Chapter 6 lecture

Stalling 9ed

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

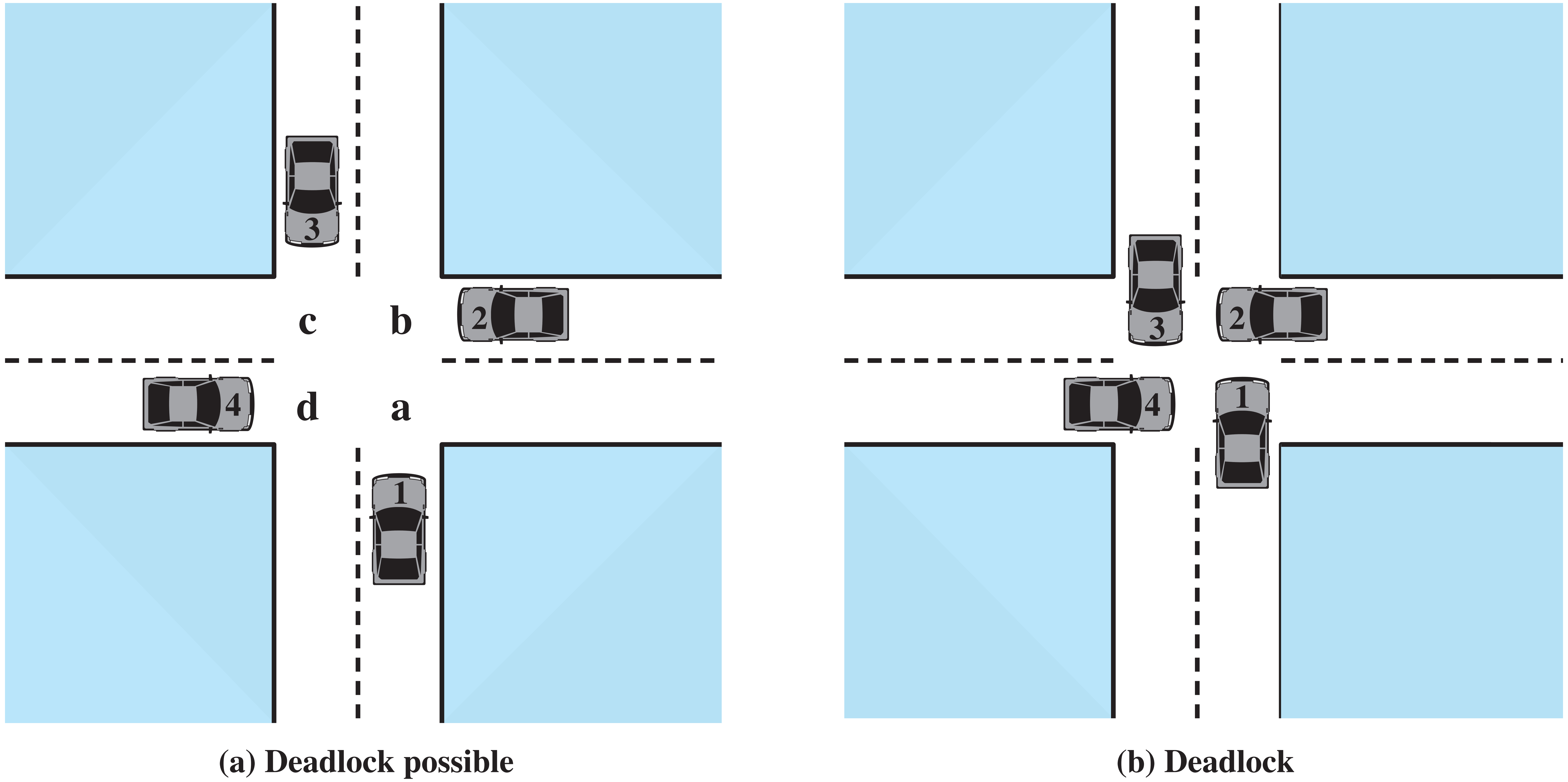
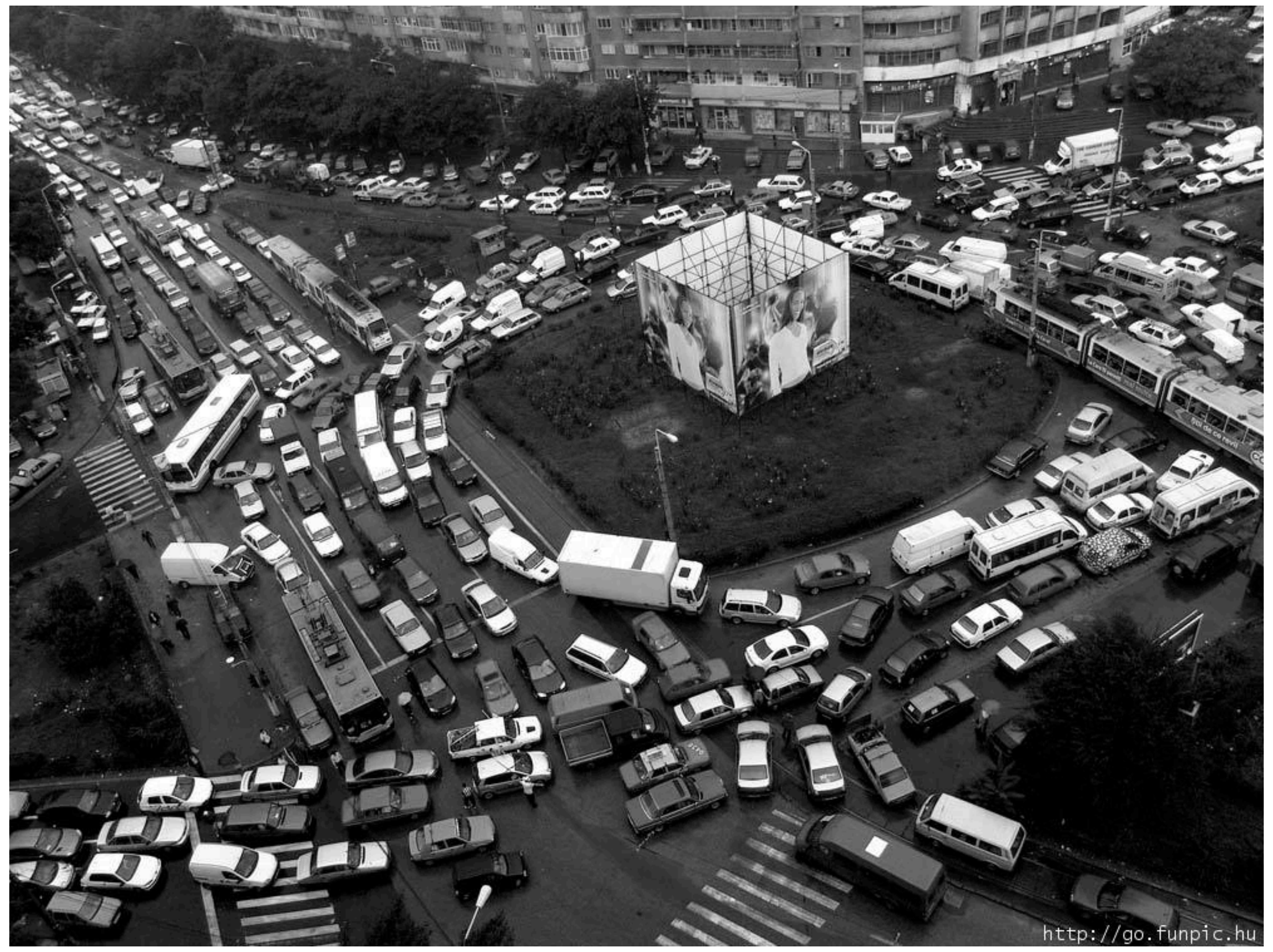


Figure 6.1 Illustration of Deadlock

Better illustrations ☺



Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
  - E.g. Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

Example of Deadlock

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

Now consider the following sequence:

$p_0\ p_1\ q_0\ q_1\ p_2\ q_2$

Another Example of Deadlock

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1	P2
. . .	. . .
Request 80 Kbytes;	Request 70 Kbytes;
. . .	. . .
Request 60 Kbytes;	Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Example of Deadlock

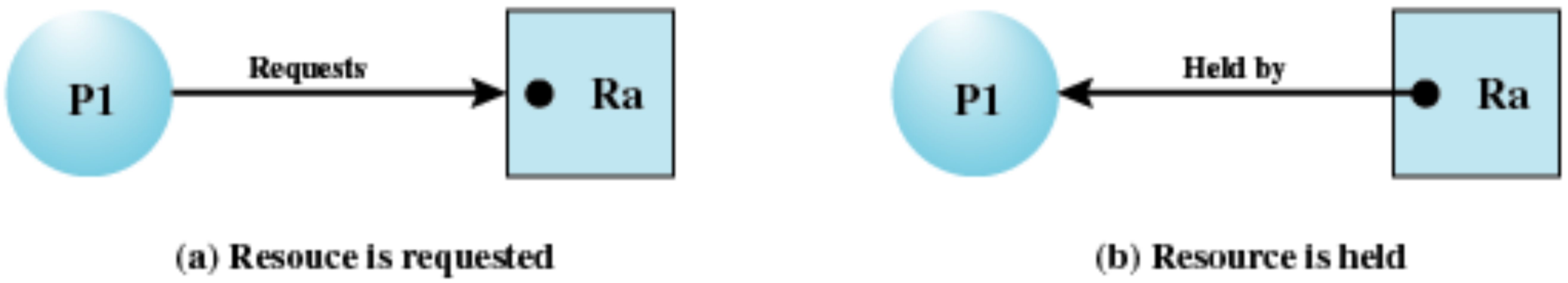
- Deadlock occurs if receive is blocking

```
P1
. . .
Receive(P2);
. . .
Send(P2, M1);
```

```
P2
. . .
Receive(P1);
. . .
Send(P1, M2);
```

Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes



Resource Allocation Graphs

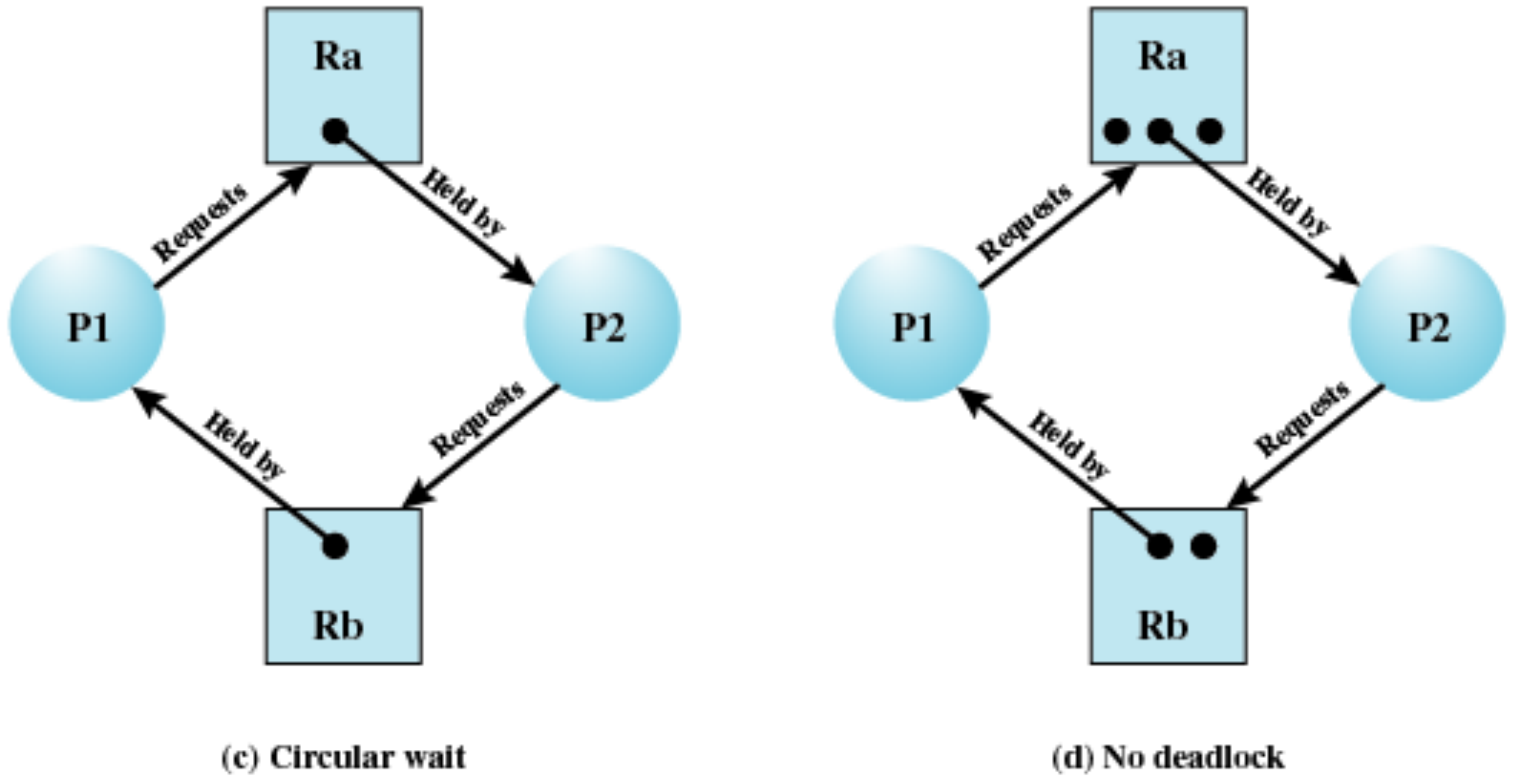


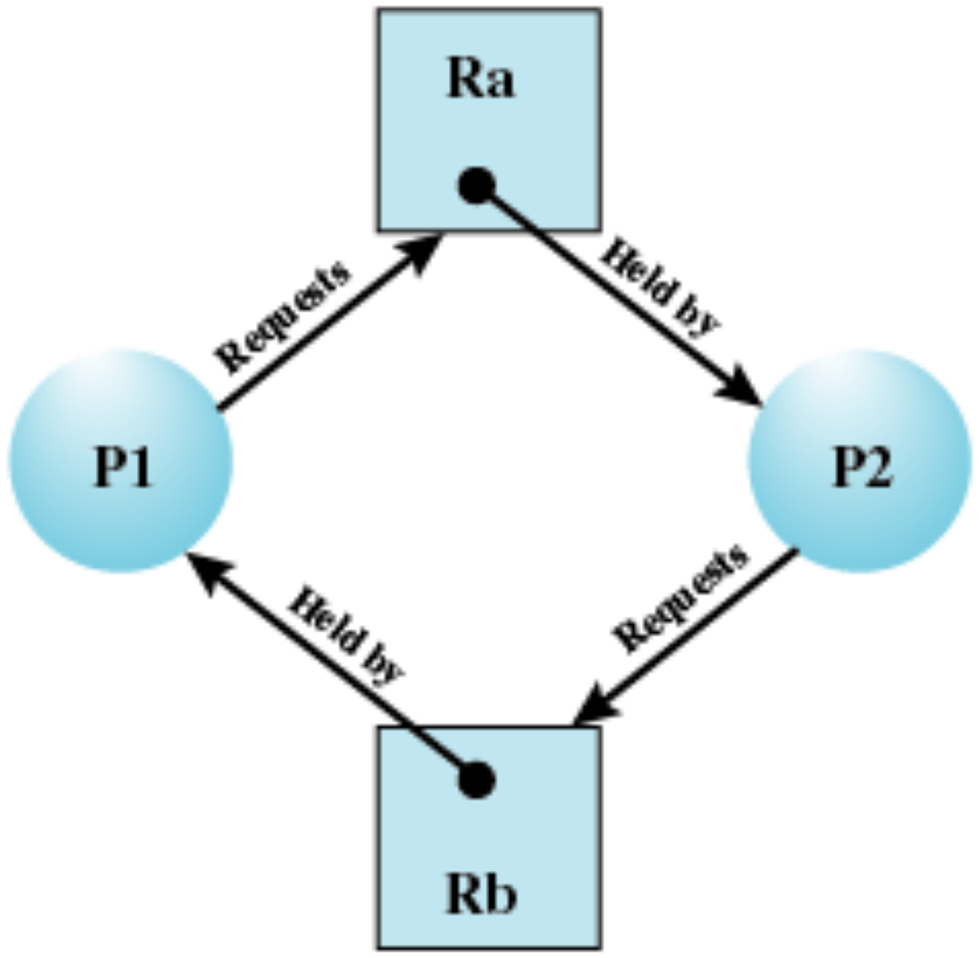
Figure 6.5 Examples of Resource Allocation Graphs

Conditions for Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others
- No preemption
  - No resource can be forcibly removed from a process holding it

Conditions for Deadlock

- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



(c) Circular wait

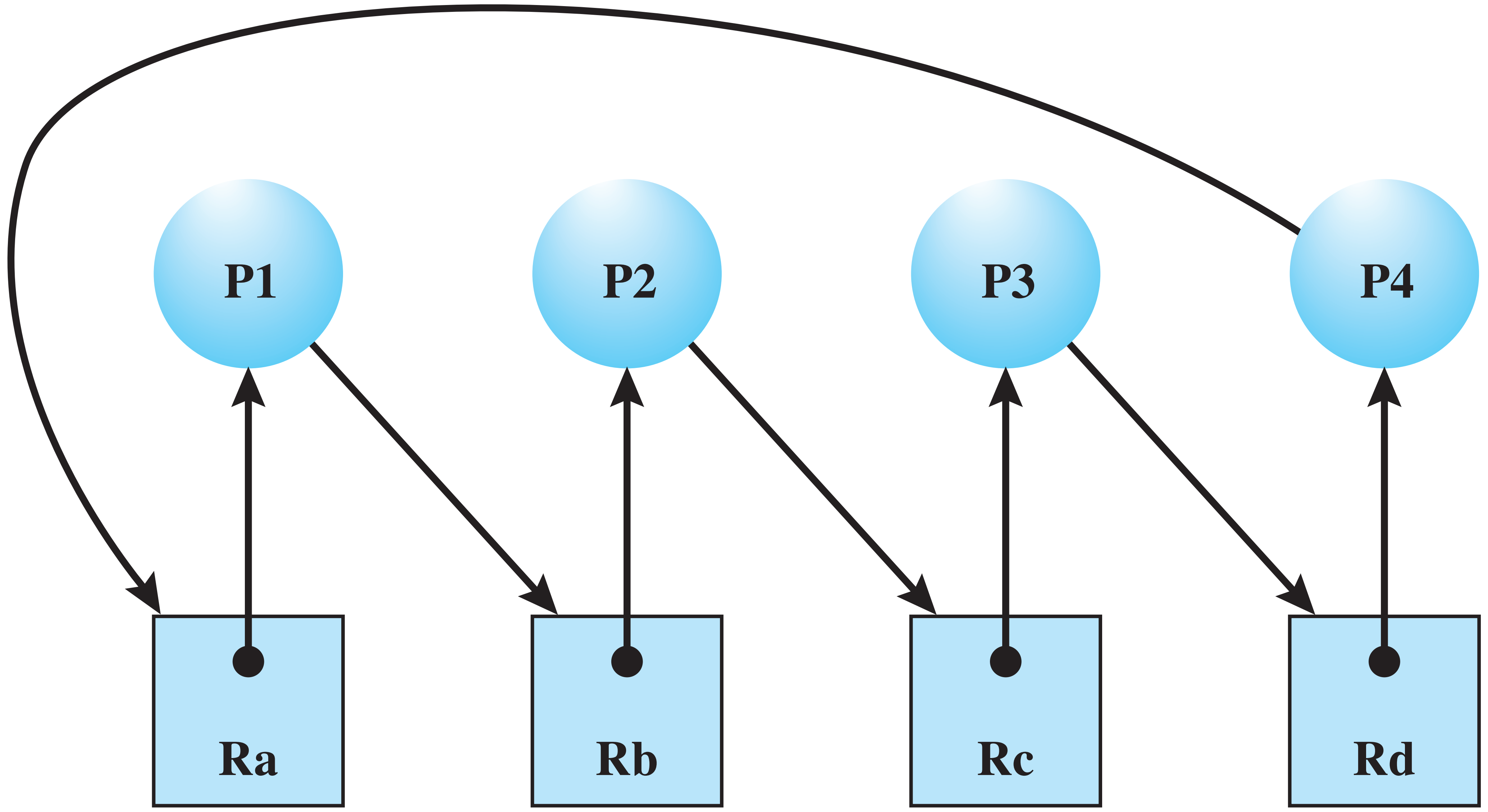


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Deadlock Approaches

- Prevent - eliminate conditions that cause deadlock
- Avoidance - don’t allow deadlock to occur
- Allow - then suffer consequences!

Deadlock Prevention

- Mutual Exclusion
  - Must be supported by the operating system
- Hold and Wait
  - Require a process request all of its required resources at one time

Deadlock Prevention

- No Preemption
  - Process must release resource and request again
  - Operating system may preempt a process to require it releases its resources
- Circular Wait
  - Define a linear ordering of resource types

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process request

Two Approaches to Deadlock Avoidance

- Do not start a process if its demands might lead to deadlock
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Resource Allocation Denial

- **Banker’s algorithm**
- **State of the system:** the current allocation of resources to processes
- **Safe state:** there is at least one sequence that does not result in deadlock
- **Unsafe state:** a state that is not safe

Determination of a Safe State
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	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

Determination of a Safe State P2 Runs to Completion

	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

Determination of a Safe State
P1 Runs to Completion

	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

Determination of a Safe State
P3 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

Determination of an Unsafe 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	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

Determination of an Unsafe 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

```
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 allocation algorithm

```
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)

Figure 6.9 Deadlock Avoidance Logic

Deadlock Avoidance

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Deadlock Detection

- Two phase process
  - deadlock detection
    - figure out that deadlock occurred
  - deadlock resolution
    - do something to resolve it

Deadlock Detection Algorithm

Use Allocation and Available matrices from safety algorithm, Create Request matrix Q

1. Mark each process that has a row in the Allocation matrix of all zeros. A process that has no allocated resources cannot participate in a deadlock.
2. Initialize a temporary vector **W** to equal the Available vector.
3. Find an index *i* such that process *i* is currently unmarked and the *i*th row of **Q** is less than or equal to **W**. That is, $Q_{ik} \leq W_k$, for $1 \leq k \leq m$. If no such row is found, terminate the algorithm.
4. If such a row is found, mark process *i* and add the corresponding row of the allocation matrix to **W**. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

	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

Allocation vector

Figure 6.10 Example for Deadlock Detection

1. Mark P4, because P4 has no allocated resources.
2. Set $\mathbf{W} = (0\ 0\ 0\ 0\ 1)$.
3. The request of process P3 is less than or equal to **W**, so mark P3 and set

$$\mathbf{W} = \mathbf{W} + (0\ 0\ 0\ 1\ 0) = (0\ 0\ 0\ 1\ 1).$$

4. No other unmarked process has a row in **Q** that is less than or equal to **W**. Therefore, terminate the algorithm.

The algorithm concludes with P1 and P2 unmarked, indicating these processes are deadlocked.

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 reoccur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Selection Criteria Deadlocked Processes

- Many criteria to select from, e.g.
  - 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

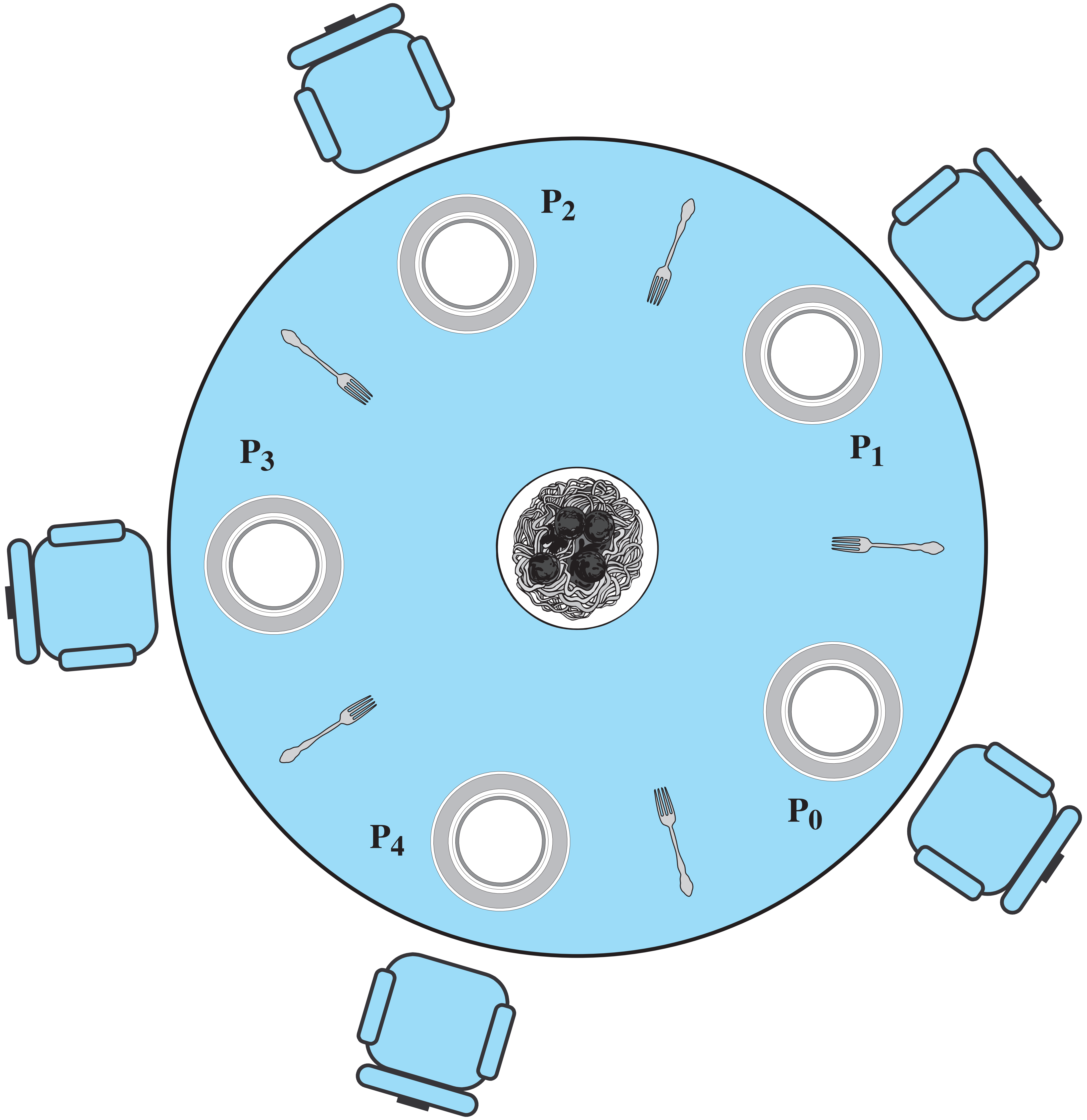


Figure 6.11 Dining Arrangement for Philosophers

```
/* 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

```
/* 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

```
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

UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

UNIX Pipes

- used to carry data from one process to another
- one process writes into the pipe
- the other reads from the other end
- essentially FIFO

UNIX Pipes

- Examples
  - ls | pr | lpr
    - pipe ls into the standard input of pr
    - pr pipes its standard output to lpr
    - pr in this case is called a *filter*
  - ls > filea
  - pr < filea > fileb
    - read input from filea and output to fileb

Signals

- Signals are a facility for handling exceptional conditions similar to software interrupts
- Generated by keyboard interrupt, error in a process, asynchronous events
  - timer
  - job control
- Kill command can generate almost any signal

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

Memory Barrier

- A class of instructions
- Enforces that CPU executes memory operations in order

- Why would one need to enforce in-order execution?

Memory Barrier Operations

- Consider the following 2 processes

  Proc #1:

  loop: load the value of location y,

  if it is 0 goto loop

  print the value in location x

  Proc #2:

  store the value 55 into location x

  store the value 1 into location y

- What is the output?

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

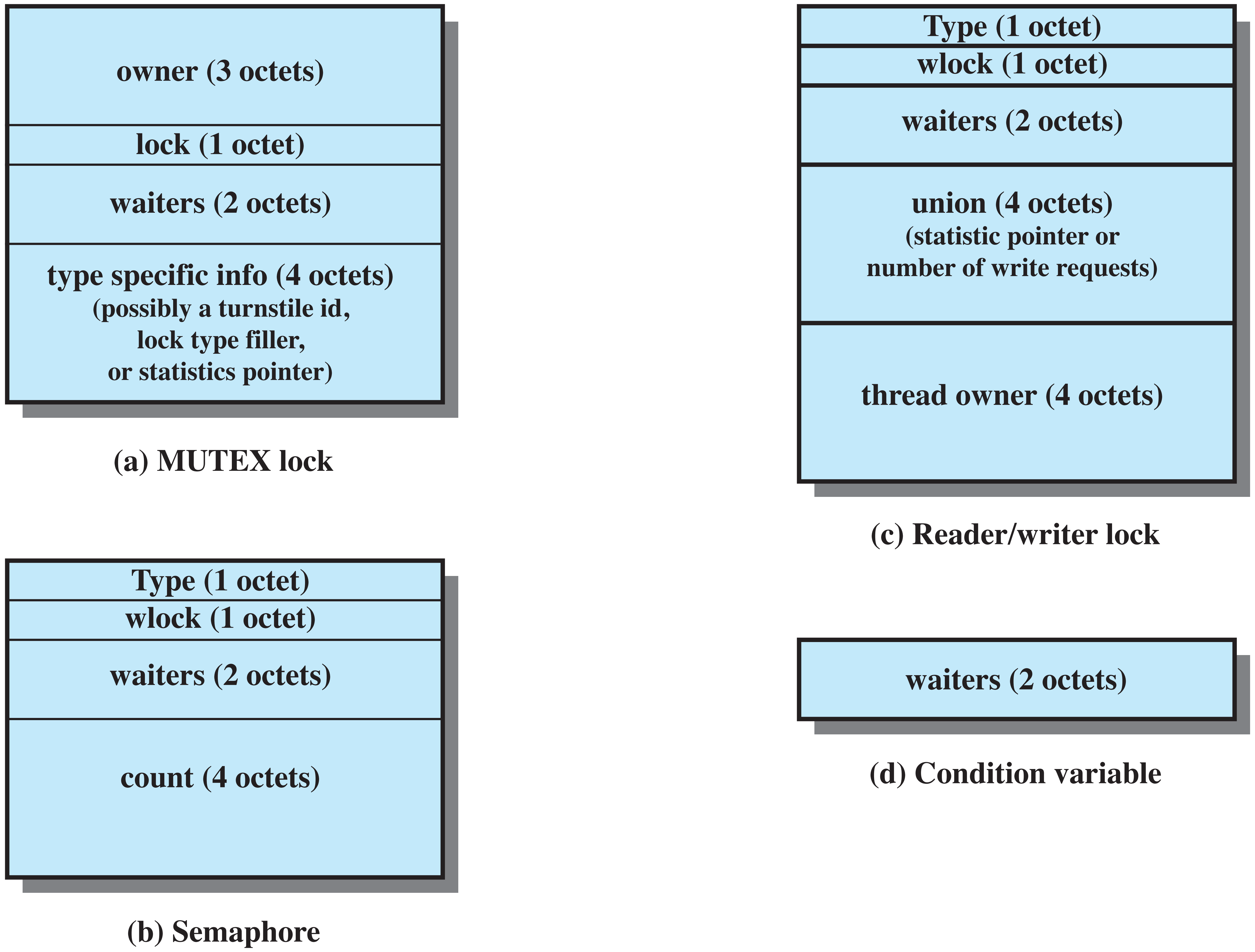


Figure 6.15 Solaris Synchronization Data Structures

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.