1
- Procesos llegan a cola de prioridad 0, si no terminan en 8 ms, pasan a la cola de la prioridad 1.
- Si la cola 0 está vacía, se ejecuta el primer proceso de la cola 1 durante 16 ms, si no termina, se mueve a cola de prioridad 2
- Procesos que esperan mucho tiempo en cola pueden moverse a colas de mayor prioridad

Planificación en colas de múltiples niveles con *feedback*

- Parámetros del algoritmo
 - Número de colas
 - Disciplina de cada cola
 - Método o criterio para mejorar la prioridad de un proceso
 - Método o criterio para empeorar la prioridad de un proceso
 - Método o criterio para determinar en qué cola se pone un proceso que recién llega al sistema
- De difícil implementación
 - Se requiere de medidas previas para definir las mejores disciplinas de colas, rodajas de tiempo, etc.

Referencias

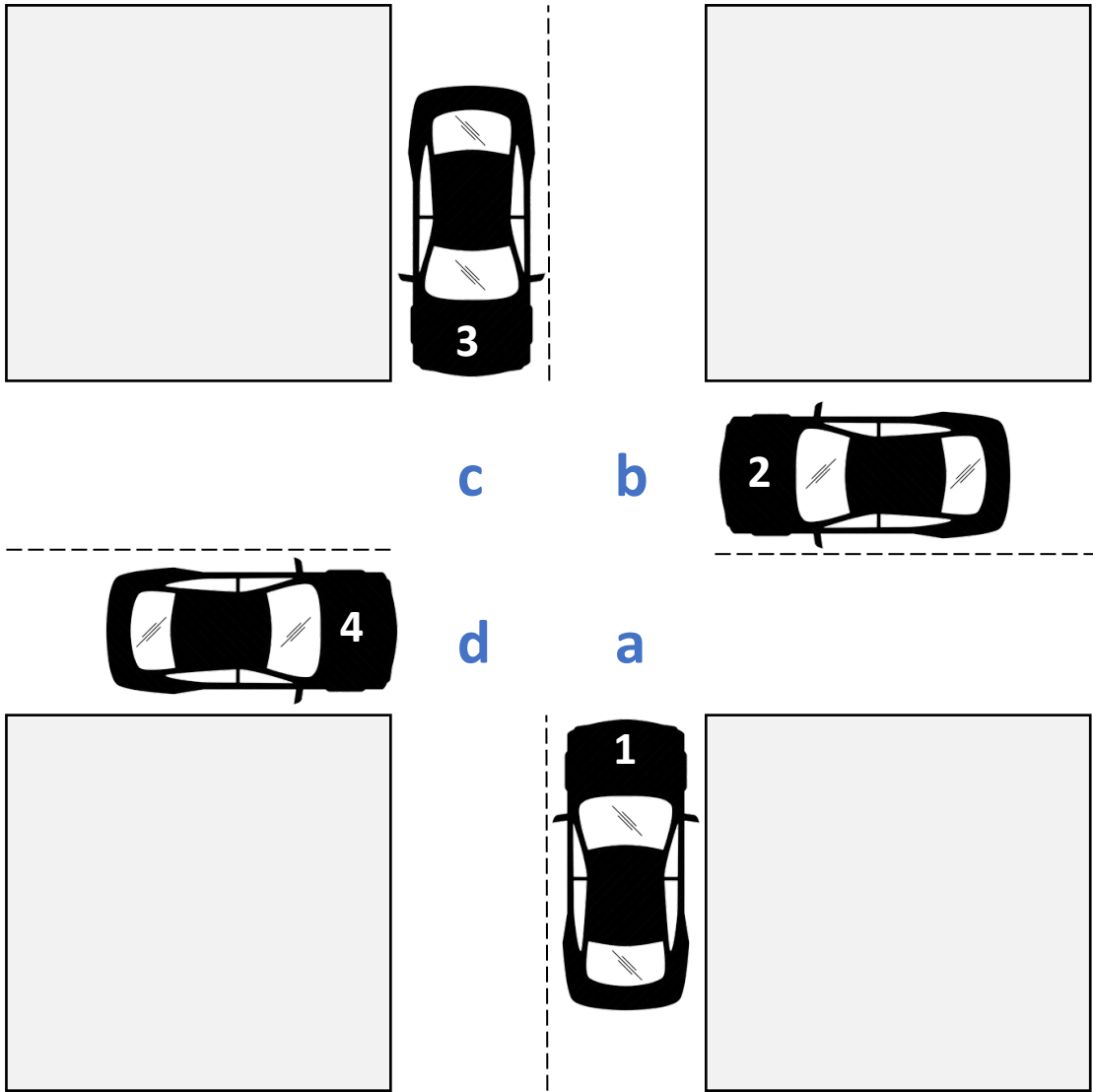
- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). CPU Scheduling. In *Operating Systems Concepts* (10th ed., pp. 207–209). John Wiley & Sons, Inc.

Interbloqueos

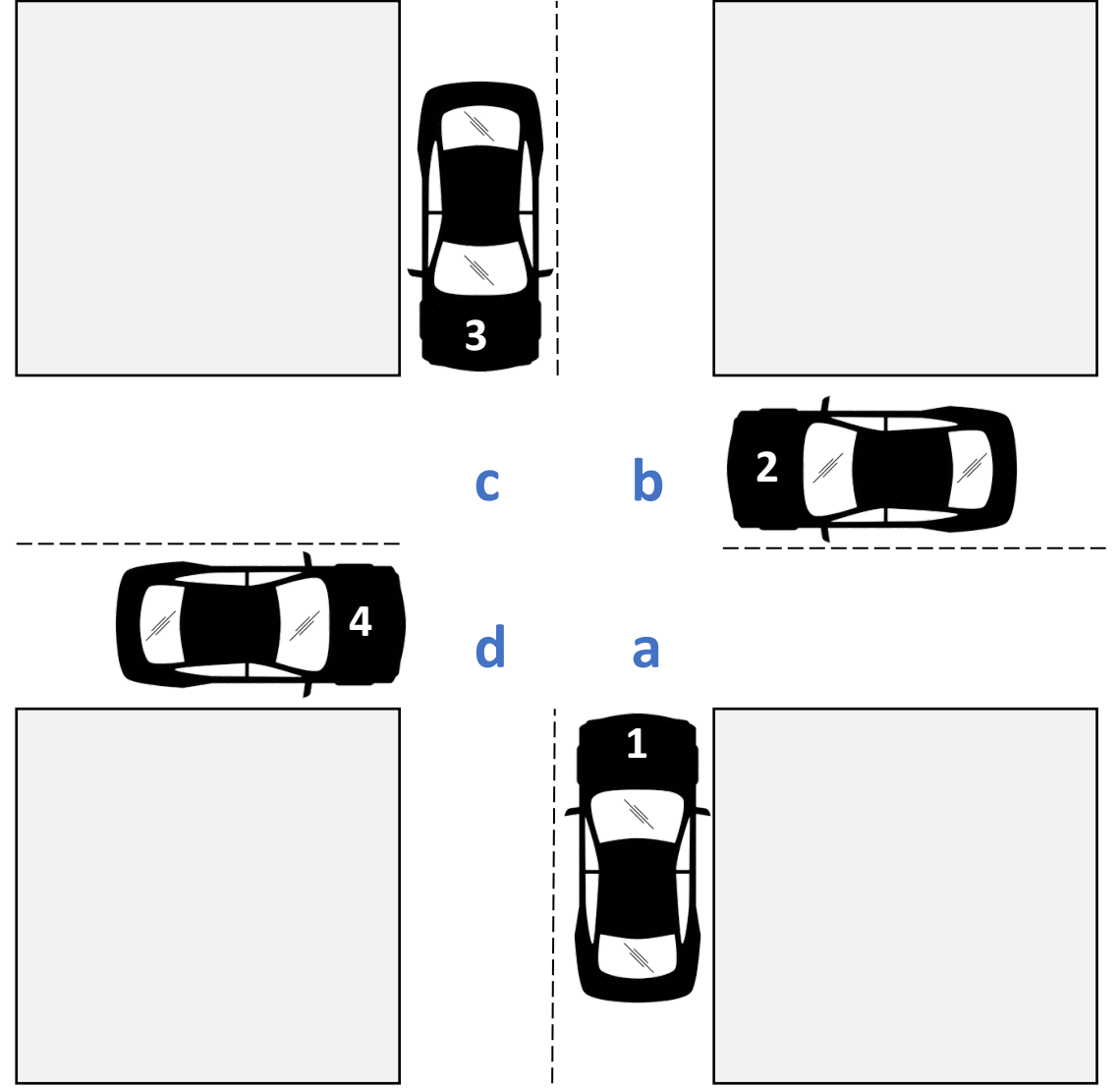
Adaptación de múltiples referencias (ver al final)

Interbloqueos

- Bloqueo permanente de
 - Procesos/hilos que compiten por recursos del sistema
 - Procesos/hilos que se comunican entre sí
- Un conjunto de procesos está en interbloqueo cuando cada proceso del conjunto está bloqueado esperando un evento que solo puede darse por uno de los procesos en el conjunto.
- El S.O no ofrece mecanismos para prevenir interbloqueos
 - Es responsabilidad de programadores evitar esta situación
 - Cómo se adquieren y se liberan recursos
- Problemas difíciles de resolver en entornos de alta concurrencia y paralelismo



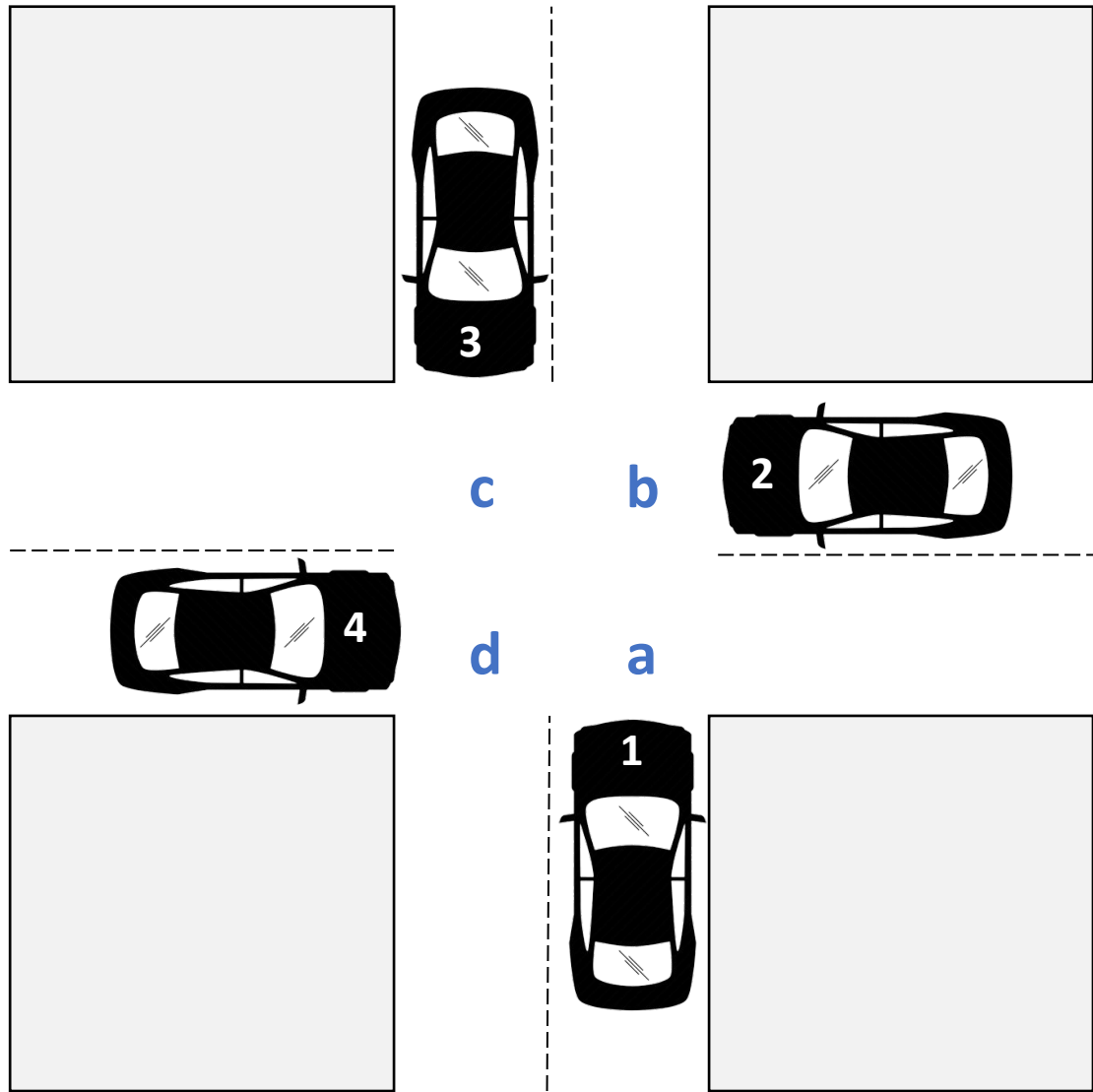
Posible interbloqueo



Interbloqueo

Interbloqueos

- Las intersecciones: **a**, **b**, **c** y **d** son recursos sobre los que se necesita control.
 - Vehículo 1 (norte): necesita intersecciones **a** y **b**
 - Vehículo 2 (oeste): necesita intersecciones **b** y **c**
 - Vehículo 3 (sur): necesita intersecciones **c** y **d**
 - Vehículo 4 (este): necesita intersecciones **d** y **a**



Interbloqueo

Todo el conjunto de procesos **(vehículos)** está esperando a que suceda un evento que solo puede ser generado por alguno de los procesos **(vehículo)** en el conjunto.

Interbloqueos

- S.O. mantiene una tabla para identificar si un recurso está disponible o está ocupado
- Para cada recurso ocupado se registra qué hilo/proceso que lo tiene asignado.
- Si un hilo/proceso solicita un recurso que está asignado a otro hilo
 - Hilo/proceso se añade a la cola de los que esperan por el mismo recurso
- Tipos de recursos que llevan al interbloqueos
 - *Mutex*, semáforos, archivos.
 - Mecanismos de IPC.
 - Recursos físicos como una tarjeta de red.

Estrategias para tratar el problema

- Detección y recuperación
 - Uno de los procesos/hilos debe liberar el recurso
 - En el caso de los vehículos: uno de ellos debe retroceder
 - Debe existir algún protocolo o conjunto de políticas que determinen cuál debe retroceder
- Prevención o predicción
 - Deben existir estrategias preventivas
 - En el caso de los vehículos: poner semáforos. Vehículo se detiene así el camino esté libre


```

pthread_mutex_t mutex1;
pthread_mutex_t mutex2;
pthread_mutex_init(&mutex1, NULL);
pthread_mutex_init(&mutex2, NULL);

void *hilo1(void *param) {
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);

    /* Región crítica */

    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}

```

```

void *hilo2(void *param) {
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);

    /* Región crítica */

    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);
    pthread_exit(0);
}

```

Se presenta interbloqueo en esta situación de ejecución paralela:

- **hilo1** obtiene **mutex1** y al mismo tiempo **hilo2** obtiene el **mutex2**

No se presenta interbloqueo si:

- **hilo1** libera **mutex1** y **mutex2** **ANTES** de que **hilo2** obtenga los *mutex*.

```

void *hilo1(void *param) {
    int done = 0;
    while (!done) {
        pthread_mutex_lock(&mutex1);
        if (pthread_mutex_trylock(&mutex2)) {

            /* Región crítica */

        }

        pthread_mutex_unlock(&mutex2);
        pthread_mutex_unlock(&mutex1);
        done = 1;
    }
    else
        pthread_mutex_unlock(&mutex1);
    }
    pthread_exit(0);
}

```

```

void *hilo2(void *param) {
    int done = 0;
    while (!done) {
        pthread_mutex_lock(&mutex2);
        if (pthread_mutex_trylock(&mutex1)) {

            /* Región crítica */

        }

        pthread_mutex_unlock(&mutex1);
        pthread_mutex_unlock(&mutex2);
        done = 1;
    }
    else
        pthread_mutex_unlock(&mutex2);
    }
    pthread_exit(0);
}

```

Se presenta situación de *livelock*, si:

- **hilo1** obtiene **mutex1** y al mismo tiempo **hilo2** obtiene el **mutex2**
- Cada hilo invoca (al tiempo) **pthread_mutex_trylock()**, lo cual falla haciendo que cada *mutex* se libere
- Esta secuencia se repite indefinidamente

No se presenta *livelock*, si:

- Se espera un tiempo aleatorio **ANTES** de intentarlo de nuevo.

Representación como grafo de asignación

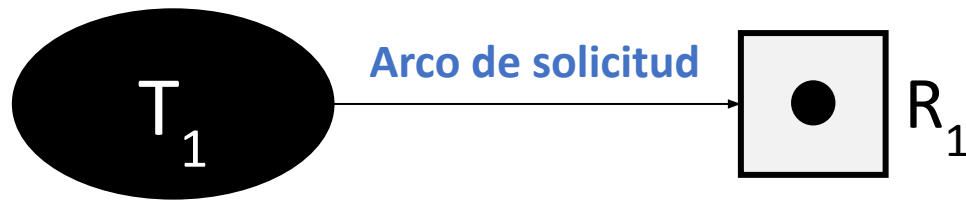
- Grafo de asignación de recursos en el sistema
 - Conjunto de vértices
 - Conjunto de arcos
- Conjuntos de vértices
 - Nodos que representan hilos/procesos activos en el sistema
$$T = \{T_1, T_2, \dots, T_n\}$$
 - Nodos que representan los tipos de recursos
$$R = \{R_1, R_2, \dots, R_m\}$$

Representación como grafo de asignación

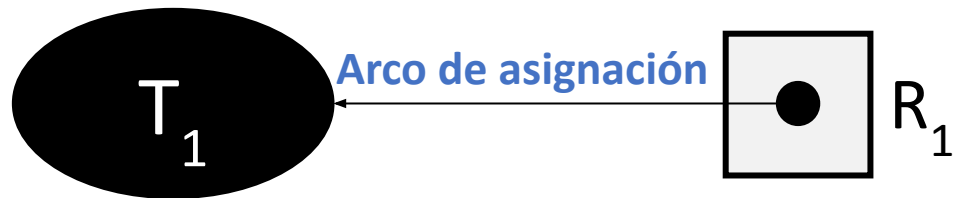
- Conjunto de arcos

- Arco dirigido desde T_i hasta R_j : $T_i \rightarrow R_j$
 - El hilo/proceso T_i solicitó **una instancia** de R_j y está esperando por el recurso.
 - Arco de solicitud
- Arco dirigido desde R_j hasta T_i : $R_j \rightarrow T_i$
 - **Una instancia** de R_j ha sido asignada al hilo/proceso T_i .
 - Arco de asignación

Representación como grafo de asignación

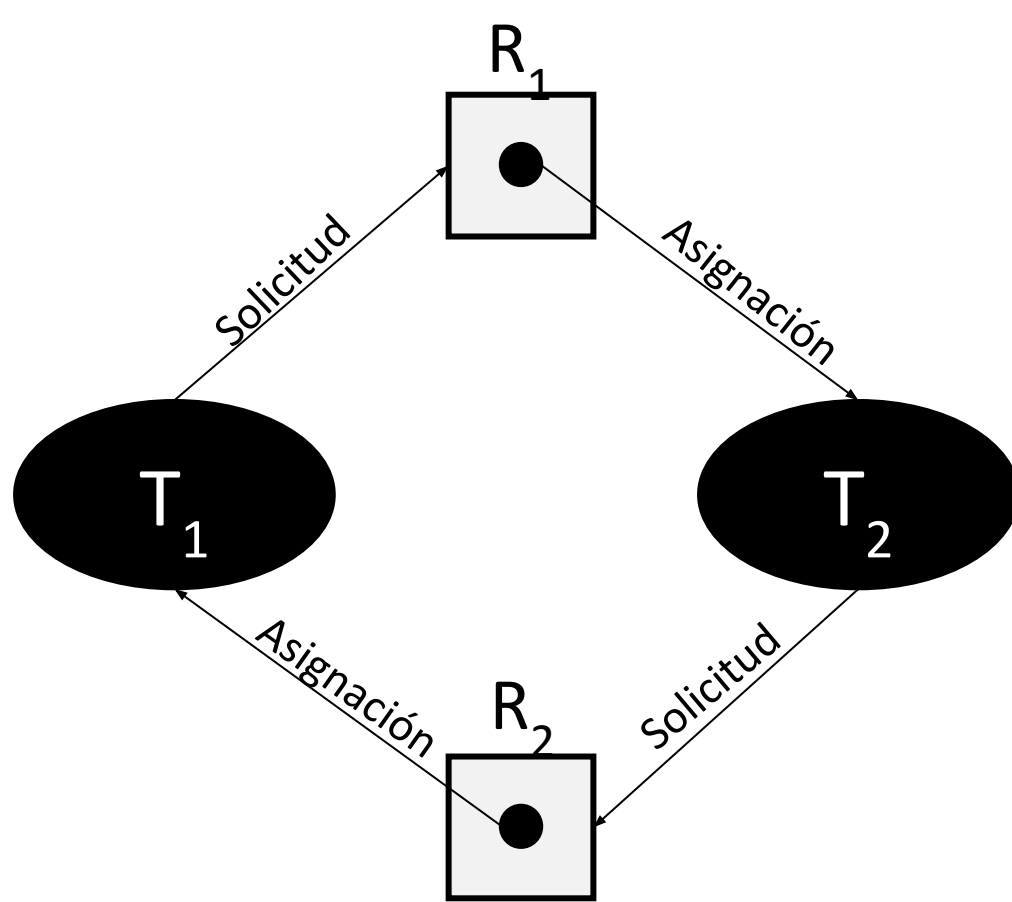


Indica que el recurso R_1 es solicitado por el proceso/hilo T_1

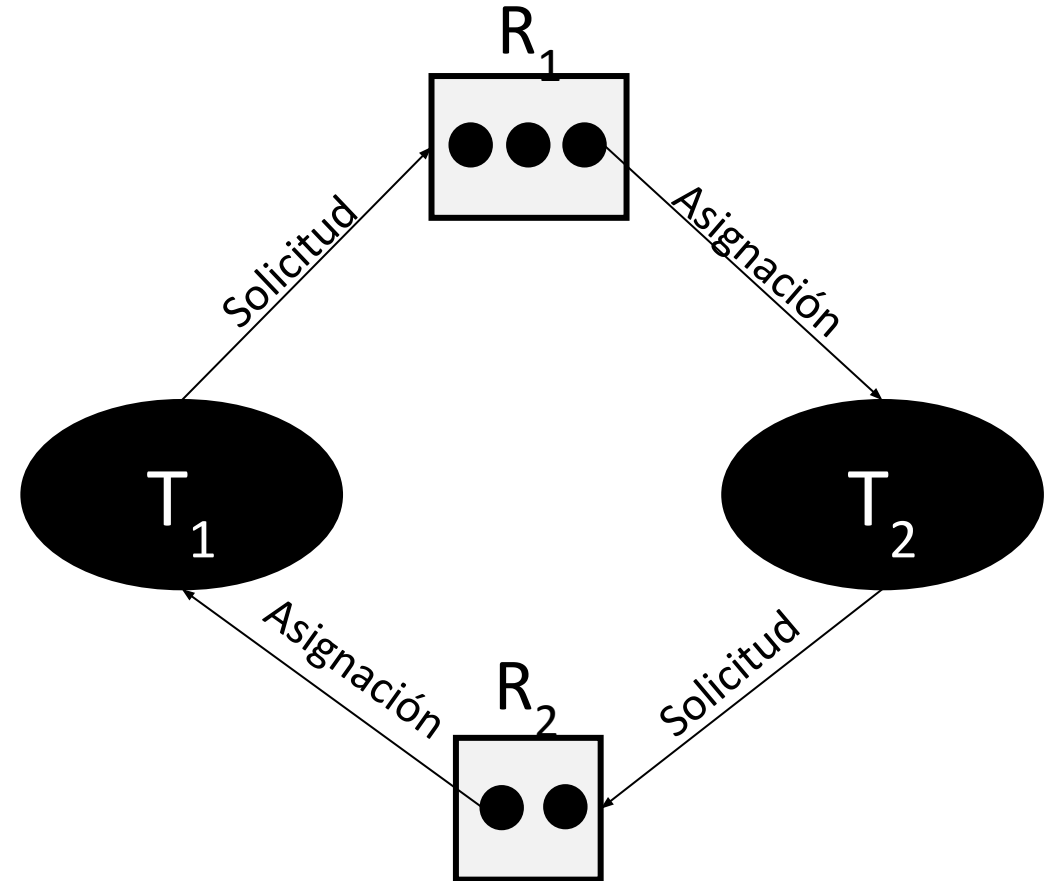


Indica que una instancia del recurso R_1 es asignada al proceso/hilo T_1

Representación como grafo de asignación



Espera circular: *deadlock*



No hay *deadlock*

Representación como grafo de asignación

- Cuando un hilo/proceso T_i solicita una instancia del recurso de tipo R_j
 - Se inserta un arco de solicitud en el grafo.
- Cuando se puede cumplir el requerimiento de T_i sobre la instancia del recurso de tipo R_j
 - El arco de solicitud se transforma en arco de asignación.
- Cuando el hilo/proceso no necesita el acceso al recurso, libera el recurso
 - El arco de asignación se elimina del grafo.

Representación como grafo de asignación

- Semántica de la representación
 - Círculos/óvalos: Procesos o hilos. Un círculo/óvalo por cada proceso/hilo.
 - Rectángulos: Recursos. Un rectángulo por cada tipo de recurso.
 - Puntos dentro de rectángulos: número de instancias disponibles de un recurso
- Restricción de asignación
 - Número de arcos que salen desde R_j debe ser menor o igual que su inventario (unidades disponibles)
- Restricción de solicitud
 - Por cada pareja T_i, R_j se debe cumplir que número de arcos desde R_j a T_i más el número de arcos de T_i a R_j debe ser menor o igual que el inventario.

Condiciones para un interbloqueo

1. Exclusión mutua

- Existe un recurso compartido que se está usando por un hilo/proceso a la vez.
- Los que necesiten el recurso deben esperar hasta que se libere.

2. Retención y espera

- Un hilo mantiene/bloquea un recurso compartido y a la vez espera a que se liberen otros recursos bloqueados por otros hilos.

3. No expropiación

- Los recursos son liberados a voluntad por quien los usa.

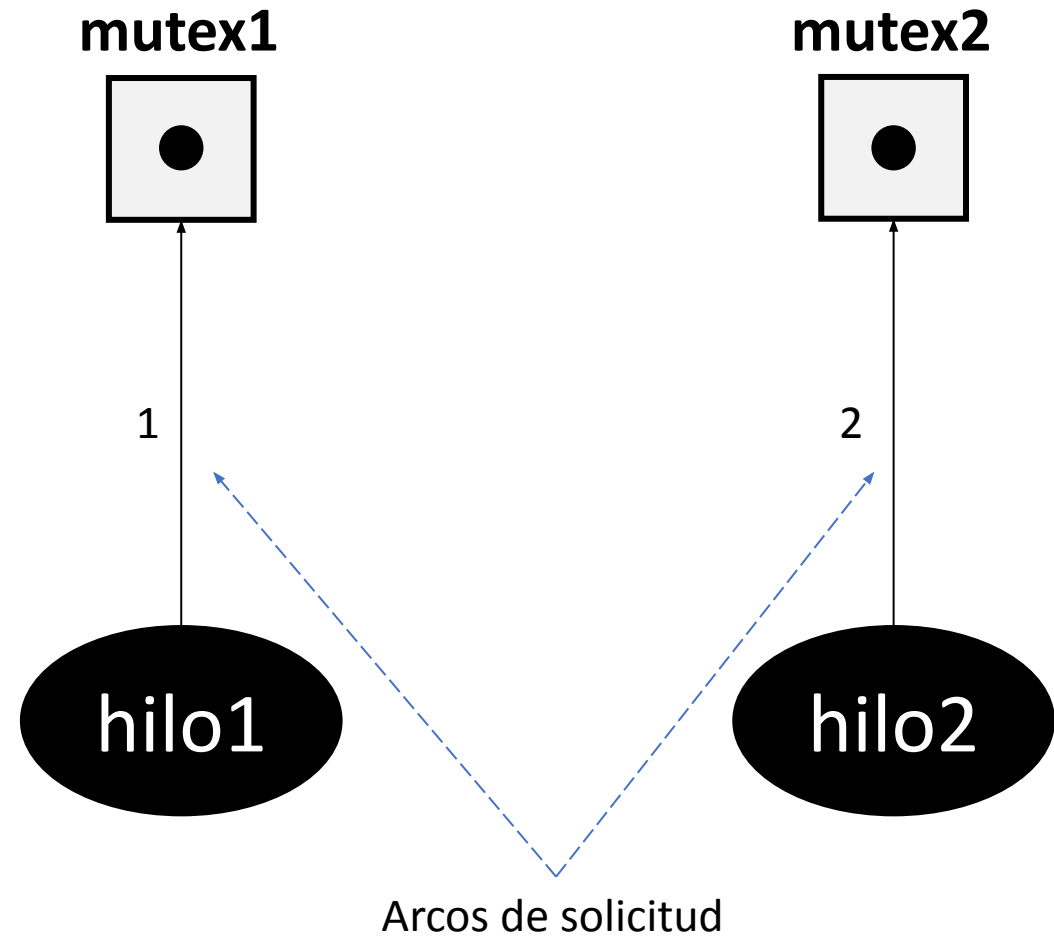
4. Espera circular

- T_1 espera por recurso asignado a T_2 y T_2 espera por un recurso asignado a T_1

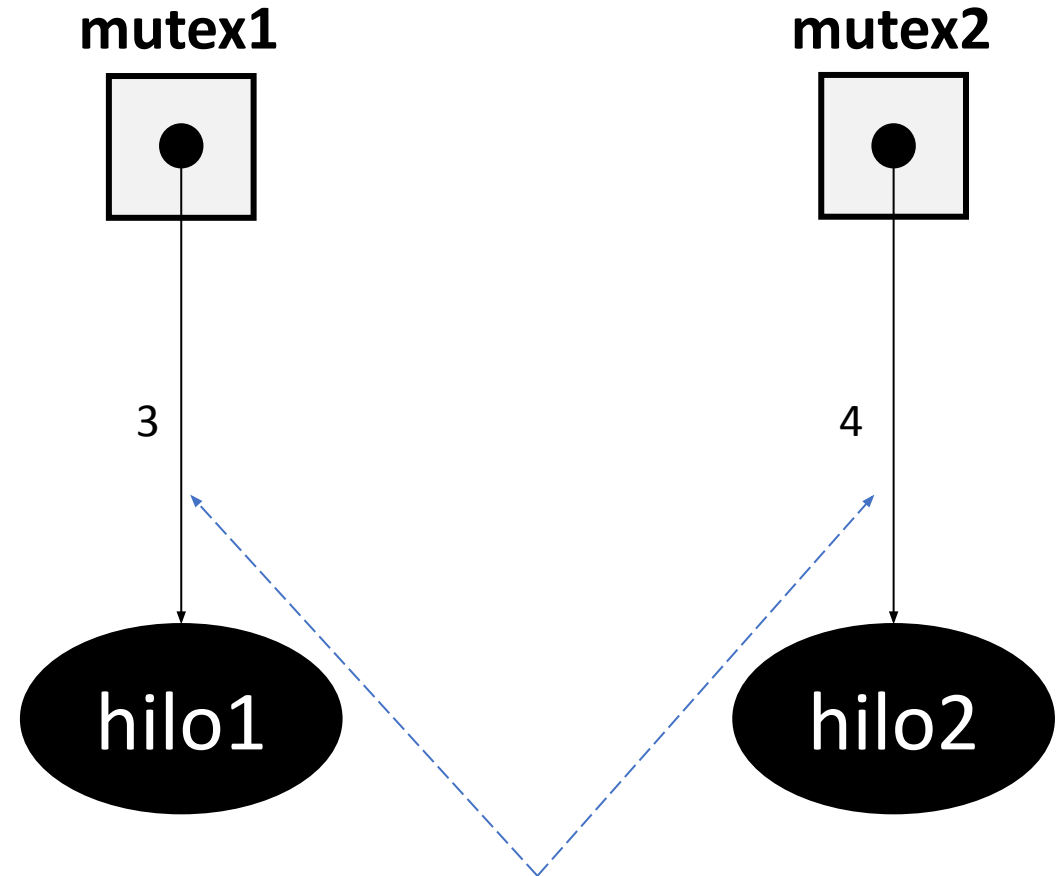
Representación como grafo de asignación

- Ejemplo con el código que produce *deadlock* con dos hilos
 1. Hilo1: solicita bloqueo de mutex1
 2. Hilo2: solicita bloqueo de mutex2
 3. Hilo1: bloquea mutex1
 4. Hilo2: bloquea mutex2
 5. Hilo1: solicita bloqueo de mutex2 ☐ Se bloquea esperando mutex2
 6. Hilo2: solicita bloqueo de mutex1 ☐ Se bloquea esperando mutex1

1. Hilo1: solicita bloqueo de mutex1
2. Hilo2: solicita bloqueo de mutex2

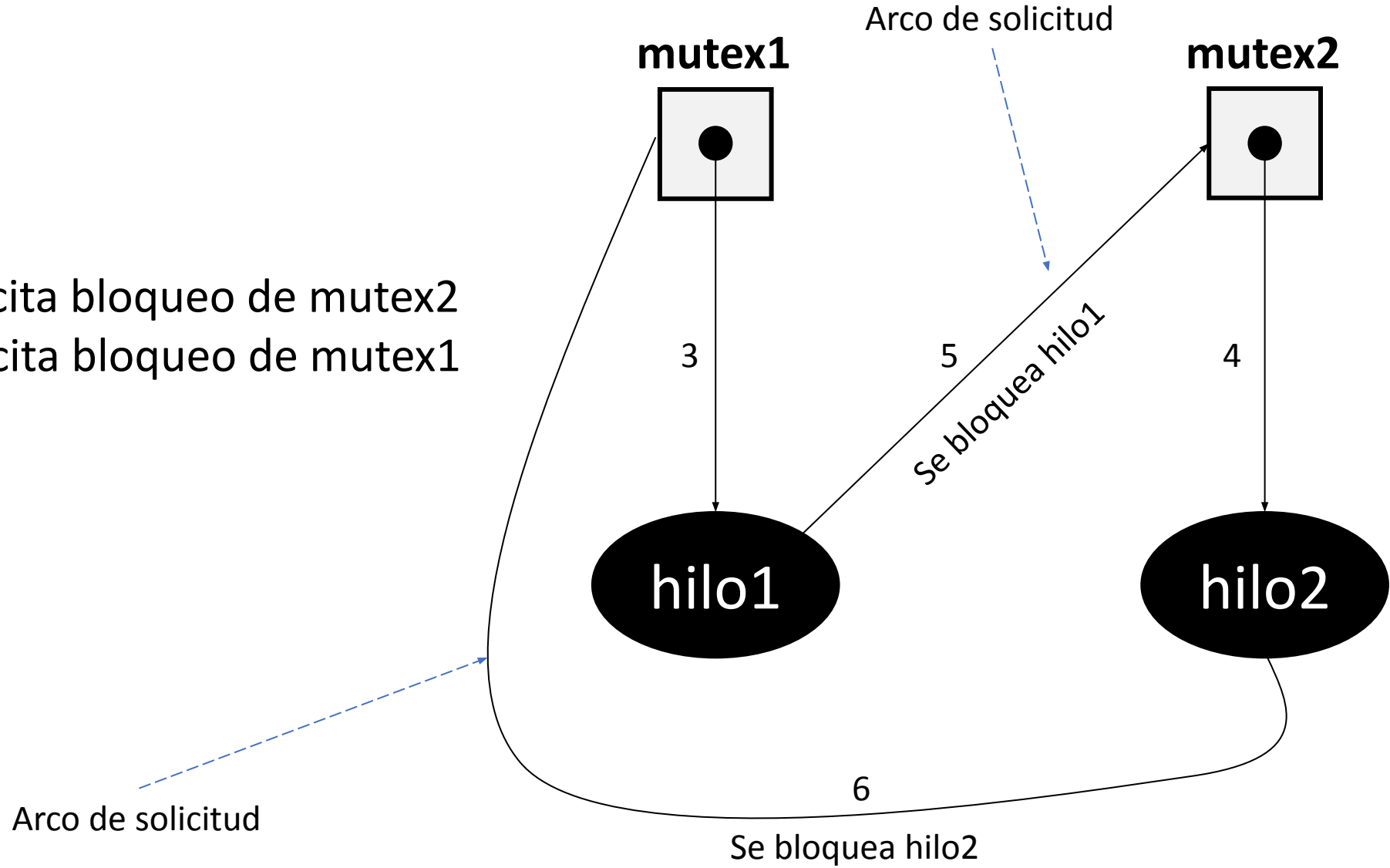


3. Hilo1: bloquea mutex1
4. Hilo2: bloquea mutex2



Arcos de asignación porque los recursos están disponibles. Note que salen desde el número de instancia de cada recurso

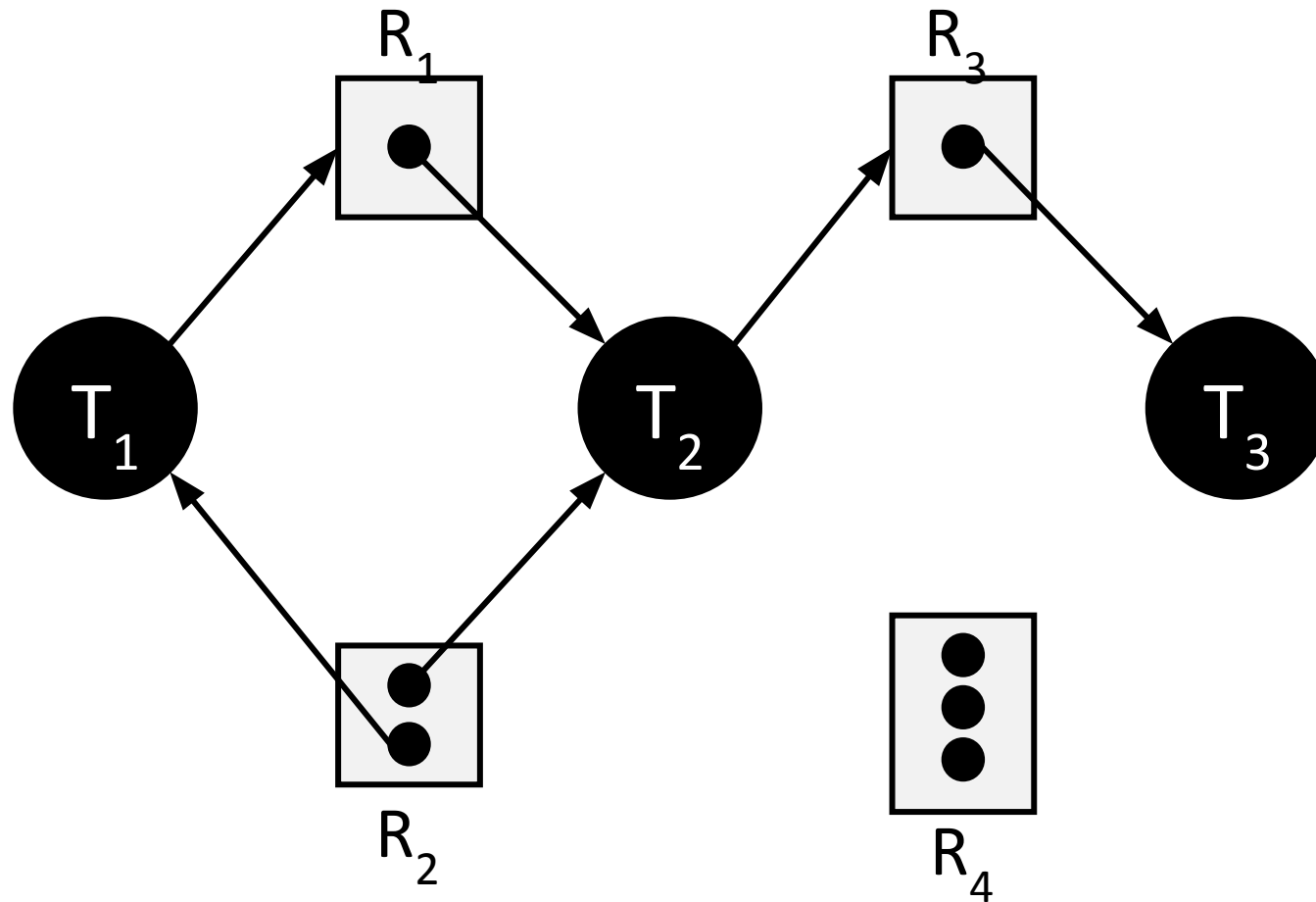
5. Hilo1: solicita bloqueo de mutex2
6. Hilo2: solicita bloqueo de mutex1



Representación como grafo de asignación

- Considere los siguientes conjuntos
 - $T = \{T_1, T_2, T_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
- Número de instancias
 - $1 \times R_1$
 - $2 \times R_2$
 - $1 \times R_3$
 - $3 \times R_4$

Representación como grafo de asignación



Recorridos

- $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3$
- $R_2 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3$

No hay ciclos: no hay interbloqueo

Representación como grafo de asignación

- Si grafo no tiene ciclos, entonces no hay interbloqueo.
- Si el grafo tiene un ciclo, **podría** existir interbloqueo.
- Si de cada R_j hay una sola instancia, entonces el ciclo implica que ocurrió interbloqueo.
- Si el ciclo involucra a únicamente a un conjunto de recursos, de los cuales solo hay una instancia, entonces ocurrió un interbloqueo.
 - Ciclo en el grafo: condición necesaria y suficiente para un interbloqueo
- Si existen varias instancias de un mismo recurso, un ciclo no necesariamente implica que ocurrió un interbloqueo
 - Ciclo en el grafo: condición necesaria pero no suficiente para un interbloqueo

Representación como grafo de asignación

- Considere los siguientes conjuntos

- $T = \{T_1, T_2, T_3\}$

- $R = \{R_1, R_2, R_3, R_4\}$

- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3, \mathbf{T_3 \rightarrow R_2}\}$

- Número de instancias

- $1 \times R_1$

- $2 \times R_2$

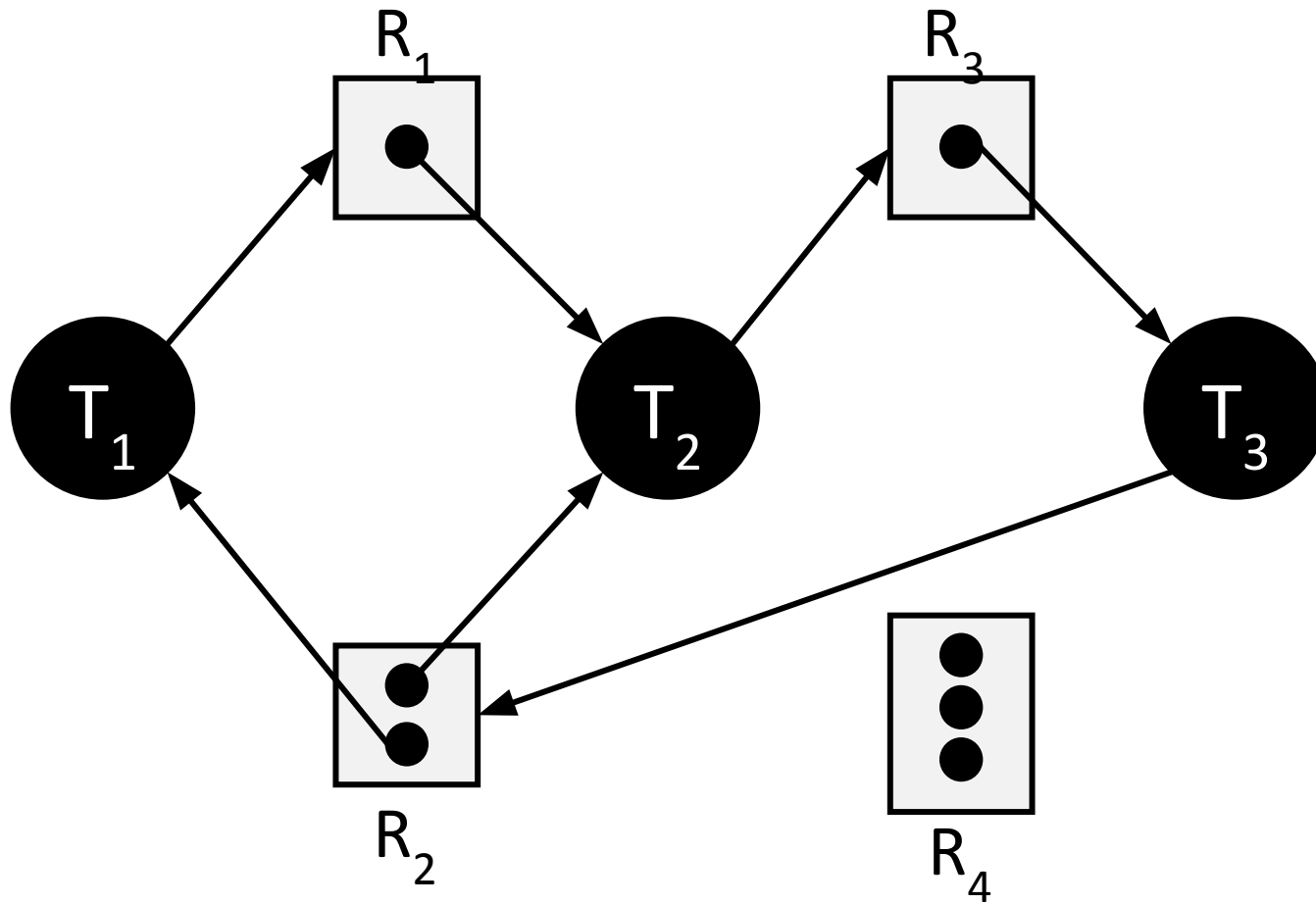
- $1 \times R_3$

- $3 \times R_4$

Se agregó esta solicitud



Representación como grafo de asignación



Primer ciclo

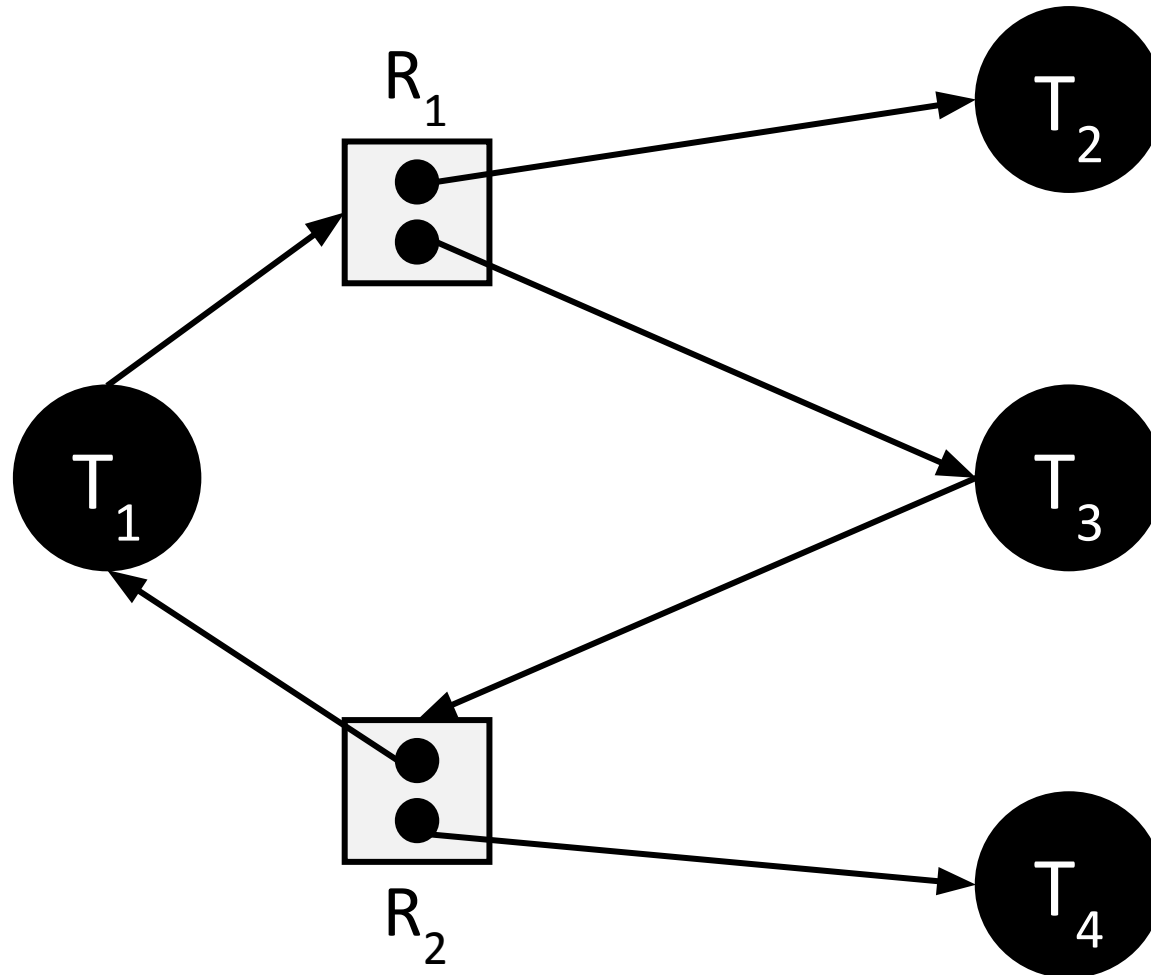
• $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$

Segundo ciclo

• $R_2 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2$

T_1, T_2, T_3 Están en interbloqueo

Representación como grafo de asignación



Recorridos

- $T_1 \rightarrow R_1 \rightarrow T_2$
- $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_4$

Existe un ciclo pero si T_4 libera voluntariamente R_2 , se rompe el ciclo.
Podría existir un interbloqueo pero no lo hay.

Estrategias para tratar el interbloqueo

- Ignorar el problema
 - Windows, Linux
- Diseñar y usar un protocolo para que nunca se entre en interbloqueo
 - Estrategias para prevenir interbloqueos
 - Estrategias para evitar interbloqueos
- Permitir entrar en interbloqueo, detectarlo y recuperar el sistema del interbloqueo
 - DBMS

Referencias

- Carretero Pérez, J., De Miguel Anasagasti, P., García Carballeira, F., & Pérez Costoya, F. (2001). Interbloqueos. In *Sistemas operativos. Una Visión Aplicada* (pp. 309–325). McGraw Hill.
- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). Deadlocks. In *Operating Systems Concepts* (10th ed., pp. 317–327). John Wiley & Sons, Inc.
- Stallings, W. (2018). Concurrency: Deadlock and Starvation. In *Operating Systems Internals and Design Principles* (9th ed., pp. 289–299). Pearson Education Limited.

Estrategias para detección y recuperación de interbloqueos

Adaptación (ver referencias al final)

Vector de recursos existentes

$E =$

R_1	R_2	R_3	R_4
4	2	3	1

Matriz de asignaciones actuales

$C =$

R_1	R_2	R_3	R_4
0	0	1	0
2	0	0	1
0	1	2	0

Vector de recursos disponibles

$A =$

R_1	R_2	R_3	R_4
2	1	0	0

Matriz de peticiones

$R =$

R_1	R_2	R_3	R_4
2	0	1	0
1	0	1	0
2	1	0	0

Algoritmo para detectar interbloqueos

- Cada recurso está asignado o está disponible

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

- Sean A y B dos vectores.
 - La relación $A \leq B$ indica que **cada** elemento de A es menor o igual al elemento correspondiente de B .
 - $A \leq B$ si y solo si $A_i \leq B_i$ para $1 \leq i \leq m$
 - m es total de recursos de tipo i .

Algoritmo para detectar interbloqueos

- C_{ij} es el número de instancias del recurso j que están actualmente asignadas al proceso P_i .
- C_{ij} número de instancias del recurso j que desea el proceso P_i .

Algoritmo para detectar interbloqueos

1. Buscar un proceso P_i tal que $R \leq A$
 - R la i -ésima fila de R (vector fila)
- En este ejemplo:
 - La fila $R_3 = [2, 1, 0, 0]$ cumple la condición
 - Proceso encontrado: P_3

Vector de recursos disponibles

R_1	R_2	R_3	R_4
2	1	0	0

$A =$

Matriz de peticiones

R_1	R_2	R_3	R_4
2	0	1	0
1	0	1	0
2	1	0	0

$R =$

Algoritmo para detectar interbloqueos

2. Si se encuentra el proceso, sumar la i -ésima fila de C a A

- En este punto P_3 se ejecuta y devuelve todos los recursos.
- Se marca P_3 como proceso completado.
- En este ejemplo:
 - La fila $C_3 = [0, 1, 2, 0]$
 - Se suma $C_3 + A$
 - Nuevo valor de $A = [2, 2, 2, 0]$

Vector de recursos disponibles

R_1	R_2	R_3	R_4
2	1	0	0

$A =$

Matriz de asignaciones actuales

R_1	R_2	R_3	R_4
0	0	1	0
2	0	0	1
0	1	2	0

$C =$

Algoritmo para detectar interbloqueos

1. Proceso se repite en busca de otro procesos P_i tal que $R \leq A$

- En este ejemplo:

- La fila $R_2 = [1, 0, 1, 0]$ cumple la condición. La primera también pero elegí esa.
- Proceso encontrado: P_2

Vector de recursos disponibles

R_1	R_2	R_3	R_4
2	2	2	0

$A =$

Matriz de peticiones

R_1	R_2	R_3	R_4
2	0	1	0
1	0	1	0
2	1	0	0

$R =$

Algoritmo para detectar interbloqueos

2. Si se encuentra el proceso, sumar la i -ésima fila de C a A

- En este punto P_2 se ejecuta y devuelve todos los recursos.
- Se marca P_2 como proceso completado.
- En este ejemplo:
 - La fila $C_2 = [2, 0, 0, 1]$
 - Se suma $C_2 + A$
 - Nuevo valor de $A = [4, 2, 2, 1]$

Vector de recursos disponibles

R_1	R_2	R_3	R_4
2	2	2	0

$A =$

Matriz de asignaciones actuales

R_1	R_2	R_3	R_4
0	0	1	0
2	0	0	1
0	1	2	0

$C =$

Algoritmo para detectar interbloqueos

1. Proceso se repite en busca de otro procesos P_i tal que $R \leq A$

- En este ejemplo:

- La fila $R_1 = [2, 0, 1, 0]$ cumple la condición.
- Proceso encontrado: P_1

Vector de recursos disponibles

R_1	R_2	R_3	R_4
4	2	2	1

$A =$

Matriz de peticiones

R_1	R_2	R_3	R_4
2	0	1	0
1	0	1	0
2	1	0	0

$R =$

Algoritmo para detectar interbloqueos

2. Si se encuentra el proceso, sumar la i -ésima fila de C a A

- En este punto P_1 se ejecuta y devuelve todos los recursos.
- Se marca P_1 como proceso completado.
- En este ejemplo:
 - La fila $C_1 = [0, 0, 1, 0]$
 - Se suma $C_1 + A$
 - Nuevo valor de $A = [4, 2, 3, 1]$

Vector de recursos disponibles

R_1	R_2	R_3	R_4
4	2	2	1

$A =$

Matriz de asignaciones actuales

R_1	R_2	R_3	R_4
0	0	1	0
2	0	0	1
0	1	2	0

$C =$

Algoritmo para detectar interbloqueos

- Todos los procesos se ejecutaron y el sistema no entró en interbloqueo.
 - Todos los procesos se marcaron como completados.
- Si al final de ejecutar el algoritmo, quedan procesos sin marcar, dichos procesos **están interbloqueos**.

Algoritmo para detectar interbloqueos

- Al principio todos los procesos están sin marcar
 1. Buscar un proceso desmarcado P_i para el que la i -ésima fila de R sea menor o igual A .
 2. Si P_i se encuentra, sumar la i -ésima fila de C a A , marcar el proceso y regresar al paso No. 1.
 3. Si P_i no se encuentra, el algoritmo termina y los proceso no marcados están en interbloqueo.

Recuperación de interbloqueo

- **Expropiación**

- Quitar temporalmente un recurso asignado a un proceso y entregarlo a otro proceso.
- No es un procedimiento simple.
- Depende del tipo de recurso.
- Se debe elegir correctamente cuál proceso se suspende para quitar con facilidad un recurso

Recuperación de interbloqueo

- **Retroceso**

- Establecer (guardar) puntos de comprobación de manera periódica.
- Guardar estados del sistema para poder devolverse en caso de un interbloqueo.
- Tomar una instantánea del sistema en un momento dado.
- Guardar la imagen de memoria
- Guardar el estado del recurso: qué recursos están asignados al proceso.
- Se mantiene las instantáneas (no sobrescribir)
- Detección de interbloqueo devolver a un estado anterior y reasignar recursos para evitar la situación de interbloqueo.

Recuperación de interbloqueo

- **Eliminar procesos**

- La estrategia más cruda y simple
- Romper el ciclo: eliminar procesos que estén en un ciclo de interbloqueo
- Elegir un proceso que se pueda eliminar sin efectos dañinos para el sistema
- P. Ej.: proceso que agrega un 1 a un registro en una BD (NO)

Referencias

- Tanenbaum, A. S. (2009). Detección y recuperación de un interbloqueo. In *Sistemas Operativos Modernos* (3rd ed., pp. 442–448). Pearson Educación.

Estrategias de prevención de interbloqueos

Adaptación (ver referencias al final)

Condiciones para un interbloqueo

1. Exclusión mutua

- Existe un recurso compartido que se está usando por un hilo/proceso a la vez.
- Los que necesiten el recurso deben esperar hasta que se libere.

2. Retención y espera

- Un hilo mantiene/bloquea un recurso compartido y a la vez espera a que se liberen otros recursos bloqueados por otros hilos.

3. No expropiación

- Los recursos son liberados a voluntad por quien los usa.

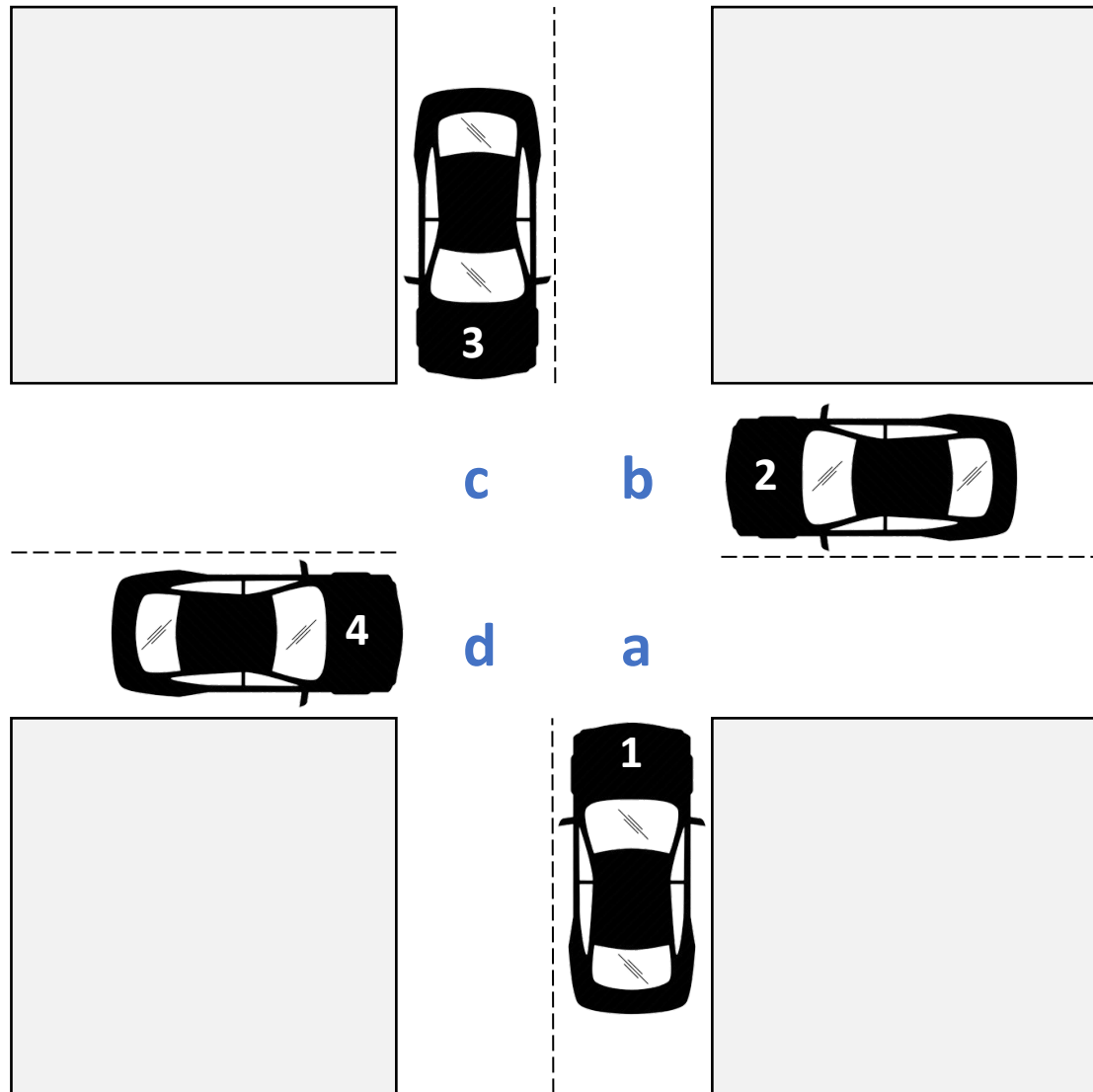
4. Espera circular

- T_1 espera por recurso asignado a T_2 y T_2 espera por un recurso asignado a T_1

Estrategias de prevención

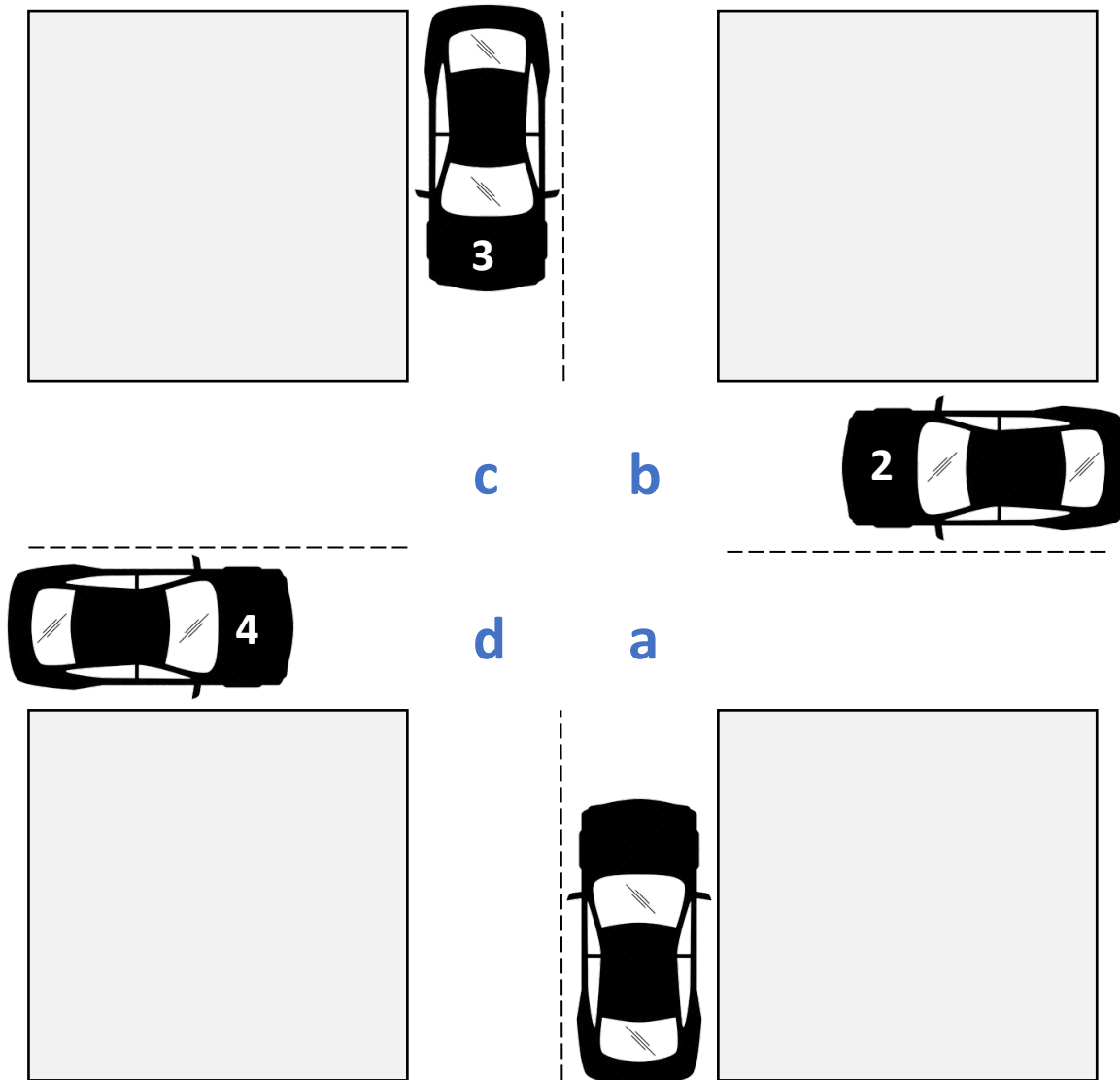
- Asegurarse que al menos una de las condiciones anteriores no se de para prevenir la ocurrencia de un interbloqueo.
- Eliminar alguna de las condiciones anteriores.

Exclusión mutua



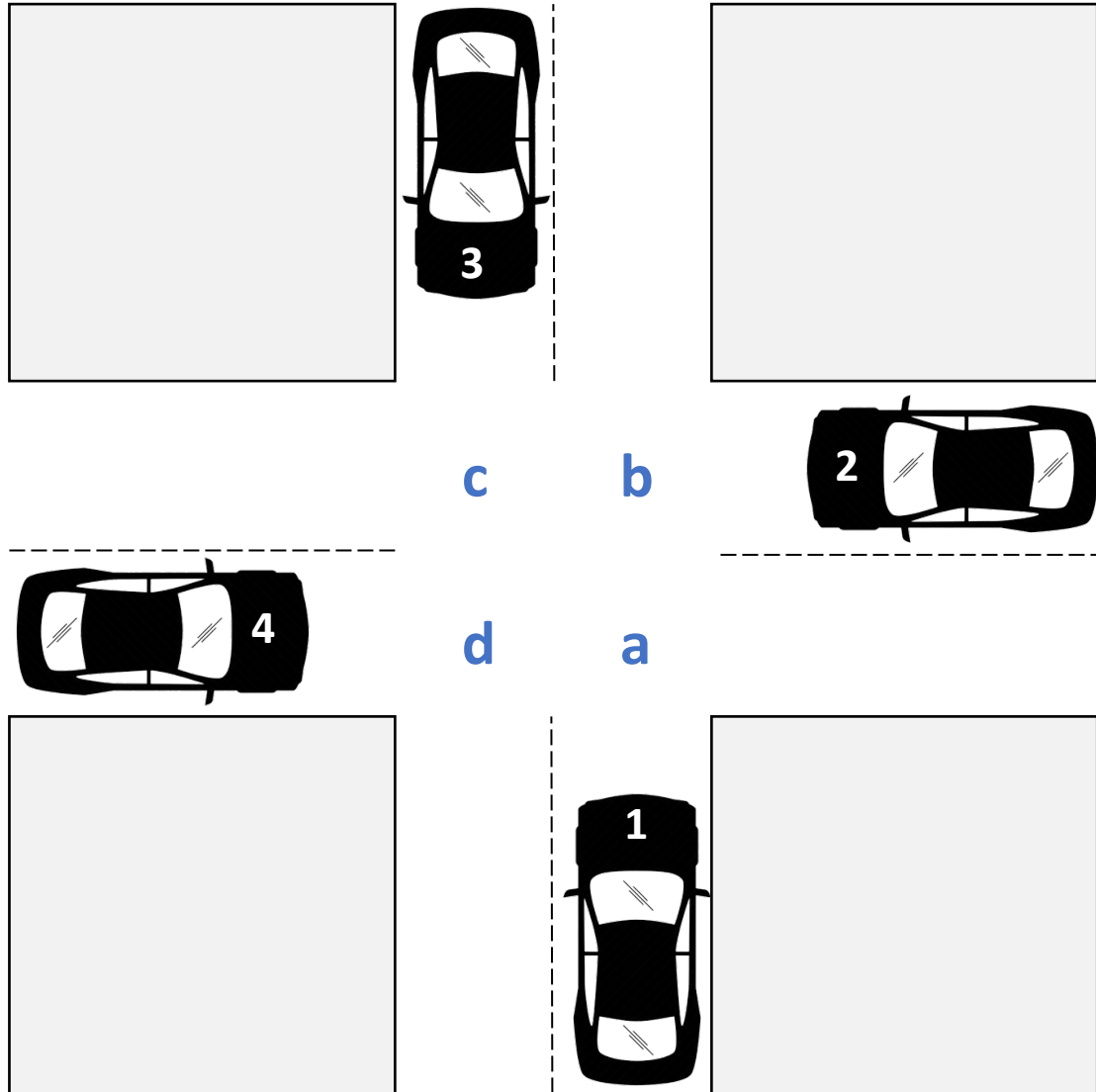
- La condición de exclusión mutua no se puede eliminar ya que algunos recursos son intrínsecamente no compartibles.
- Por ejemplo el bloqueo de un *mutex* no puede ser compartido por varios hilos/procesos.
- No todos los recursos compartidos tiene acceso en exclusión mutua.
 - P. Ej.: acceso a archivos de solo lectura.
- En este ejemplo tenemos cuatro recursos que son compartidos (**a**, **b**, **c** y **d**). Sin embargo, solo un vehículo a la vez puede usar uno de ellos.

Retención y espera



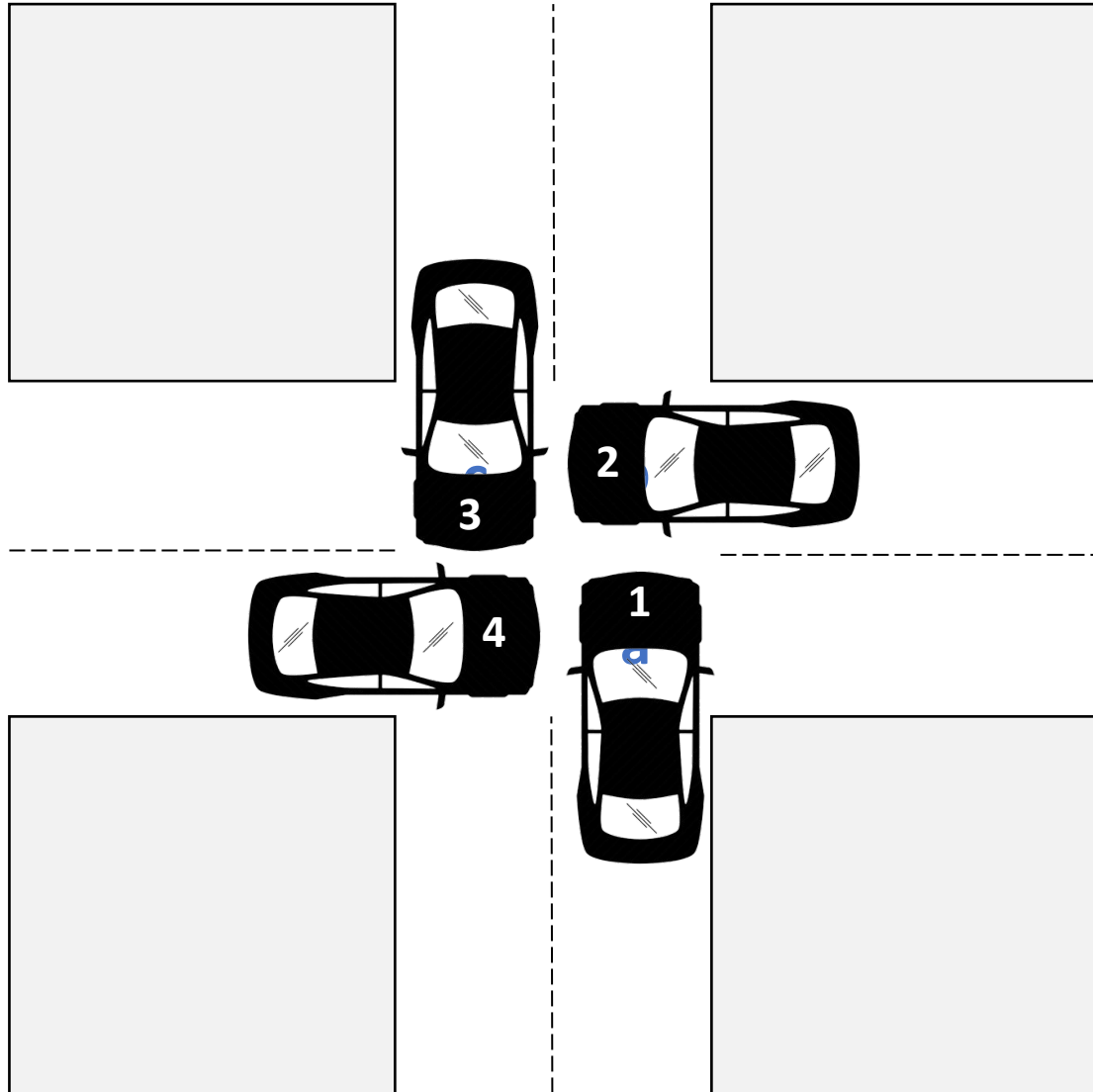
- Habría que asegurarse de que cuando un hilo/proceso solicite un recurso, éste no retenga ningún otro recurso.
- Solicitud de nuevo recursos: liberar recursos retenidos para intentar obtenerlos luego.
- Cada hilo/proceso solicita y se le asigna (**a priori**) todos los recursos que necesita para la ejecución. No se ejecuta hasta que no tenga todos los recursos.
- Vehículo No. 3 solicita con anterioridad intersecciones **c** y **d** libres.
- La intersección **c** queda sin poder usar por el vehículo No. 2.

No expropiación



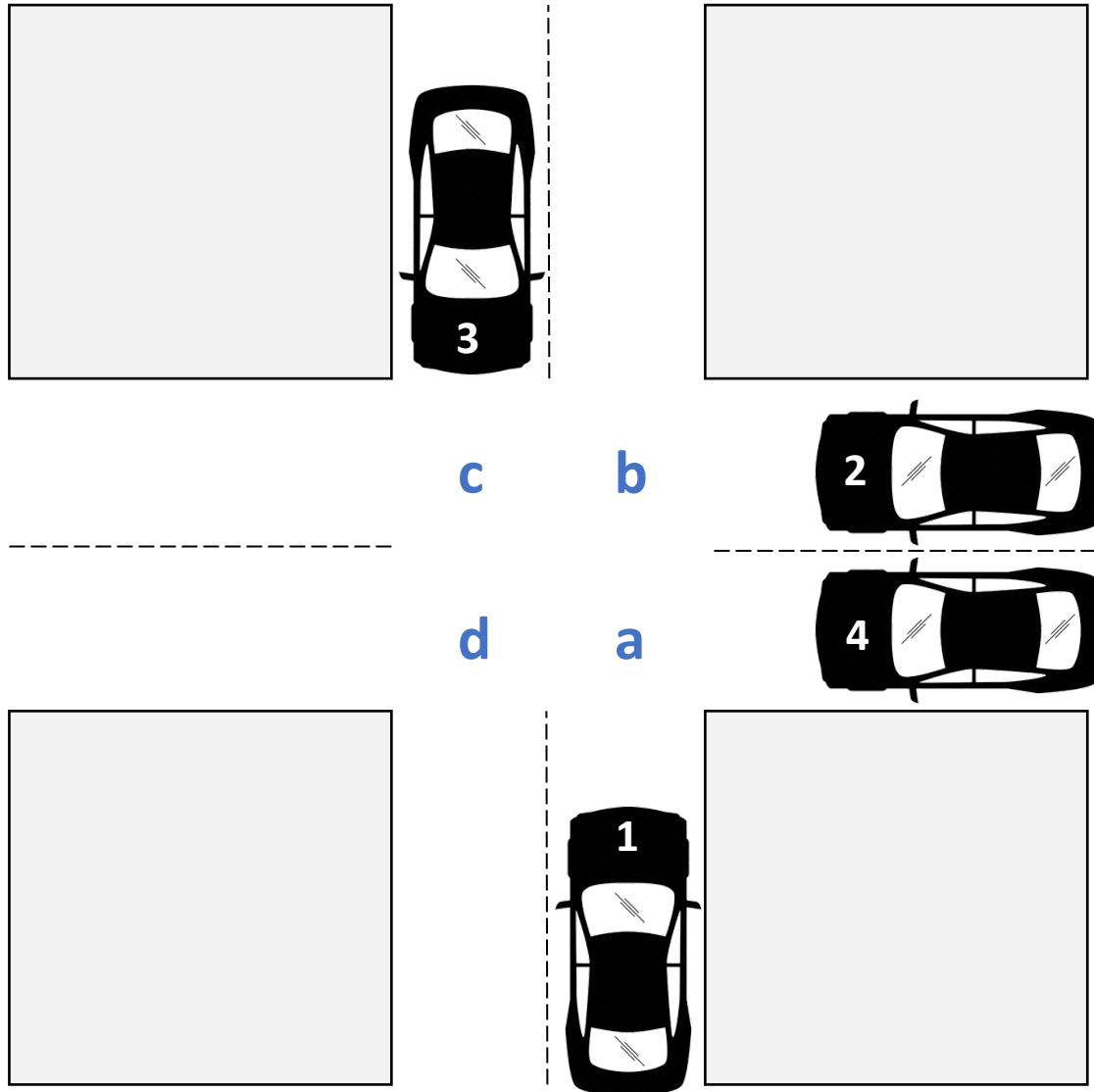
- Si un hilo/proceso solicita un recurso mientras mantiene otro, todos sus recursos se expropian.
- Recursos expropiados se añaden a recursos por los que espera el hilo/proceso.
- El hilo o proceso se reinicia cuando de nuevo tenga acceso los recursos que tenía más los nuevos que solicita.

No expropiación



- Se expropian los recursos del vehículo 3.
- Vehículo 3 queda a la espera de que las intersecciones **c** y **d** queden disponibles.
- Implica devolver el sistema a un estado anterior.
 - Guardar el estado anterior
 - Restaurar el estado anterior
 - Pueden existir muchos vehículos atrás.
- Registros de CPU, registros de BD, transacciones de BD

Espera circular



- Definir un ordenamiento lineal de los recursos
 - Solicitar **a** antes de **b**
 - Solicitar **b** antes de **c**
 - Solicitar **c** antes de **d**
- Una vez obtenido el recurso, solo los que siguen en orden pueden obtenerse.
- Usar solo un recurso en cualquier momento, si necesita un segundo recurso, liberar el primero.

Estrategias de prevención

- Consiste en diseñar protocolos para la solicitud y la asignación de recursos
- Efectos secundarios
 - Baja utilización de los recursos.
 - Recursos que se quedan sin asignar durante largos períodos de tiempo.
 - Reducción de *Throughput* del sistema.

Referencias

- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). Deadlock prevention. In *Operating Systems Concepts* (10th ed., pp. 327–330). John Wiley & Sons, Inc.
- Tanenbaum, A. S. (2009). Cómo prevenir interbloqueos. In *Sistemas Operativos Modernos* (3rd ed., pp. 454–457). Pearson Educación.

Estrategias para evitar interbloqueos

Algoritmo del banquero

Adaptación (ver referencias al final)

Evitar la ocurrencia de un interbloqueo

- En las estrategias de prevención
 - Diseñar protocolos para que no ocurran al menos una de las cuatro condiciones que llevan a un interbloqueo.
- En las estrategias para evitar entrar en un interbloqueo
 - Se necesita información previa sobre las peticiones futuras de recursos
 - Se analiza el sistema con la información de peticiones futuras
 - Si la petición futura lleva a un interbloqueo, no hay asignación de recursos

R_1	R_2	R_3
10	5	7

Inventario de recursos

Asignación actual

R_1	R_2	R_3
0	1	0
2	0	0
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
1	2	2
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
3	3	2

Estado del sistema en un momento dado

- **Asignación actual:** recursos asignados a cada T_i en este momento.
- **Máximo:** Máximo de recursos que serán solicitados por cada T_i
- **Necesario:** Recursos que necesita cada T_i para terminar. Máximo – Asignación actual.

R_1	R_2	R_3
10	5	7

Inventario de
recursos

Asignación actual

R_1	R_2	R_3
0	1	0
2	0	0
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
1	2	2
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
3	3	2

Llega una nueva solicitud de T_1

- Solicitud de $T_1 = [1, 0, 2]$
- ¿Solicitud de $T_1 \leq$ Disponible?

R_1	R_2	R_3
10	5	7

Inventario de recursos

Asignación actual

R_1	R_2	R_3
0	1	0
3	0	2
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
0	2	0
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
2	3	0

¿Se puede asignar la solicitud sin entrar en un estado **no seguro**?

- Supongamos que se realiza la asignación de recursos a $T_1 = [1, 0, 2]$. $[2, 0, 0] + [1, 0, 2] = [3, 0, 2]$
- La asignación se hace de lo que hay en Disponible: $[3, 3, 2] - [1, 0, 2] = [2, 3, 0]$

- Para responder a la pregunta es necesario verificar si con los recursos disponibles puedo hacer otra asignación de tal manera que algún T_i termine y devuelva los recursos asignados.
 - La idea es no quedarse sin recursos para permitir que otros T_i terminen y devuelvan lo asignado
- Después de la asignación, con lo que queda disponible se puede nuevamente asignar a T_1 todo lo que necesite para que termine y devuelva lo asignado.
 - Hacemos esta asignación...

R_1	R_2	R_3
10	5	7

Inventario de
recursos

Asignación actual

R_1	R_2	R_3
0	1	0
3	2	2
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	1	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
0	0	0
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
2	1	0

- Se hace la asignación a $T_1 = [0, 2, 0]$ para que termine: $[3, 0, 2] + [0, 2, 0] = [3, 2, 2]$
- Se resta Máximo – Asignación actual = Necesario:
 $[3, 2, 2] - [3, 2, 2] = [0, 0, 0]$

R_1	R_2	R_3
10	5	7

Inventario de recursos

Asignación actual

R_1	R_2	R_3
0	1	0
0	0	0
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
0	0	0
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
5	3	2

- T_1 ya no requiere más recursos y puede devolver lo asignado: [3, 2, 2] que se suma a lo disponible [2, 1, 0]: [5, 3, 2]
- Con lo disponible podemos ver que se puede satisfacer Necesario de T_3 para que termine: hacemos esta asignación.

R_1	R_2	R_3
10	5	7

Inventario de
recursos

Asignación actual

R_1	R_2	R_3
0	1	0
0	0	0
3	0	2
2	2	2
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
0	0	0
6	0	0
0	0	0
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
5	2	1

T_3 ya no requiere más recursos y puede devolver lo asignado: $[2, 2, 2]$ que se suma a lo disponible $[5, 2, 1] = [7, 4, 3]$

R_1	R_2	R_3
10	5	7

Inventario de
recursos

Asignación actual

T_0
T_1
T_2
T_3
T_4

R_1	R_2	R_3
0	1	0
0	0	0
3	0	2
0	0	0
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
0	0	0
6	0	0
0	0	0
4	3	1

Disponible

R_1	R_2	R_3
7	4	3

Con lo disponible se puede satisfacer a T_4 a T_0 y por último a T_2

- La solicitud $T_1 = [1, 0, 1]$ se puede satisfacer en el momento solicitado ya que la secuencia $\langle T_1, T_3, T_4, T_0, T_2 \rangle$ garantiza un estado seguro del sistema.
 - El sistema no se queda sin recursos para asignar y permitir que todos los T_i terminen y devuelvan lo asignado.

R_1	R_2	R_3
10	5	7

Inventario de
recursos

Asignación actual

R_1	R_2	R_3
0	1	0
3	0	2
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
0	2	0
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
2	3	0

Nos devolvemos al estado en el que se asignó $T_1 = [1, 0, 1]$
En donde el sistema tiene el estado indicado

- Se hace la solicitud $T_4 = [3, 3, 0]$
- ¿Solicitud $T_4 \leq$ Disponible ? No se puede satisfacer.

R_1	R_2	R_3
10	5	7

Inventario de
recursos

Asignación actual

R_1	R_2	R_3
0	1	0
3	0	2
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	4	3
0	2	0
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
2	3	0

Nos devolvemos al estado en el que se asignó $T_1 = [1, 0, 1]$

En donde el sistema tiene el estado indicado

- Se hace la solicitud $T_0 = [0, 2, 0]$
- ¿Solicitud $T_0 \leq$ Disponible? Si, pero analicemos qué pasa si se hace la asignación solicitada

R_1	R_2	R_3
10	5	7

Inventario de
recursos

Asignación actual

R_1	R_2	R_3
0	3	0
3	0	2
3	0	2
2	1	1
0	0	2

Máximo

R_1	R_2	R_3
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Necesario

R_1	R_2	R_3
7	1	3
0	2	0
6	0	0
0	1	1
4	3	1

T_0
T_1
T_2
T_3
T_4

Disponible

R_1	R_2	R_3
2	1	0

- Se hace asignación $T_0 = [0, 2, 0]$ desde Disponible
- Se resta a Disponible la asignación $[0, 2, 0] = [2, 1, 0]$
- Se suma a Asignación actual de T_0 lo asignado $[0, 2, 0]$.
- Se resta a Necesario de T_0 lo Actual de T_0 .

- Es este punto vemos que la solicitud $T_0 = [0, 2, 0]$, de ser asignada, lleva al sistema a un estado en el queda sin recursos disponibles.
 - Todos los T_i se quedarán esperando en interbloqueo
- La solicitud $T_0 = [0, 2, 0]$, aunque puede pensarse que es viable ser asignada, llevaría al sistema a un estado de interbloqueo.
- T_0 tendría que esperar a que aparezcan otras solicitudes o devoluciones de recursos que lo satisfagan y que no pongan al sistema en un estado no seguro.

Algoritmo del banquero

- Se requieren varias estructuras de datos para su implementación.
- Sea n el número de hilos/procesos en el sistema (T_1, T_2, \dots, T_n)
- Sea m el número de tipos de recursos (R_1, R_2, \dots, R_m)
- Disponible
 - Vector de longitud m . Indica el número de recursos disponibles de cada tipo.
 - $Disponible[j] = k$, indica que hay k instancias disponibles del recurso R_j .
- Máximo
 - Matriz de $n \times m$ que indica la demanda máxima de recursos por cada hilo/proceso.
 - $Máximo[i][j] = k$, indica que el hilo T_i requiere máximo k instancias del recurso R_j .

Algoritmo del banquero

- Asignación (actual)
 - Matriz de $n \times m$ que indica el número de recursos actualmente asignados a cada T_i .
 - $Asignación[i][j] = k$, indica que T_i tiene asignado k instancias del recurso R_j .
- Necesario
 - Matriz de $n \times m$ que indica el número de cada recurso que le hacen falta a cada hilo/proceso para terminar.
 - $Necesario[i][j] = k$, indica que T_i necesita k instancias adicionales de R_j para terminar.
 - $Necesario[i][j] = Máximo[i][j] - Asignación[i][j]$

Algoritmo del banquero: estado seguro

- Paso 1: Inicializar las siguientes estructuras
 - Sea *Trabajo*[] un vector de tamaño *m*
 - Sea *Finaliza*[] un vector de tamaño *n*
 - Inicializar *Trabajo* = *Disponible*
 - Inicializar *Finaliza*[*i*] = *Falso* para $i = 0, 1, 2, \dots, n - 1$
- Paso 2: Encontrar un índice *i* tal que, se cumplan la siguiente condición
 - *Finaliza*[*i*] = *Falso* y $Necesario_i \leq Trabajo$
 - Si *i* no existe, ir al paso 4.
 - $Necesario_i$ es el vector fila de la matriz *Necesario*

Algoritmo del banquero: estado seguro

- Paso 3

- $Trabajo = Trabajo + Asignación_i$
- $Finaliza[i] = Verdadero$
- Regresa al paso 2

- Paso 4

- Si $Finaliza[i] == Verdadero$ para todo i , entonces el sistema está en estado seguro.

Algoritmo del banquero: solicitud de recurso

- Sea $Solicitud_i$ un vector que indica la solicitud de T_i .
- Si $Solicitud_i[j] == k$, entonces T_i solicita k instancias del recurso R_j .
- Cuando llega una nueva solicitud al sistema de T_i , se ejecutan las siguientes acciones.
 - Paso 1: Si $Solicitud_i \leq Necesita_i$, ir al paso 2. Sino, error ya que T_i excede su demanda máxima de recursos
 - Paso 2: $Solicitud_i \leq Disponible$, ir al paso 2. Sino, T_i debe esperar por recursos disponibles.
 - Paso 3: Se modifican los estados de la siguiente manera (siguiente *slide*)

Algoritmo del banquero: solicitud de recurso

- Paso 3: Se modifican los estados de la siguiente manera
 - $Disponible = Disponible - Solicitud_i$
 - $Asignación_i = Asignación_i + Solicitud_i$
 - $Necesita_i = Necesita_i - Solicitud_i$
- Si la asignación lleva a un estado seguro
 - Se asignan los recursos a T_i
- Si la asignación lleva a un estado inseguro
 - T_i debe esperar por $Solicitud_i$ y se restaura el estado anterior del sistema (antes de la asignación de recursos).

Referencias

- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). Deadlock avoidance. In *Operating Systems Concepts* (10th ed., pp. 330–337). John Wiley & Sons, Inc.