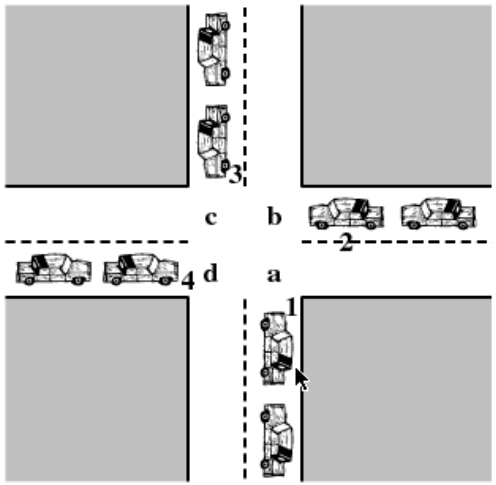
Concurrencia: Deadlock e Inhanición

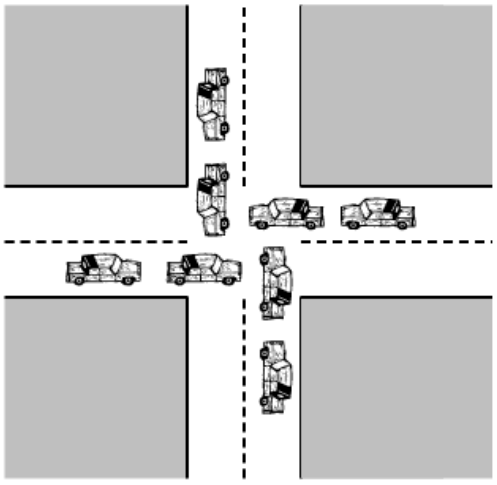
Capítulo 6

Deadlock

- Bloqueo permanente de un conjunto de procesos que compiten por un recurso del sistema o cooperan (y se comunican) mútuamente
- En un conjunto de procesos en deadlock, cada proceso está esperando que suceda un evento, el cual sólo puede ser generado por algún otro proceso en deadlock.
- Involucra necesidades conflicitvas de recursos por dos o más procesos
- No existe una solución eficiente



(a) Deadlock possible


(b) Deadlock

Ejemplo de deadlock

Proceso P
Get A
…
Get B
…
Release A
….
Release B

Proceso Q
Get B
…
Get A
…
Release B
….
Release A

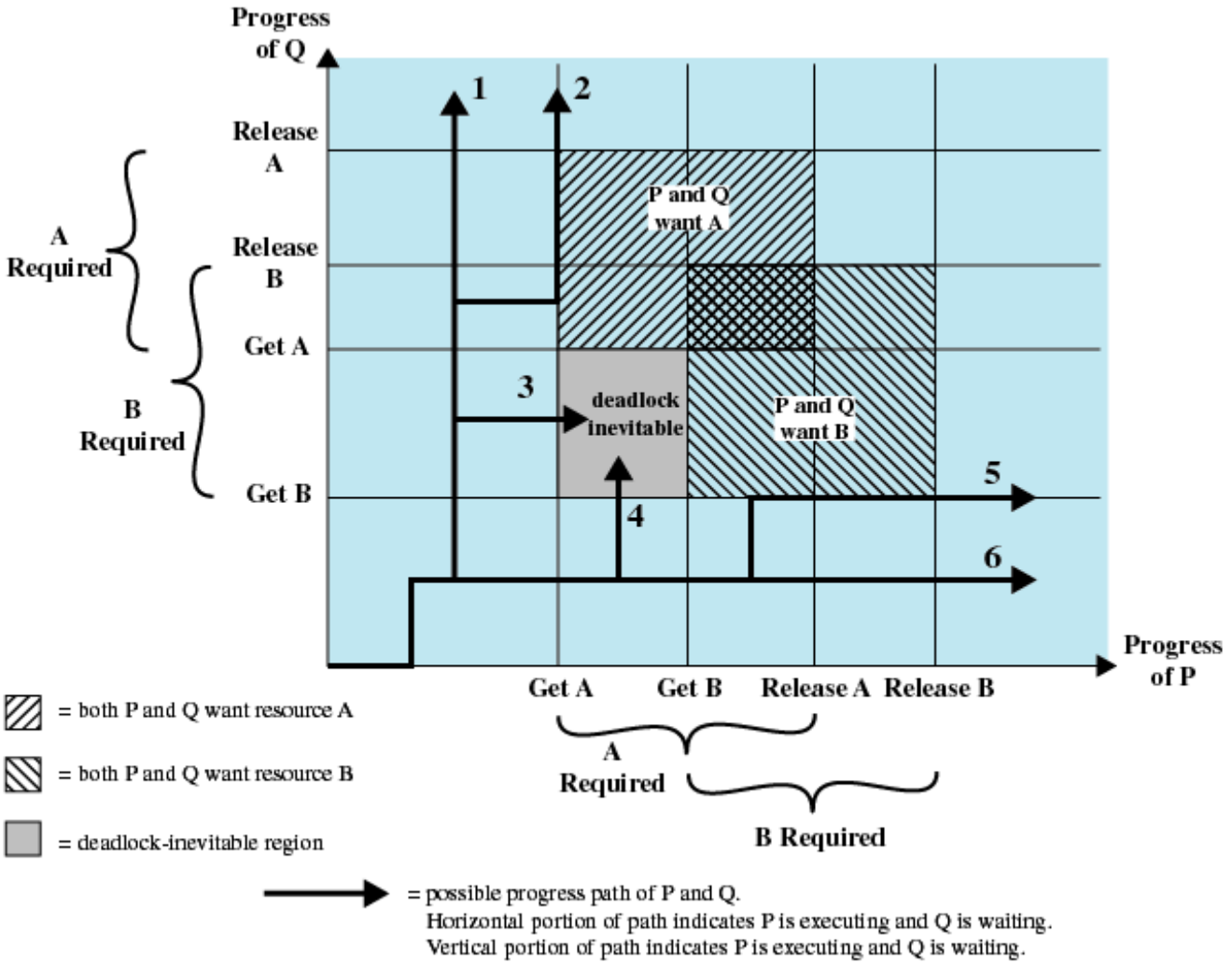


Figure 6.2 Example of Deadlock

Ejemplo de no deadlock

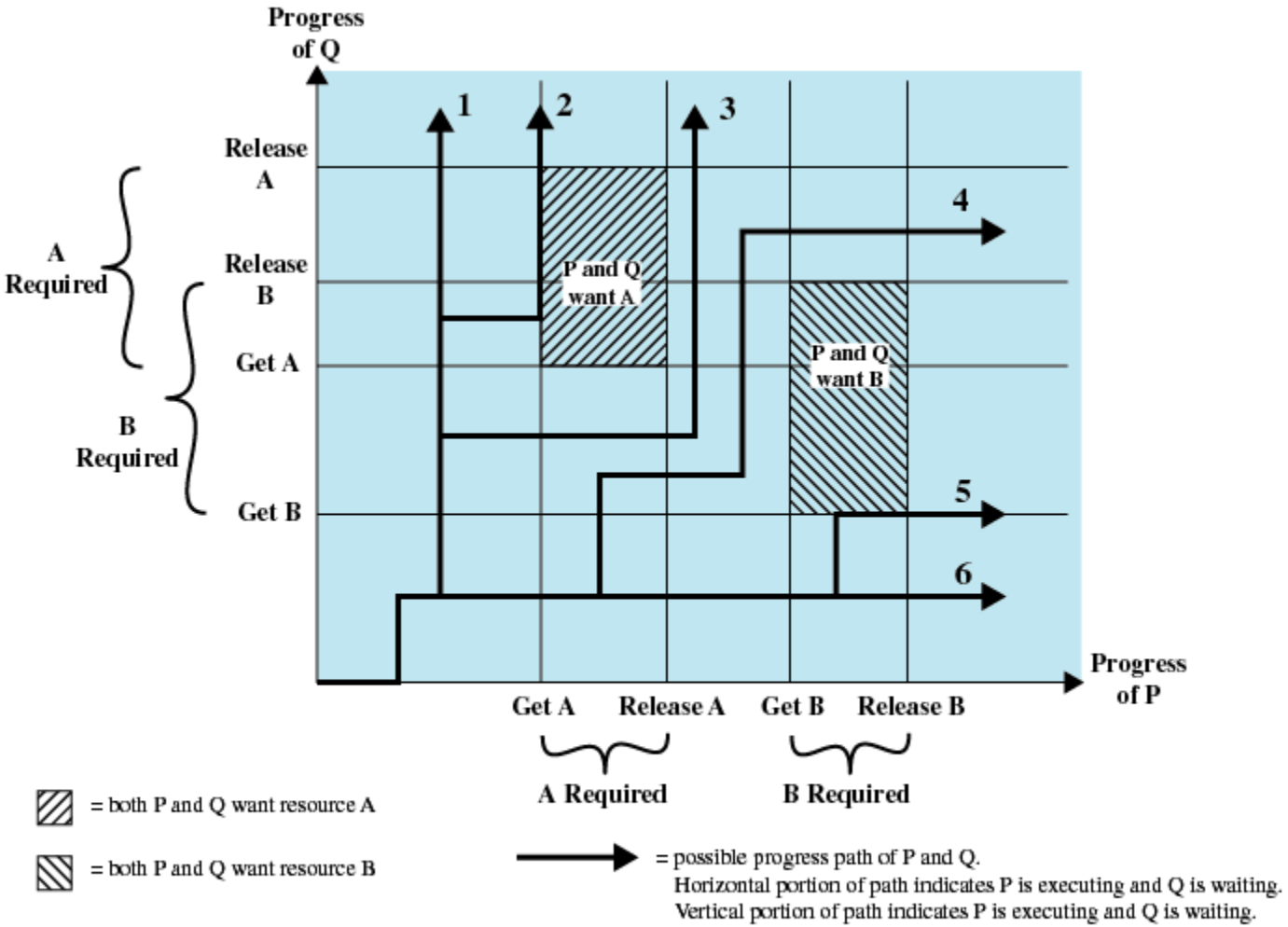


Figure 6.3 Example of No Deadlock [BACO03]

Proceso P
Get A
…
Release A

Get B
….
Release B

Proceso Q
Get B
…
Get A
…
Release B
….
Release A

Recursos resusables

- Usados por sólo un proceso a la vez y no consumido (acabado) por el uso
- Los procesos obtienen dichos recursos y los liberan para que otros procesos los usen
- Ejemplos de tales recursos son: canales de I/O, memoria primaria y secundaria, dispositivos, archivos.
- Deadlock ocurre cuando cada proceso retiene un recurso y solicita otro
- Ejemplo de deadlock, solicitud de disco y cinta

D: disco
T: cinta

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Otro ejemplo

- Dos procesos solicitan memoria
- Existe 200K en total y la secuencia de requerimientos es la siguiente
- Deadlock ocurrirá si ambos procesos continuan su ejecución hasta alcanzar el segundo requerimiento

P1

. . .

Request 80 Kbytes;

. . .

Request 60 Kbytes;

P2

. . .

Request 70 Kbytes;

. . .

Request 80 Kbytes;

Recursos consumibles

- Creados (producidos) y destruidos (consumidos) dinámicamente por procesos
- Ejemplos de tales recursos son interrupciones, señales, información en buffers de I/O
- Ejemplo de deadlock: requeriemiento de memoria. Si la función Receive() es blocking, deadlock podria ocurrir

P1

```
. . .
Receive(P2);
. . .
Send(P2, M1);
```

P2

```
. . .
Receive(P1);
. . .
Send(P1, M2);
```

Grafos de procuramiento de recursos

- Grafo dirigido que representa el estado actual de un sistema de recursos y procesos
- Procesos son representados por círculos y recursos por rectángulos
- Un arco desde un proceso a un recurso indican que el proceso ha solicitado el recurso, pero éste aún no ha sido asignado
- Un arco desde un recurso a un procesos indica que el recurso ha sido asignado al proceso

(a) Resouce is requested (b) Resource is held

Grafos de procuramiento de recursos

- Los círculos en un recurso indican el número de instancias de dicho recurso

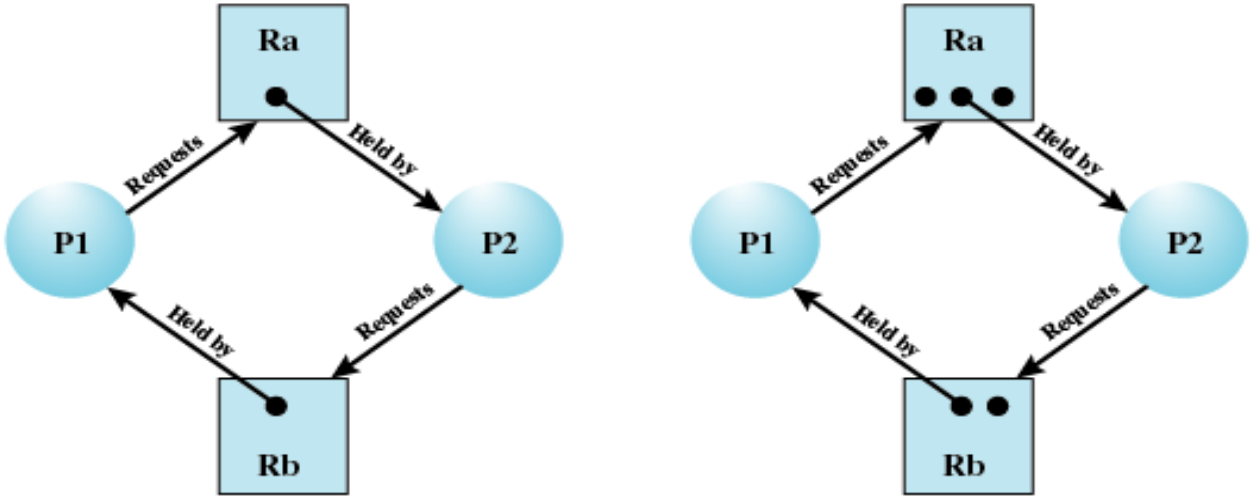
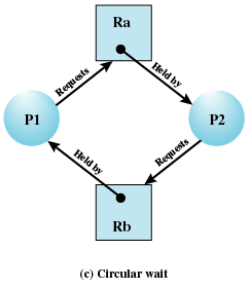


Figure 6.5 Examples of Resource Allocation Graphs

Condiciones necesarias y suficientes para que ocurra Deadlock

- Existen tres condiciones que deben cumplirse en un sistema para que exista la posibilidad que el sistema entre en un deadlock
  - **Exclusión mútua**
    - Sólo un proceso puede usar un recurso a la vez
  - **Hold-and-wait**
    - Se permite que un proceso pueda mantener un recurso mientras espera por otro
  - **No desapropiación**
    - Los recursos no pueden ser desapropiados de los procesos que los están usando
- Estas condiciones pueden llevar a un deadlock, pero no son suficientes. Una cuarta condición suficiente es
- **Espera circular**
  - Existe una cadena cerrada de procesos, tal que cada proceso mantiene al menos un recurso que es solicitado por otro proceso en la cadena



(c) Circular wait

Prevención de Deadlock

- La idea es encontrar métodos que garanticen que el sistema no entrará en deadlock.
- Un **método indirecto** es prevenir o evitar que algunas de las tres condiciones necesarias para deadlock ocurran.
  - Exclusión mútua:
    - ¿Sería razonable excluir la condición de exclusión mútua en sistemas concurrentes? NO. Por lo tanto exclusión mútua debe permitirse
  - Hold and wait:
    - Esta característica puede evitarse. Por ejemplo, se puede requerir que los procesos soliciten todos los recursos de una vez y si ellos no están dispoibles, el proceso se bloquea hasta que todos estén disponibles
    - Ineficiente
  - No desapropiación:
    - Por ejemplo, si un proceso tiene varios recursos asignados y se le niega la petición de uno nuevo, entonces debería liberar los que ya tiene asignados

Prevención de Deadlock

- Un **método directo** sería garantizar que nunca se satisfaga la condición suficiente, es decir que se forme una cadena circular.
- Por ejemplo, se puede evitar la formación de un loop requiriendo que todos los recursos sean ordenados en una lista lineal y exigiendo que los procesos soliciten recursos de acuerdo al orden.
- Luego R_i precede a R_j si $i < j$
- Sea R_i y R_j dos recursos y A y B dos procesos
  - Suponga que A y B están en deadlock, tal que A tiene al recurso R_i y solicita a R_j, y B tiene a R_j y solicita a Ri
  - Esto es imposible pues implicaría que $i < j$ y $j < i$

“Evitar” Deadlock (avoidance)

- Se determina dinámicamente si la asignación de un recurso produciría o no una posibilidad de deadlock
- Más eficiente, pero requiere conocer la secuencia total de requerimientos de todos los procesos
- Deadlock avoidance permite las tres condiciones necesarias vistas anteriormente, pero niega la asignación de un recurso si dicha asignación podría llevar a producir un deadlock
- Dos técnicas:
  - Denegación de inicio de un proceso
  - Denegación de requerimiento de un recurso

Denegación inicio de un proceso

- Considere un sistema con *n* procesos y *m* recursos distintos
- Definamos

$$R = (R_1, R_2, \ldots, R_m)$$

$$V = (V_1, V_2, \cdots, V_m)$$

$$C = \begin{vmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{vmatrix}$$

$$A = \begin{vmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{vmatrix}$$

R_i: cantidad de recurso del tipo i

V_i: cantidad del recurso i sin asignar

C_{ij} : solicitud de recurso j por proceso i

A_{ij} : asignación actual de recurso j a proceso i

$$R_j = V_j + \sum_{i=0}^{n} A_{ij}, \text{para todo j}$$

$$C_{ij} \leq R_j, \text{para todo i, j}$$

$$A_{ij} \leq C_{ij}, \text{para todo i, j}$$

<u>Política:</u> iniciar un nuevo proceso (n+1) ssi $R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}, \text{para todo j}$

Es decir, ssi el requerimiento máximo de todos los procesos existentes más aquellos del nuevo proceso, pueden ser satisfechos

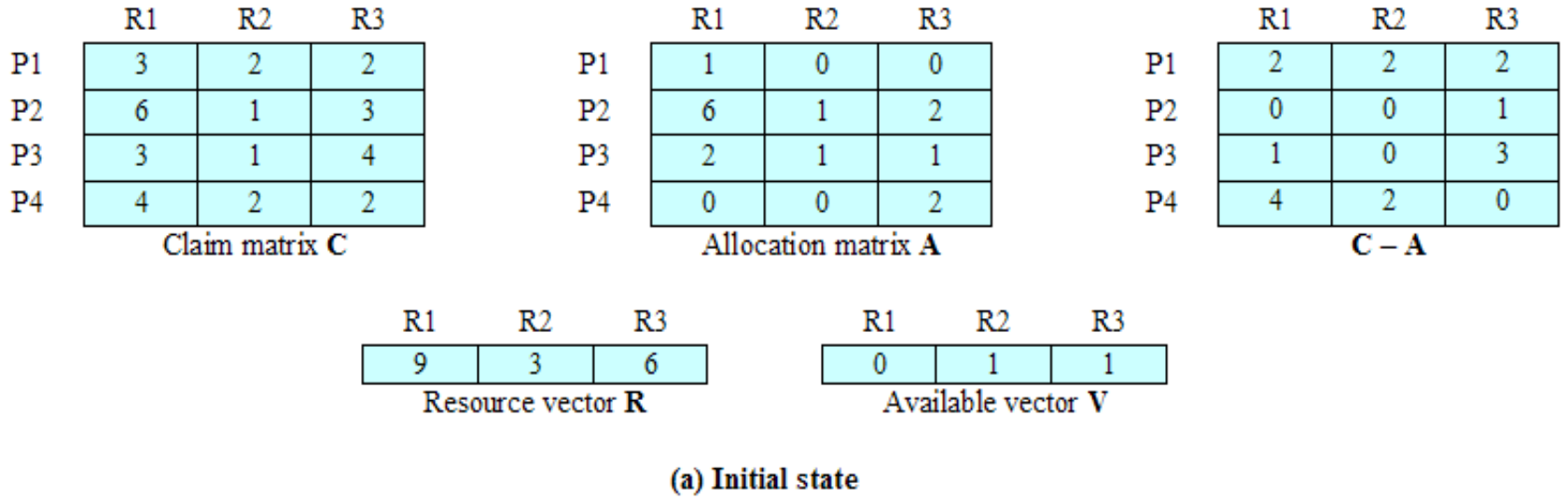
$$R_j \geq C_{(n-1)j} + \sum_{i=1}^{n} C_{ij}$$

Denegación asignación de recurso

- Conocido como el *algoritmo del banquero*
- Se define el estado del sistema como la asignación actual de recursos a procesos, es decir por los vectores *R*, *V*, y las matrices *C* y *A*
- El sistema está en estado seguro si existe al menos una secuencia de asignación que no resulta en deadlock
- Un estado inseguro es cuando el sistema no está en estado seguro

Determinación de estado seguro
Estado inicial

- Suponga que el estado actual es el siguiente



	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

- Claramente P1 no podría correr hasta su término por que sólo tiene 1 unidad de R1 y todavia necesita 2 más.

Determinación de estado seguro
P2 corre hasta terminar

- Sin embargo, si asignaramos una unidad más de R3 a P2, P2 si podria continuar su ejecución hasta el fin y así liberar todos sus recursos asignados

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Determinacion de estado seguro
P1 y P3 corren hasta el fin

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Determinación de estado inseguro

- Suponga que a partir del estado actual, P1 realiza una petición de un recurso de R1 y un recurso de R3

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

Ningún proceso podría continuar, pues todos requieren al menos una instancia de R1 y actualmente hay 0

➔ Estado inseguro

Determinación de estado seguro

- En general, un sistema está en estado seguro si al realizar la asignación solicitada se cumple que

$$C_{ij} \; ¡ \; A_{ij} \cdot V_j \; ; 8j$$

```
struct state
{
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else                                                /* simulate alloc */
{
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else
{
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Determinación estado seguro

- La determinación si la asignación conduce o no a un estado inseguro es el algoritmo del banquero

```
boolean safe (state S)
{
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible)
    {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found)                          /* simulate execution of Pk */
        {
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else
            possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Algunas características de deadlock avoidance

- Todos los procesos deben comunicar con anticipación el número máximo de recursos de cada tipo que solicitará
- Los procesos son independientes, es decir no debe existir ningún tipo de sincronización entre ellos
- El número de recursos debe ser fijo
- Ningún proceso puede terminar o salir mientras posee recursos asignados

Deadlock Detection

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix **Q**

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix **A**

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Figure 6.10 Example for Deadlock Detection

Strategies once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Selection Criteria Deadlocked Processes

- Least amount of processor time consumed so far
- Least number of lines of output produced so far
- Most estimated time remaining
- Least total resources allocated so far
- Lowest priority

Strengths and Weaknesses of the Strategies

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates on-line handling	•Inherent preemption losses

Dining Philosophers Problem

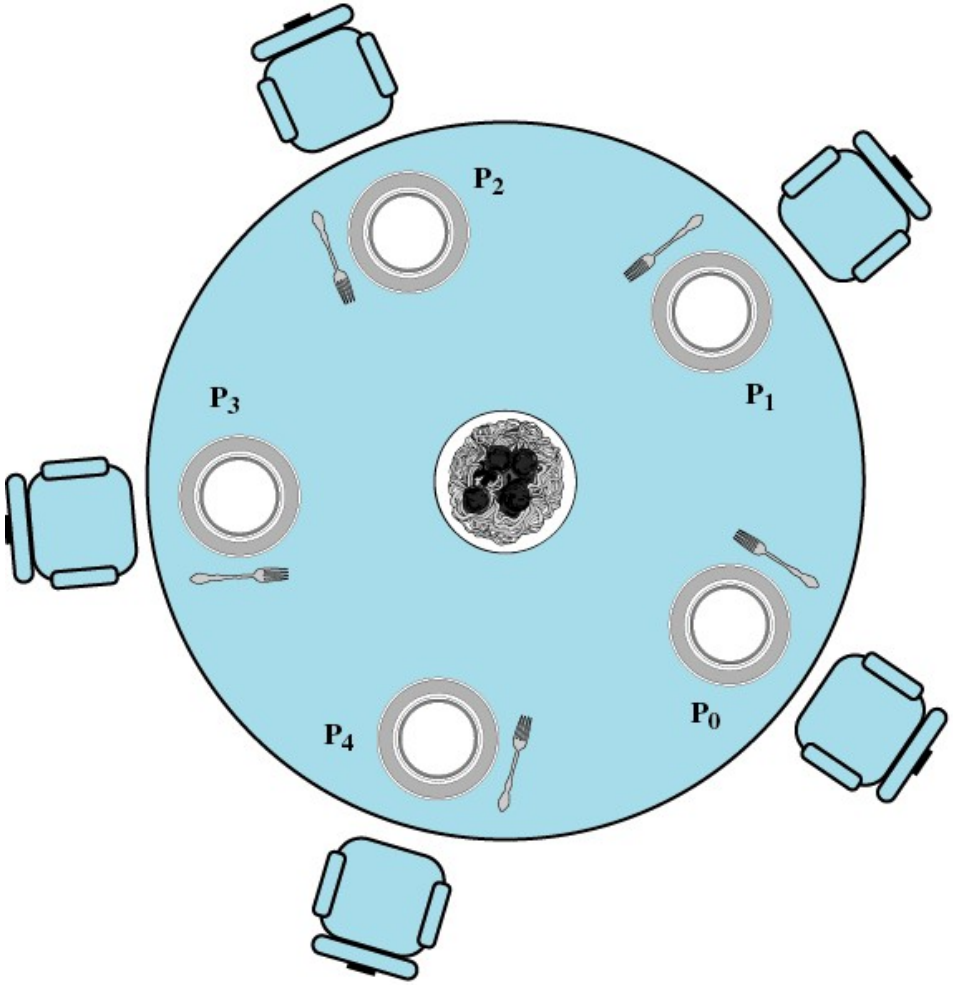


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true)
     {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Dining Philosophers Problem

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int I)
{
    while (true)
    {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

Dining Philosophers Problem

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)          /* pid is the philosopher id number */
{
   int left = pid;
   int right = (pid++) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);         /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (pid++) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                       /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true)
   {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

Dining Philosophers Problem

```
monitor dining_controller;
enum states (thinking, hungry, eating} state[5];
cond needFork[5]                                /* condition variable */

void get_forks(int pid)          /* pid is the philosopher id number */
{
   state[pid] = hungry;                    /* announce that I'm hungry */
   if (state[(pid+1) % 5] == eating
   || (state[(pid-1) % 5] == eating
   cwait(needFork[pid]);         /* wait if either neighbor is eating */
   state[pid] = eating;     /* proceed if neither neighbor is eating */
}

void release_forks(int pid)
{
   state[pid] = thinking;
   /* give right (higher) neighbor a chance to eat */
   if (state[(pid+1) % 5] == hungry)
   || (state[(pid+2) % 5]) != eating)
   csignal(needFork[pid+1]);
   /* give left (lower) neighbor a chance to eat */
   else if (state[(pid-1) % 5] == hungry)
   || (state[(pid-2) % 5]) != eating)
   csignal(needFork[pid-1]);
}
```

```
void philosopher[k=0 to 4]           /* the five philosopher clients */
{
   while (true)
   {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```


Figure 6.17 Another Solution to the Dining Philosophers Problem Using a Monitor

UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

Table 6.2 UNIX Signals

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Linux Kernel Concurrency Mechanisms

- Includes all the mechanisms found in UNIX
- Atomic operations execute without interruption and without interference

Linux Atomic Operations

Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic__set(atomic_t *v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i, atomic_t *v)`	Subtract i from v
`void atomic__inc(atomic_t *v)`	Add 1 to v
`void atomic__dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise

Linux Atomic Operations

Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr

Linux Kernel Concurrency Mechanisms

- Spinlocks
  - Used for protecting a critical section

Table 6.4 Linux Spinlocks

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise

Table 6.5 Linux Semaphores

Traditional Semaphores	
`void sema_init(struct semaphore *sem, int count)`	Initializes the dynamically created semaphore to the given count
`void init_MUTEX(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
`void init_MUTEX_LOCKED(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 0 (initially locked)
`void down(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
`int down_interruptible(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received.
`int down_trylock(struct semaphore *sem)`	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
`void up(struct semaphore *sem)`	Releases the given semaphore
Reader-Writer Semaphores	
`void init_rwsem(struct rw_semaphore, *rwsem)`	Initalizes the dynamically created semaphore with a count of 1
`void down_read(struct rw_semaphore, *rwsem)`	Down operation for readers
`void up_read(struct rw_semaphore, *rwsem)`	Up operation for readers
`void down_write(struct rw_semaphore, *rwsem)`	Down operation for writers
`void up_write(struct rw_semaphore, *rwsem)`	Up operation for writers

Linux Kernel Concurrency Mechanisms

Table 6.6 Linux Memory Barrier Operations

`rmb()`	Prevents loads from being reordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents loads and stores from being reordered across the barrier
`barrier()`	Prevents the compiler from reordering loads or stores across the barrier
`smp_rmb()`	On SMP, provides a `rmb()` and on UP provides a `barrier()`
`smp_wmb()`	On SMP, provides a `wmb()` and on UP provides a `barrier()`
`smp_mb()`	On SMP, provides a `mb()` and on UP provides a `barrier()`

SMP = symmetric multiprocessor
UP = uniprocessor

Solaris Thread Synchronization Primitives

- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables

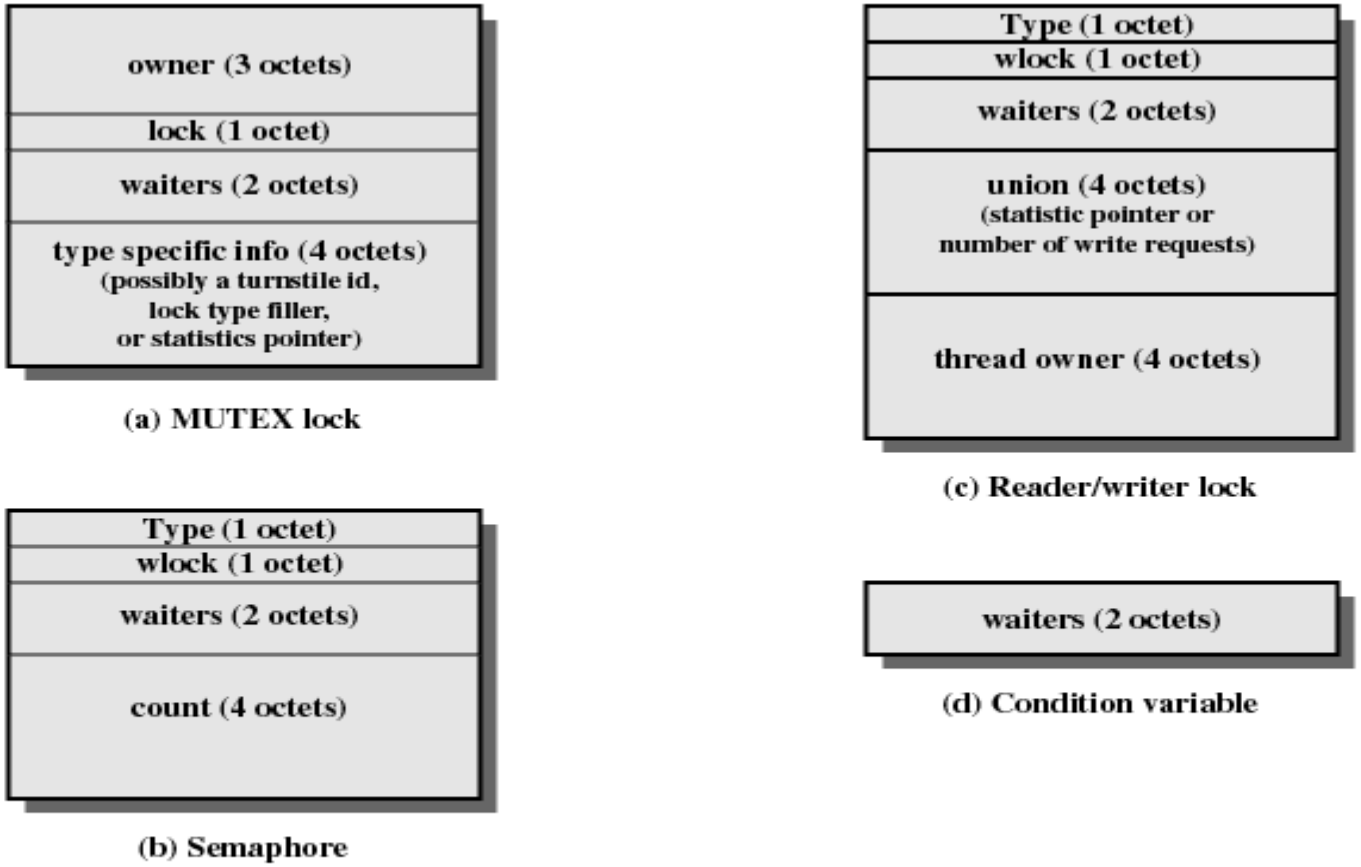


Figure 6.15 Solaris Synchronization Data Structures

Table 6.7 Windows Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Event	An announcement that a system event has occurred	Thread sets the event	All released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File change notification	A notification of any file system changes.	Change occurs in file system that matches filter criteria of this object	One thread released
Console input	A text window screen buffer (e.g., used to handle screen I/O for an MS-DOS application)	Input is available for processing	One thread released
Job	An instance of an opened file or I/O device	I/O operation completes	All released
Memory resource notification	A notification of change to a memory resource	Specified type of change occurs within physical memory	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Note: Colored rows correspond to objects that exist for the sole purpose of synchronization.