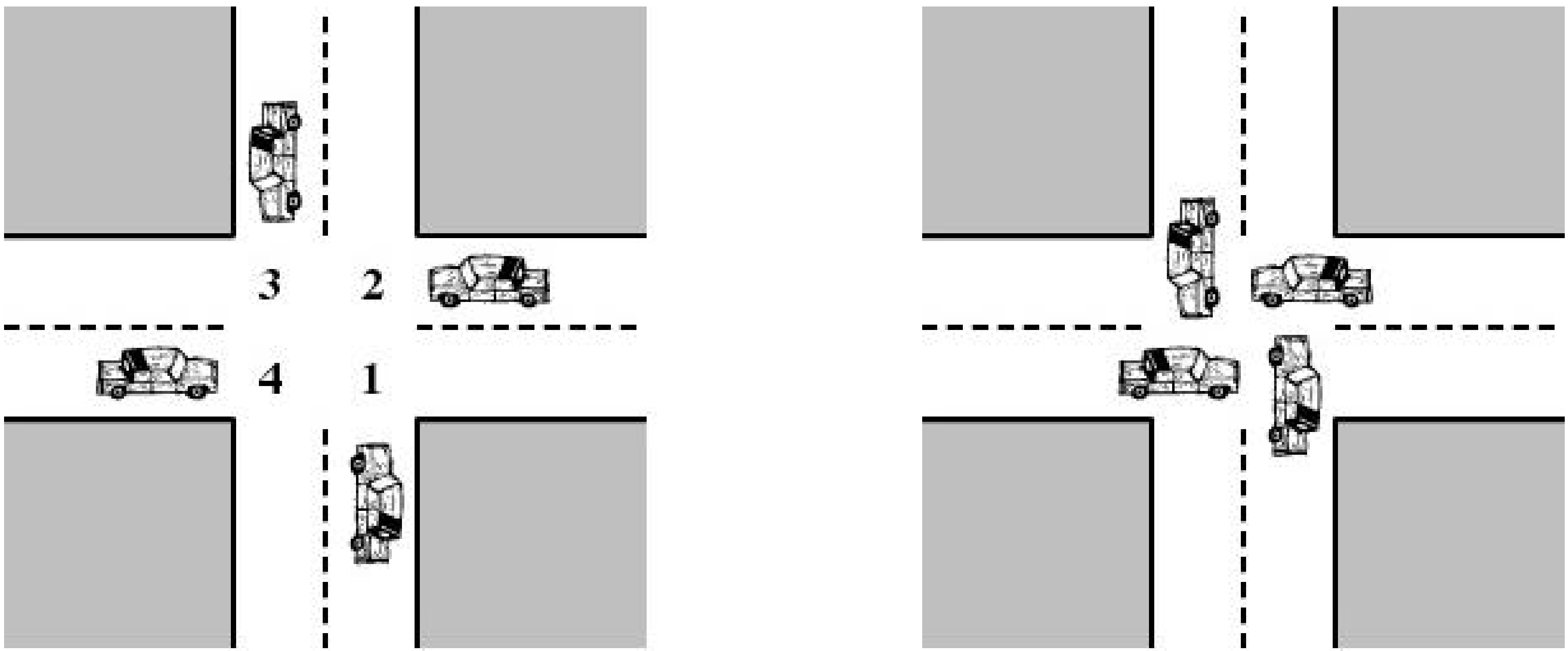
Concurrency: Deadlock and Starvation

Chapter 6

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- Involve conflicting needs for resources by two or more processes



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

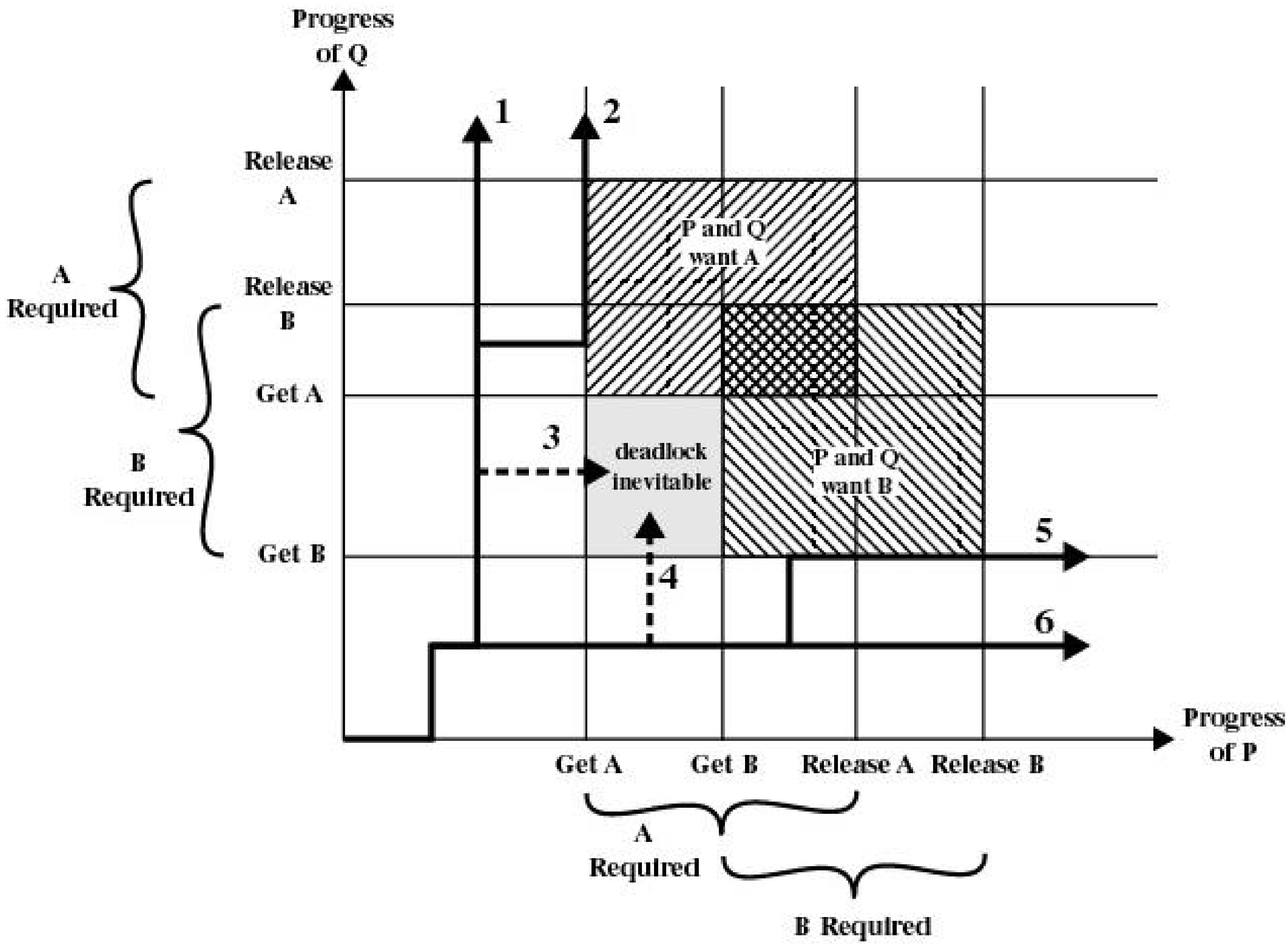


Figure 6.2 Example of Deadlock [BACO98]

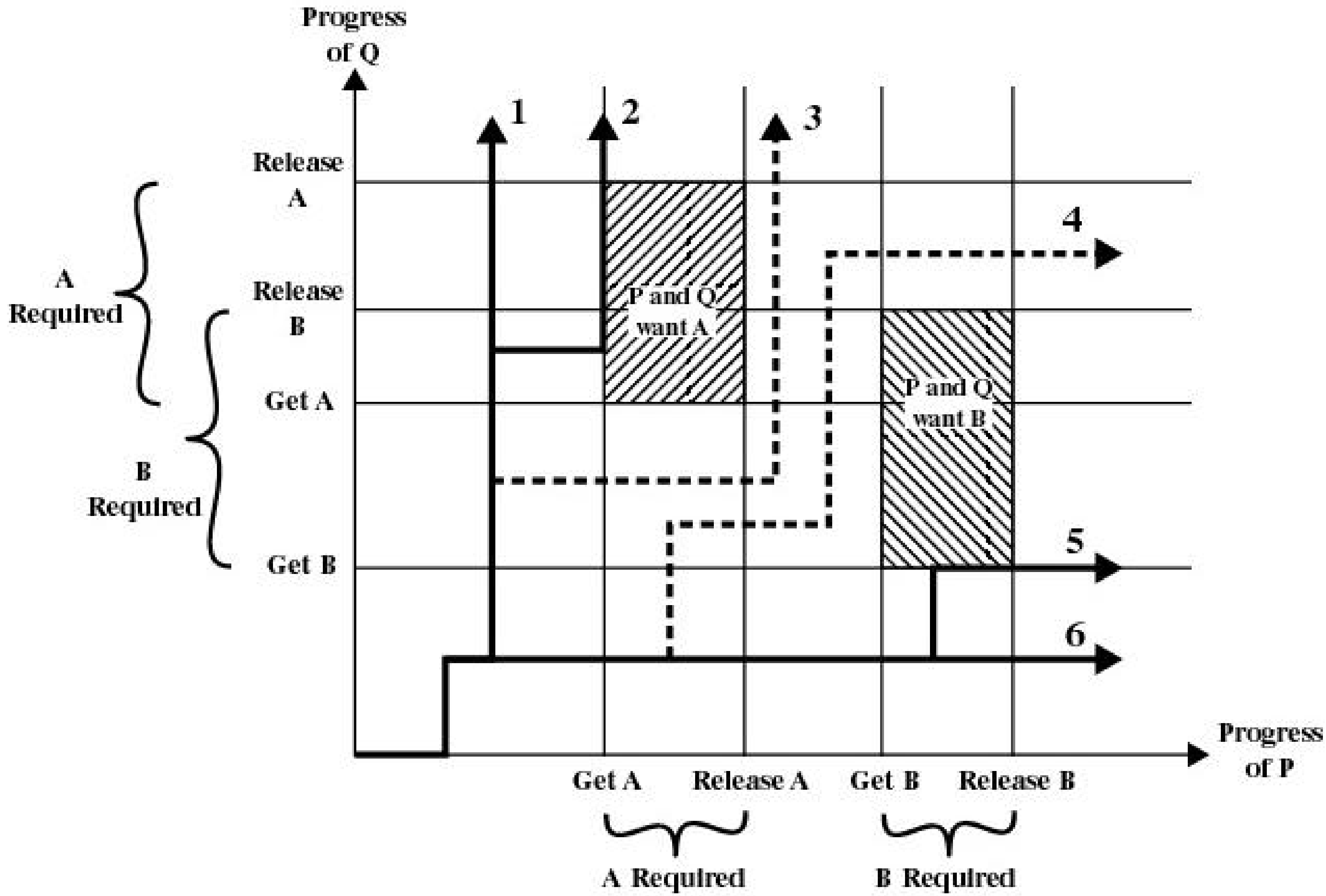


Figure 6.3 Example of No Deadlock [BACO98]

Reusable Resources

- Used by one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processor time, I/O channels, main and secondary memory, 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

Example of Deadlock

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

P1	P2
. . .	. . .
Request 80K bytes;	Request 70K bytes;
. . .	. . .
Request 60K bytes;	Request 80K bytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources

- Created (produced) and destroyed (consumed) by a process
- 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	P2
. . .	. . .
Receive(P2);	Receive(P1);
. . .	. . .
Send(P2, M1);	Send(P1, M2);

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
  - consequence of the first three conditions
- Four conditions constitute necessary and sufficient conditions for deadlock

Circular Wait

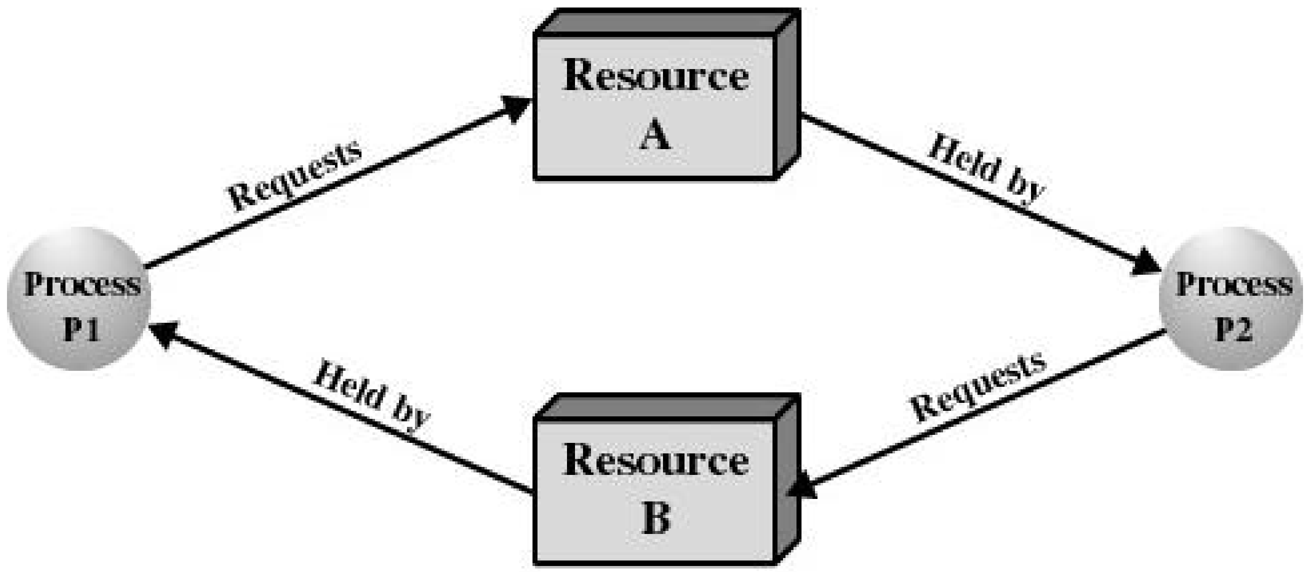


Figure 6.5 Circular Wait

Methods for Handling Deadlocks

- Deadlock prevention
  - design a system in such a way that the possibility of deadlock is excluded
- Deadlock avoidance
  - allows the three necessary conditions, but assures that deadlock point is never reached
- Deadlock detection
  - detects the deadlock asap
- Deadlock recovery

Deadlock Prevention

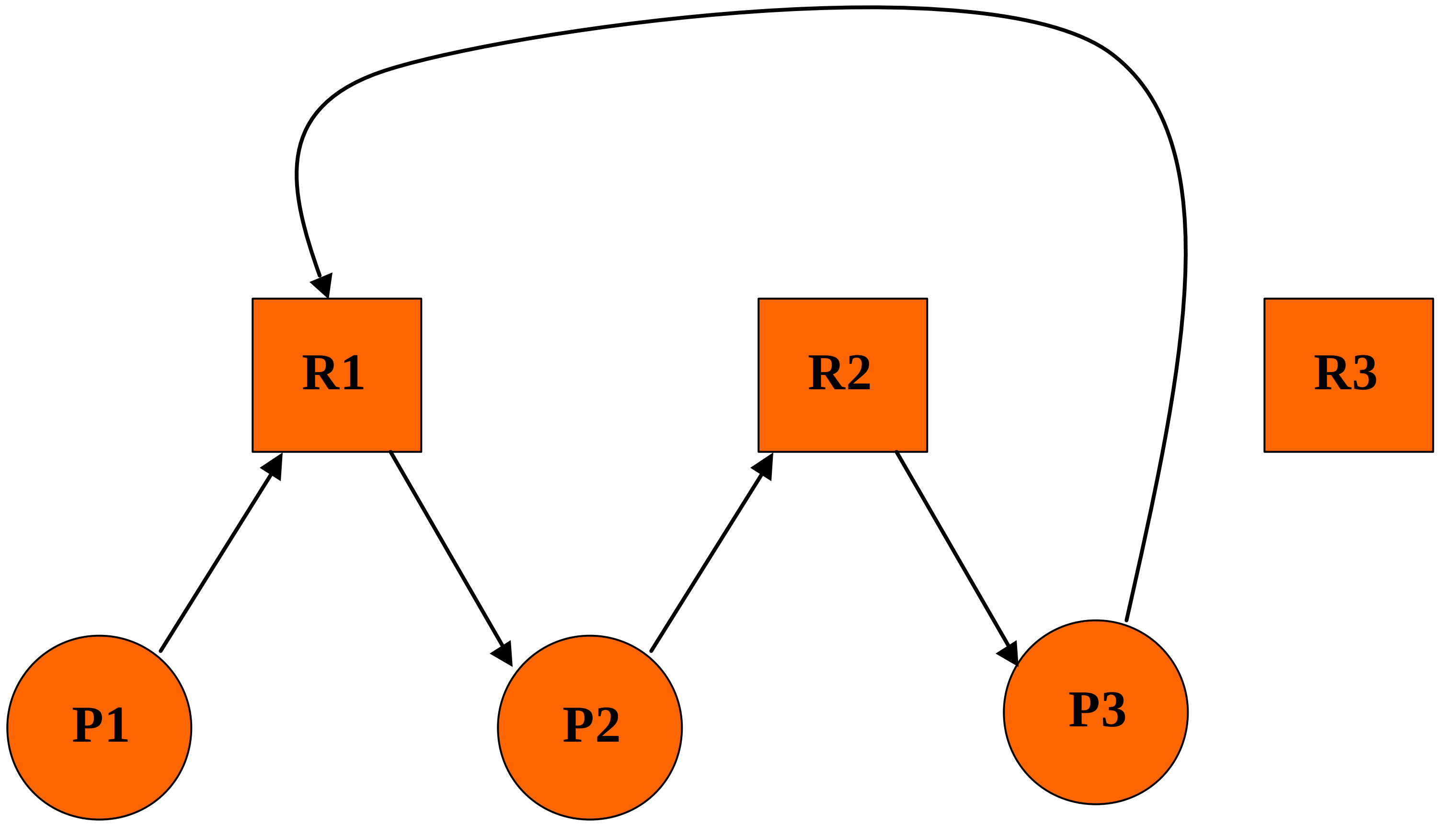
- Denying Mutual Exclusion
  - unthinkable
- Denying Hold-and-Wait
  - require that a process request all its required resources at one time
    - block the process until all requests can be granted simultaneously
    - process may be held up for a long time waiting for all its requests
  - resources allocated to a process may remain unused for a long time. These resources could be used by other processes

Deadlock Prevention

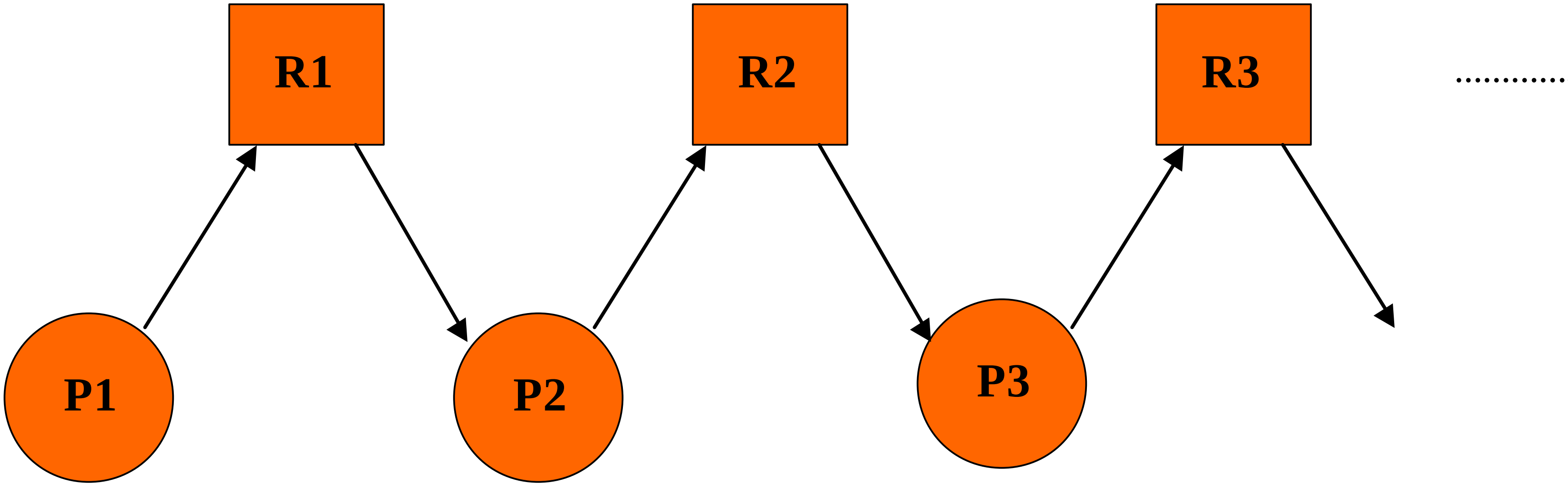
- Denying No preemption
  - if a process is denied a further request, the process must release the original resources
  - if a process requests a resource that is currently held by another process, OS may preempt the second process and require it to release its resources
  - practical only when the state can be easily saved and restored later, such as the processor

Deadlock Prevention

- Denying Circular wait
  - define a linear ordering for resources
  - once a resource is obtained, only those resources higher in the list can be obtained
  - may deny resources unnecessarily



Deadlock situation



No deadlock situation

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

- Referred to as the banker's algorithm
- State of the system is the current allocation of resources to process
  - Safe state is one in which there is at least one order in which all the processes can be run to completion without resulting in a deadlock
  - Unsafe state is a state that is not safe

Determination of a Safe State - Initial State

Claim Matrix

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

Allocation Matrix

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

Resource Vector

R1	R2	R3
9	3	6

Available Vector

R1	R2	R3
0	1	1

(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

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

Allocation Matrix

R1	R2	R3
6	2	3

Available Vector

(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

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

Allocation Matrix

R1	R2	R3
7	2	3

Available Vector

(c) P1 runs to completion

Determination of a Safe State - P3 Runs to Completion

Claim Matrix

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

Allocation Matrix

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

Available Vector

R1	R2	R3
9	3	4

(d) P3 runs to completion

Determination of an Unsafe State

Claim Matrix

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

Allocation Matrix

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

Resource Vector

R1	R2	R3
9	3	6

Available Vector

R1	R2	R3
1	1	2

(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

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

Allocation Matrix

R1	R2	R3
0	1	1

Available Vector

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

Banker's algorithm

- Pros and Cons
  - no rollback and no preemption
  - less restrictive than deadlock prevention
  - maximum resource requirement for each process must be stated in advance
  - the process under consideration must be independent
  - there must be a fixed number of resources to allocate
  - no process may exit while holding resources

Deadlock Detection

- Deadlock prevention is very conservative
  - limiting access to resources
  - imposing restrictions on processes
- Deadlock detection
  - requested resources are granted to processes whenever possible
  - periodically, OS performs detection algorithm

Deadlock Detection

- Detection algorithm
  - Allocation matrix, Resource vector, and Available vector
  - Request matrix Q is defined such that q_{ij} represents the amount of resources of type j requested by process I
  - Initially, all processes are unmarked
  - A deadlock exists iff there are unmarked processes at the end of the algorithm

✍Detection algorithm

1. Mark each process that has a row in the Allocation matrix of all zeros
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 ith row of Q is less than or equal to W. 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. Return to step 3

Deadlock Detection

Request Matrix Q

	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

Allocation Matrix A

	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

Resource Vector

R1	R2	R3	R4	R5
2	1	1	2	1

Available Vector

R1	R2	R3	R4	R5
0	0	0	0	1

Figure 6.9 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

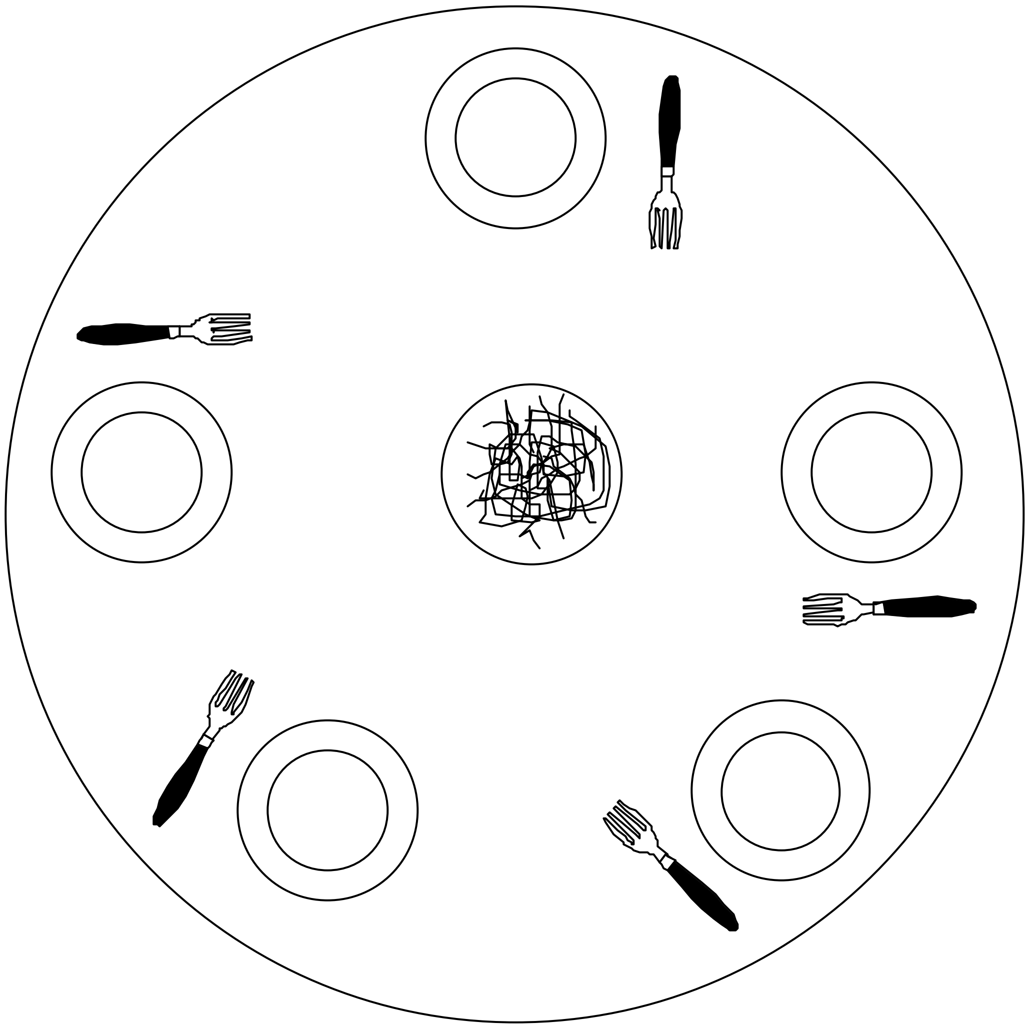
Selection Criteria Deadlocked Processes

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

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

Principle	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
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary •Subject to cyclic restart
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Preempts without much use •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 •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



```
/* program dining philosophers */
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 6.11 A first solution to the Dining Philosophers Problem

Figure 6.12 A second solution to the Dining Philosophers Problem

```
/* program dining philosophers */
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));
}
```

Dining Philosophers Problem

- Possible solutions
  - buy 5 additional forks(more sanitary solution)
  - teach the philosophers to eat spaghetti with just one fork
  - only allow 4 philosophers at a time into the dining room
  - arrange the seating of lefties and righties with at least one of each

UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

UNIX Concurrency Mechanisms

- Pipes
  - circular buffer allowing two processes to communicate
  - queue written by one process and read by another
  - operating system enforces mutual exclusion for writing and reading the pipe
  - write requests are immediately executed if there is room in the pipe, otherwise the process is blocked
  - read request is blocked if attempts to read more bytes than currently in the pipe

UNIX Concurrency Mechanisms

- Messages
  - block of text with accompanying type
  - receiver can either retrieve messages in FIFO order or by type
  - process suspends when trying to send a message to a full queue
  - process suspends when reading from an empty queue

UNIX Concurrency Mechanisms

- Shared memory
  - common block of virtual memory shared by multiple processes
  - fast form of interprocess communication
  - mutual exclusion must be provided by the processes, not the operating system

UNIX Concurrency Mechanisms

- Semaphores
  - wait and signal
  - operating system handles all these requests
- Signals
  - software mechanism that informs a process of the occurrence of asynchronous events

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	SIGFPT	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	SIGALARM	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	SIGCLD	Death of a child
19	SIGPWR	Power failure

Solaris Thread Synchronization Primitives

- Mutual exclusion(mutex) lock
  - prevents more than one thread from proceeding when the lock is acquired
- Semaphores
  - used for incrementing and decrementing

Solaris Thread Synchronization Primitives

- Multiple readers, single writer (readers/writer) locks
  - multiple threads have simultaneous read-only access
  - single thread has access for writing
- Condition variables
  - used to wait until a particular condition is true

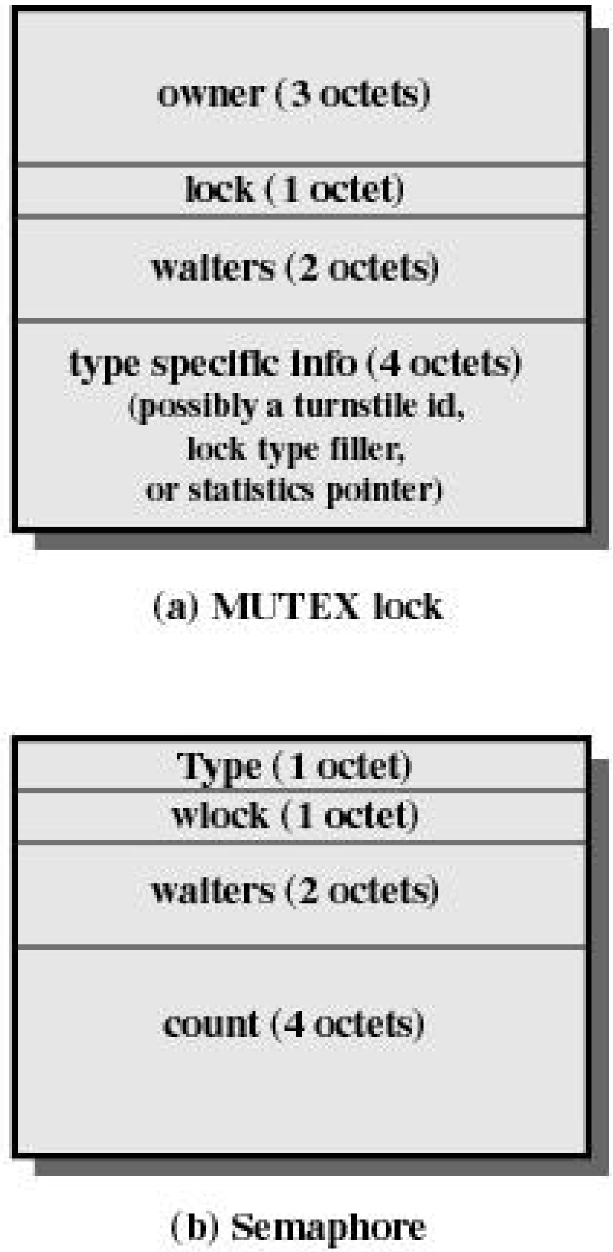
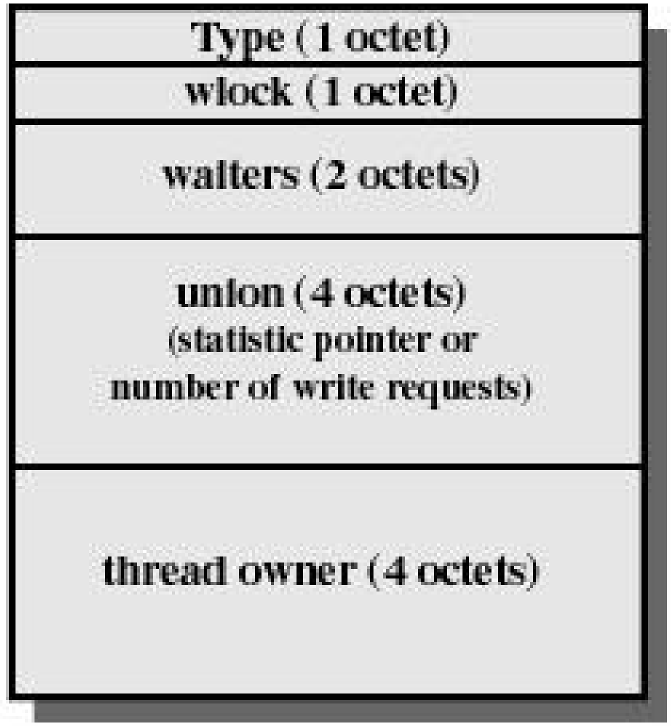


Figure 6.13 Solaris Synchronization Data Structures



(c) Reader/writer lock

(d) Condition variable

Figure 6.13 Solaris Synchronization Data Structures

Windows 2000 Concurrency Mechanisms

- Synchronization Objects
  - Process
  - Thread
  - File
  - Console input
  - File change notification
  - Mutex
  - Semaphore
  - Event
  - Waitable timer

Table 6.3 Windows 2000 Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
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
File	An instance of an opened file or I/O device	I/O operation completes	All 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
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
Mutex	A mechanism that provides mutual exclusion capabilities for the Win32 and OS/2 environments	Owning thread or other thread releases the mutant	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Event	An announcement that a system event has occurred	Thread sets the event	All released
Waitable Timer	A counter that records the passage of time	Set time arrives or time interval expires	All released

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