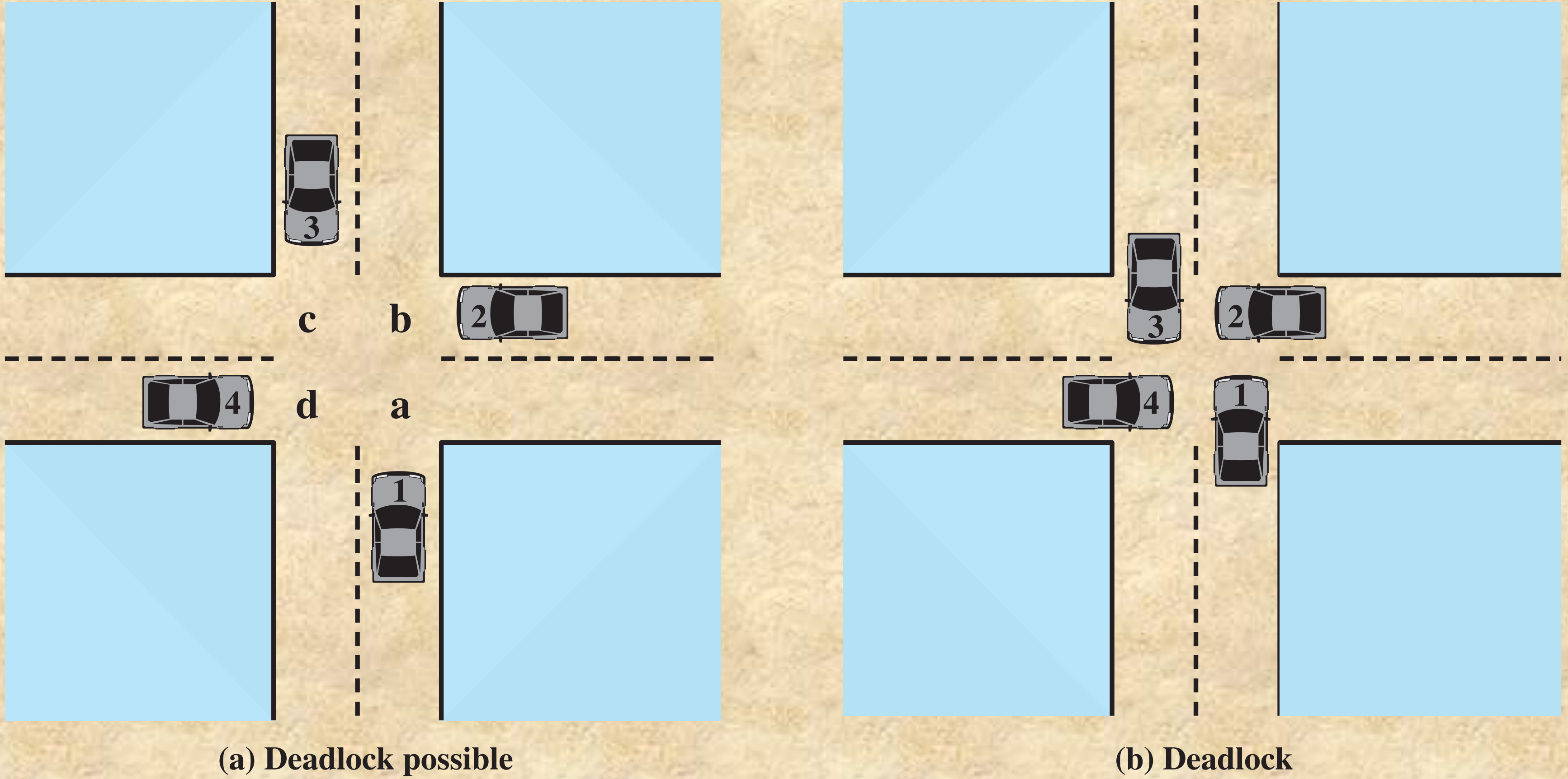
Operating Systems: Internals and Design Principles

Chapter 6
Concurrency: Deadlock and Starvation

Ninth Edition
By William Stallings

Deadlock

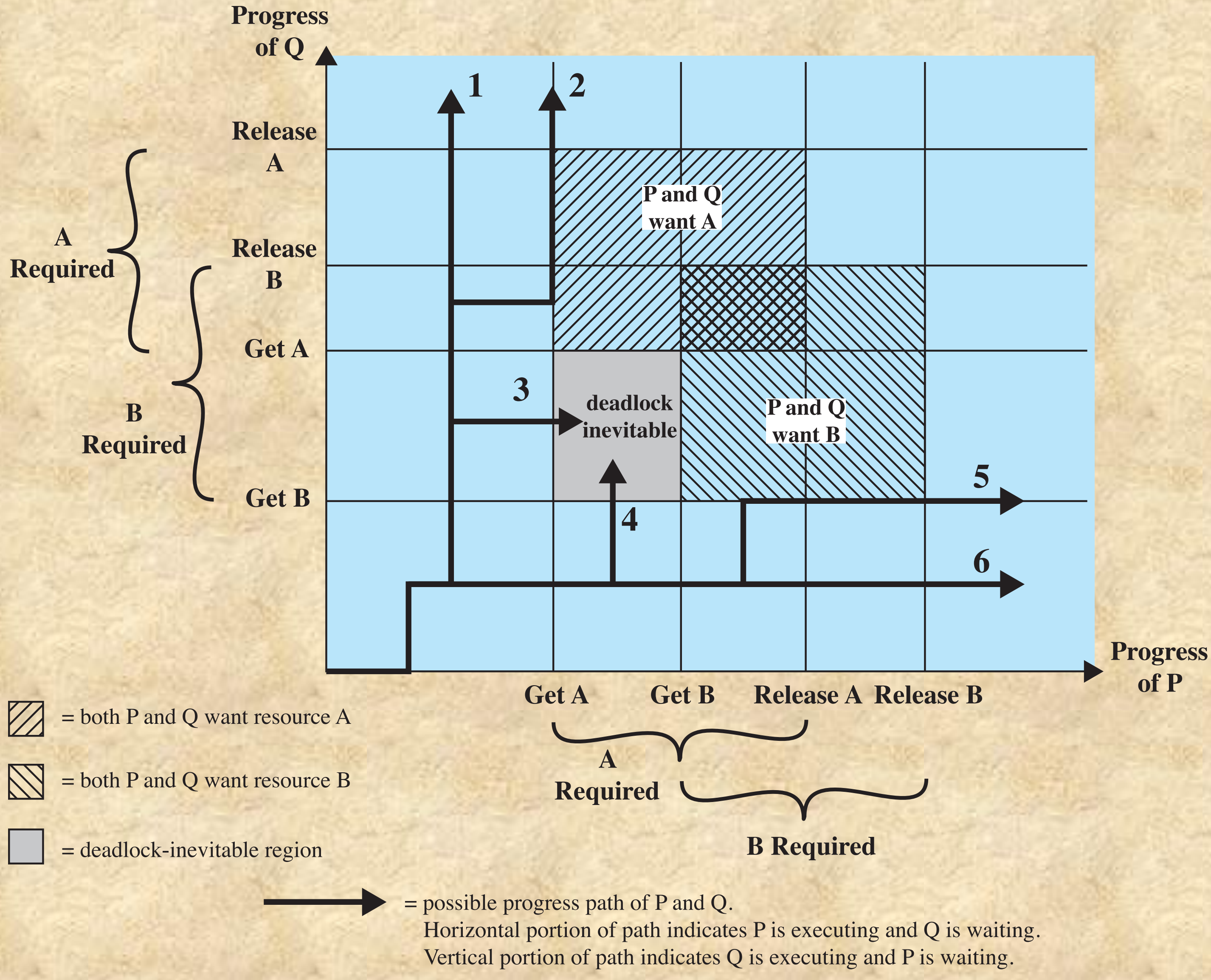
- The *permanent* blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent because none of the events is ever triggered
- No efficient solution in the general case



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock



▨ = both P and Q want resource A

▧ = both P and Q want resource B

⬜ = deadlock-inevitable region

→ = possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Figure 6.2 Example of Deadlock

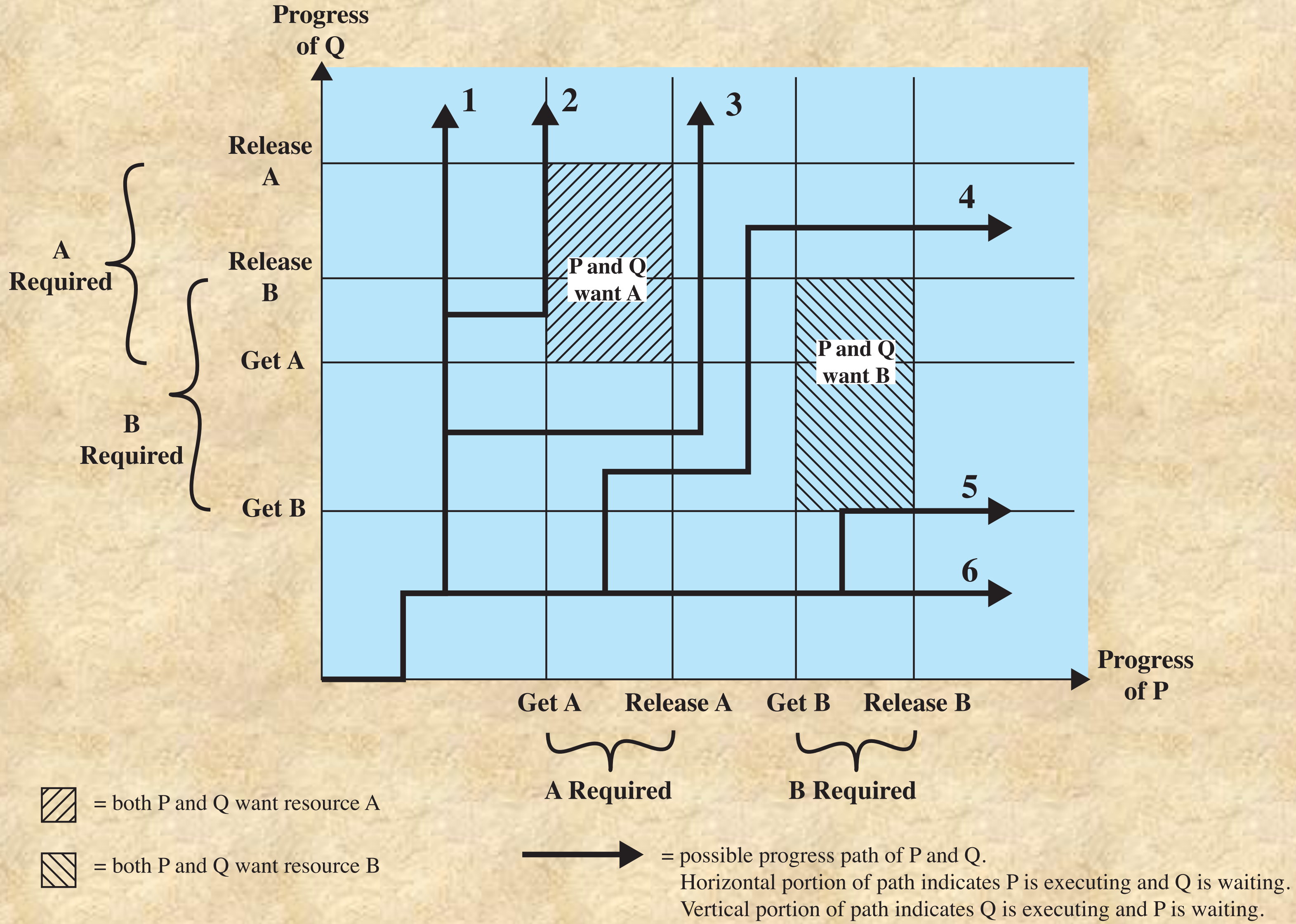


Figure 6.3 Example of No Deadlock

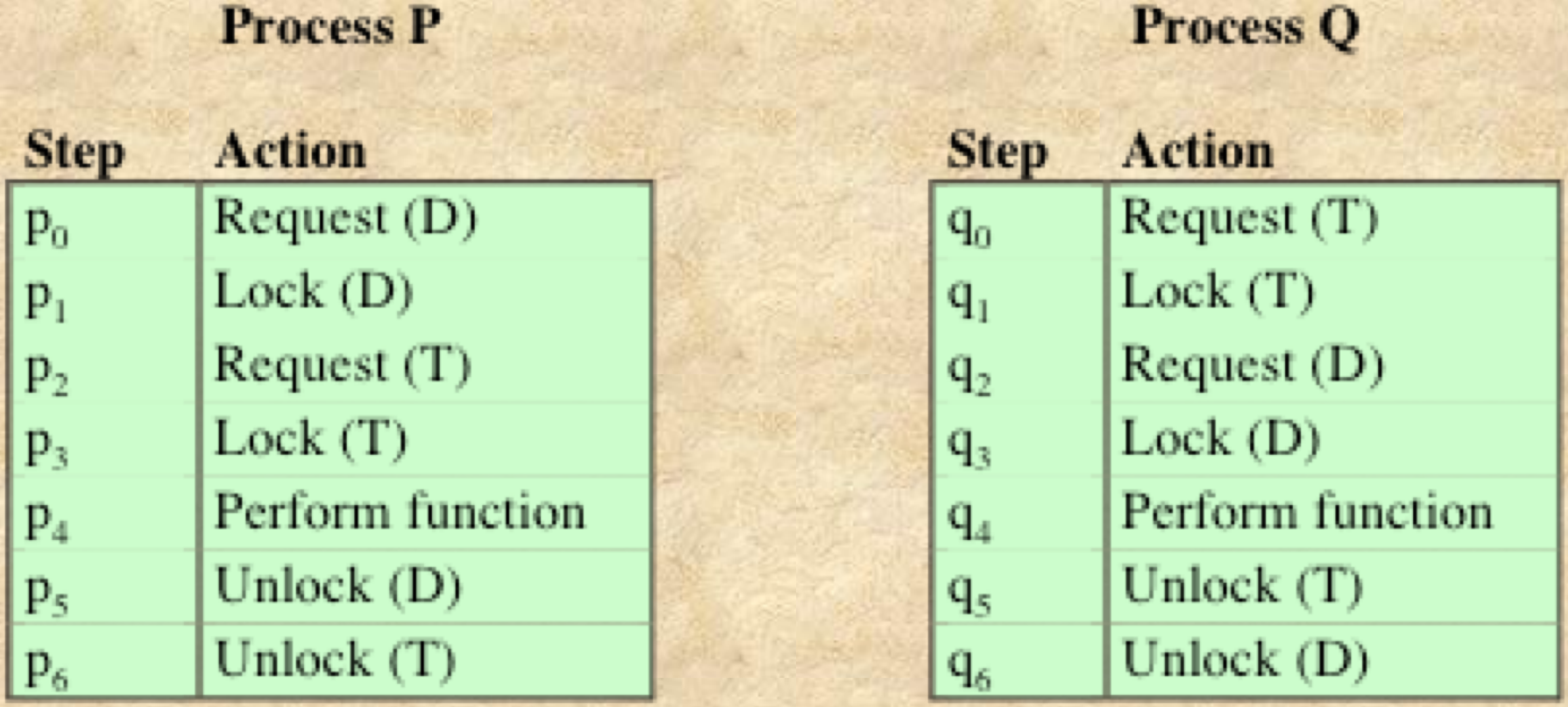
Resource Categories

Reusable

- Can be safely used by only one process at a time and is not depleted by that use
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- One that can be created (produced) and destroyed (consumed)
  - Interrupts, signals, messages, and information
  - In I/O buffers



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 2: Memory Request

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

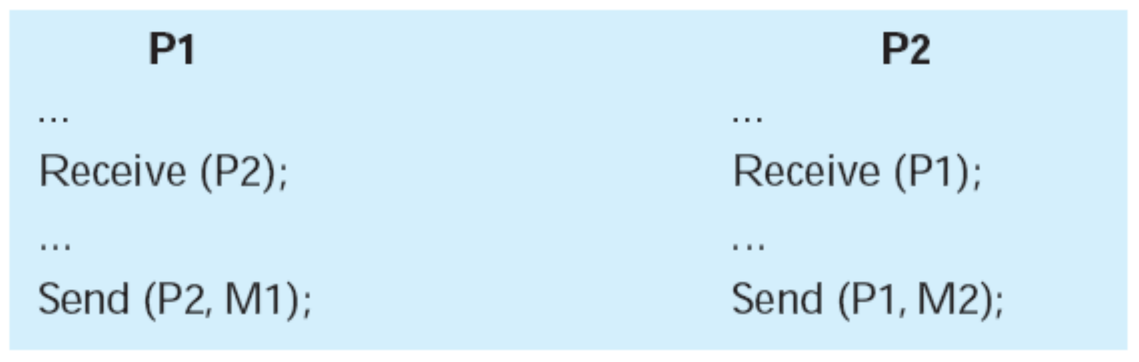
```
P1
 . . .
Request 80 Kbytes;
 . . .
Request 60 Kbytes;
```

```
P2
 . . .
Request 70 Kbytes;
 . . .
Request 80 Kbytes;
```

- Deadlock occurs if both processes progress to their second request

Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:



```
P1                         P2
...                        ...
Receive (P2);              Receive (P1);
...                        ...
Send (P2, M1);             Send (P1, M2);
```

- Deadlock occurs if the Receive is blocking

Deadlock Approaches

- There is no single effective strategy that can deal with all types of deadlock
- Three approaches are common:

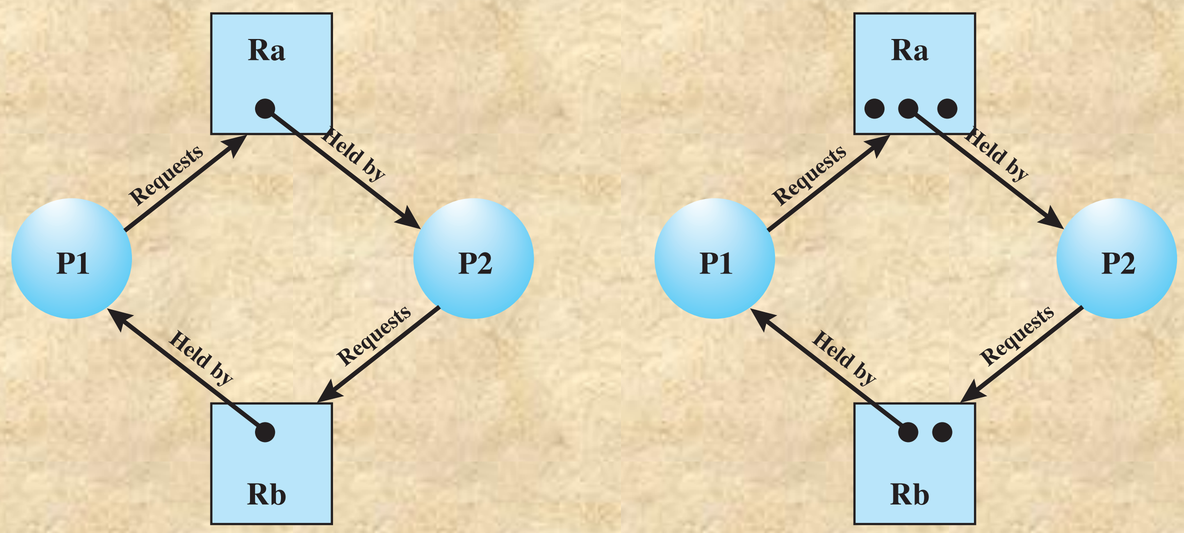
- **Deadlock prevention**
    - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening

- **Deadlock avoidance**
    - Do not grant a resource request if this allocation might lead to deadlock

- **Deadlock detection**
    - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

(a) Resouce is requested

(b) Resource is held



(c) Circular wait

(d) No deadlock

Figure 6.5 Examples of Resource Allocation Graphs

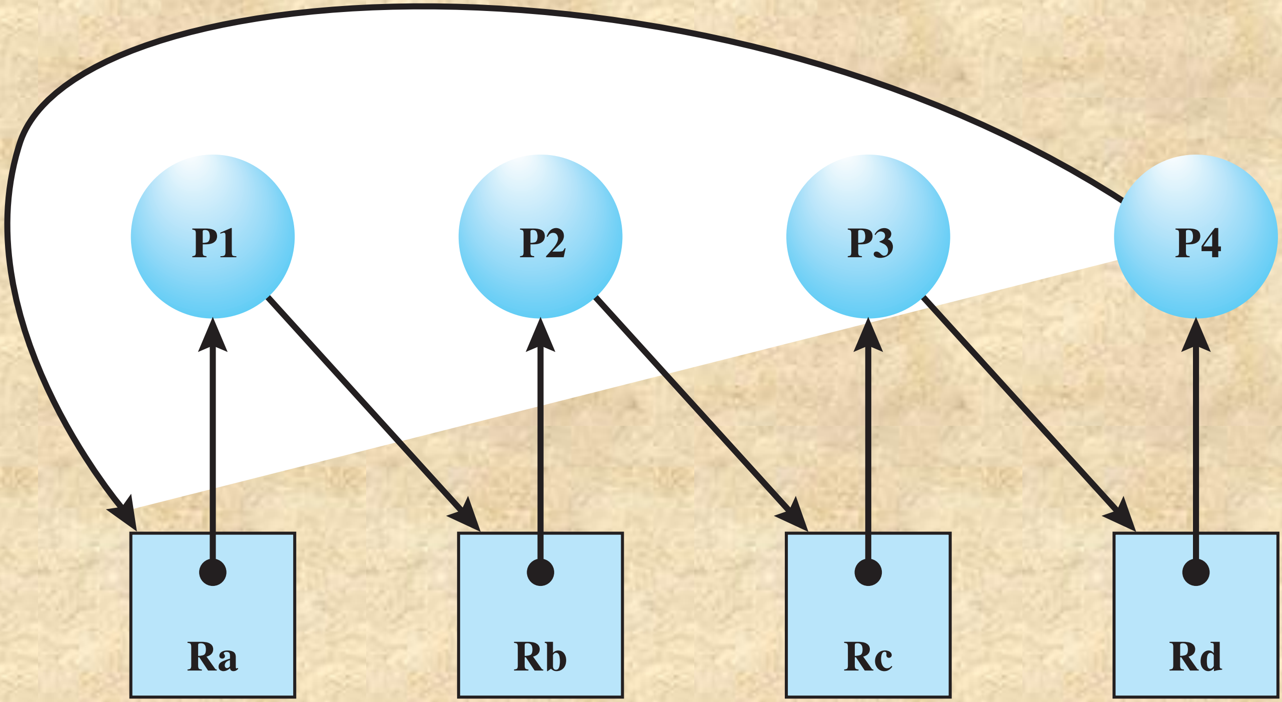


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• Only one process may use a resource at a time • No process may access a resource until that has been allocated to another process	• A process may hold allocated resources while awaiting assignment of other resources	• No resource can be forcibly removed from a process holding it	• A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
  - Indirect
    - Prevent the occurrence of one of the three necessary conditions
  - Direct
    - Prevent the occurrence of a circular wait

Deadlock Condition Prevention

- Mutual exclusion
  - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS
  - Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes
  - Even in this case, deadlock can occur if more than one process requires write permission
- Hold and wait
  - Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously

Deadlock Condition Prevention

- No Preemption
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OS may preempt the second process and require it to release its resources
- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types

Deadlock Avoidance

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- Requires knowledge of future process requests

Two Approaches to Deadlock Avoidance

Deadlock Avoidance

Resource Allocation Denial

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

Process Initiation Denial

- Do not start a process if its demands might lead to deadlock

Resource Allocation Denial

- Referred to as the *banker's algorithm*
- ***State*** of the system reflects the current allocation of resources to processes
- ***Safe state*** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock
- ***Unsafe state*** is a state that is not safe

	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

Figure 6.7 Determination of a Safe State

	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

Figure 6.7 Determination of a Safe State

	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

Figure 6.7 Determination of a Safe State

	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

Figure 6.7 Determination of a Safe 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

	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

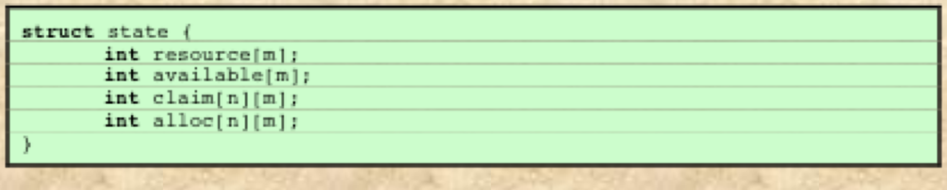
Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

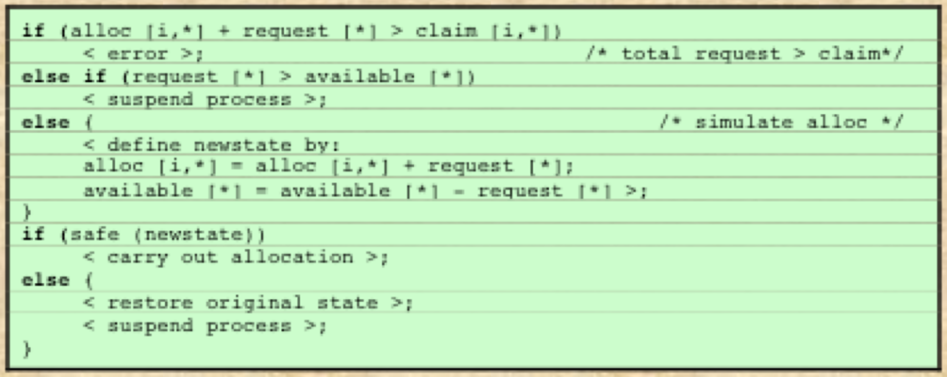
(b) P1 requests one unit each of R1 and R3

Figure 6.8 Determination of an Unsafe State



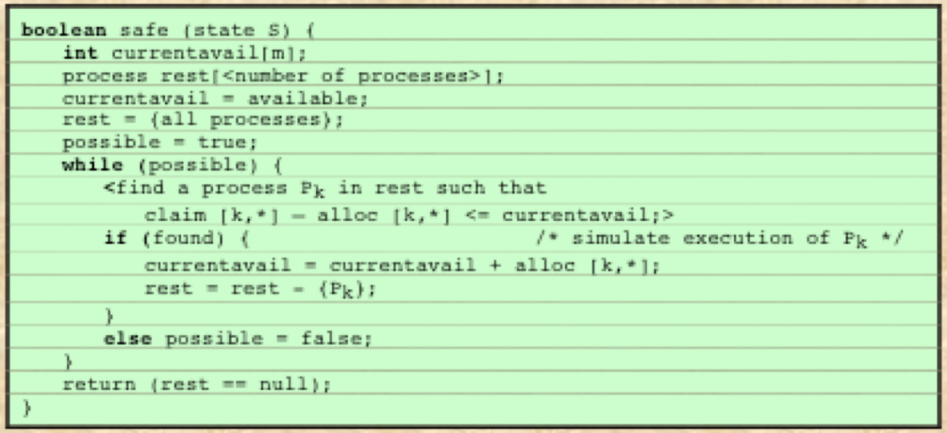
```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(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 Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

Deadlock Avoidance Restrictions

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

Deadlock Strategies

Deadlock prevention strategies are very conservative

- Limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

- Resource requests are granted whenever possible

Deadline Detection Algorithm

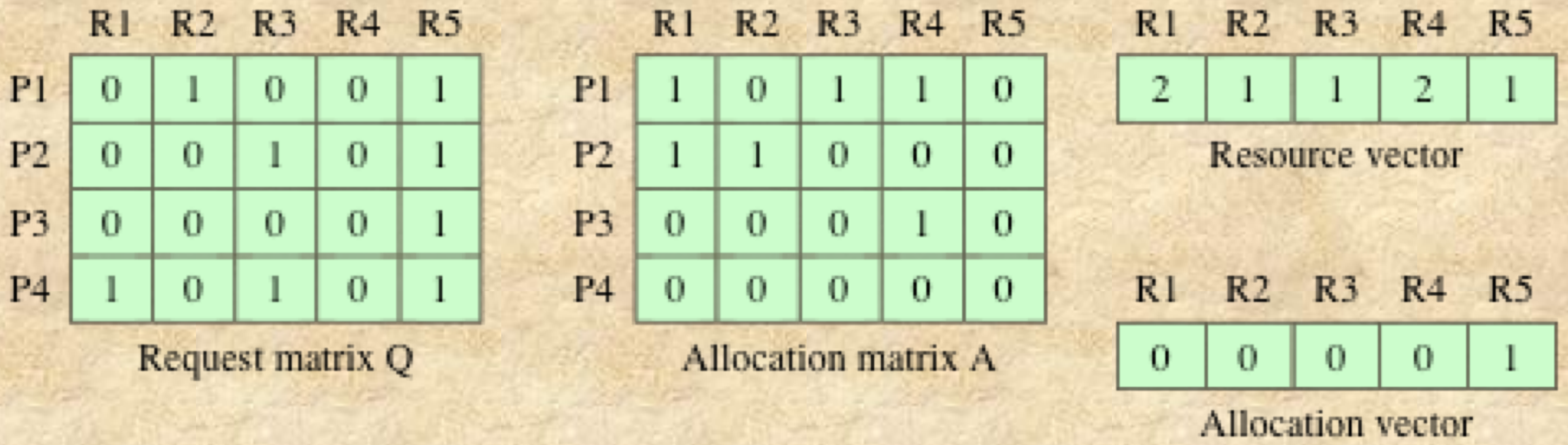
- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur

Advantages:

- It leads to early detection
- The algorithm is relatively simple

Disadvantage

- Frequent checks consume considerable processor time



	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

Recovery Strategies

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

Integrated Deadlock Strategy

- Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use different strategies in different situations
    - Group resources into a number of different resource classes
    - Use the linear ordering strategy defined previously for the prevention of circular wait to prevent deadlocks between resource classes
    - Within a resource class, use the algorithm that is most appropriate for that class

- Classes of resources
    - Swappable space
        - Blocks of memory on secondary storage for use in swapping processes
    - Process resources
        - Assignable devices, such as tape drives, and files
    - Main memory
        - Assignable to processes in pages or segments
    - Internal resources
        - Such as I/O channels

Class Strategies

- Within each class the following strategies could be used:
  - **Swappable space**
    - Prevention of deadlocks by requiring that all of the required resources that may be used be allocated at one time, as in the hold-and-wait prevention strategy
    - This strategy is reasonable if the maximum storage requirements are known
  - **Process resources**
    - Avoidance will often be effective in this category, because it is reasonable to expect processes to declare ahead of time the resources that they will require in this class
    - Prevention by means of resource ordering within this class is also possible
  - **Main memory**
    - Prevention by preemption appears to be the most appropriate strategy for main memory
    - When a process is preempted, it is simply swapped to secondary memory, freeing space to resolve the deadlock
  - **Internal resources**
    - Prevention by means of resource ordering can be used

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)

- No philosopher must starve to death (avoid deadlock and starvation)

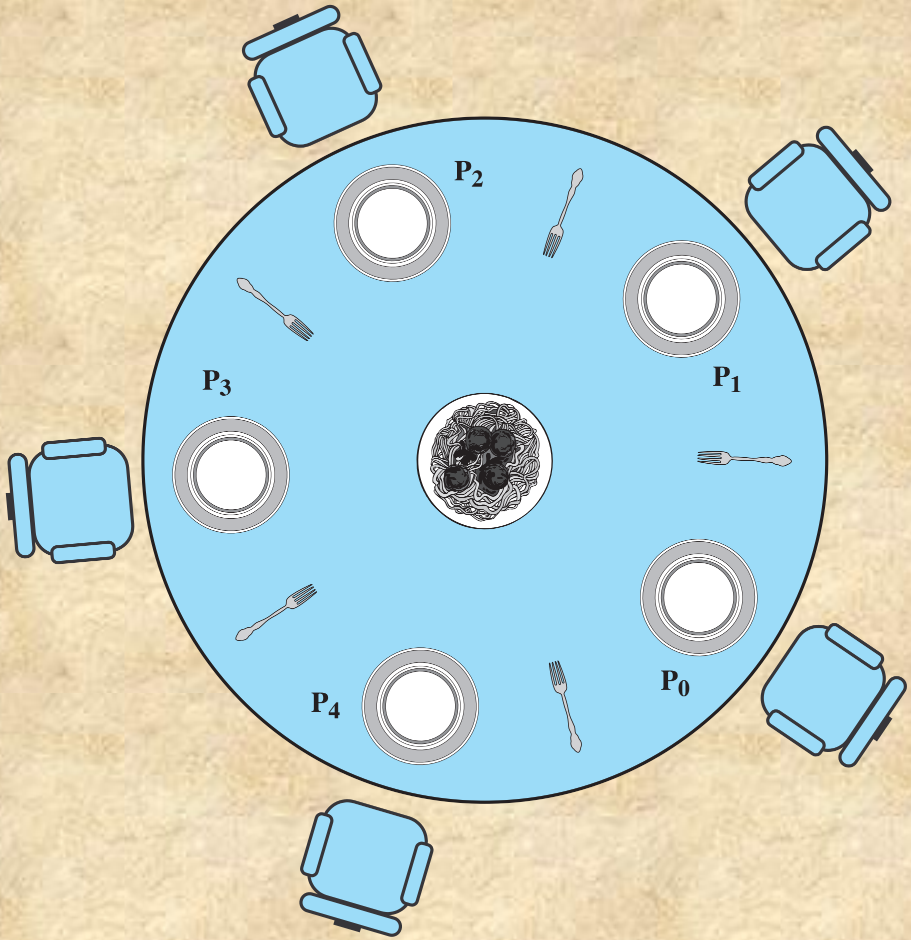
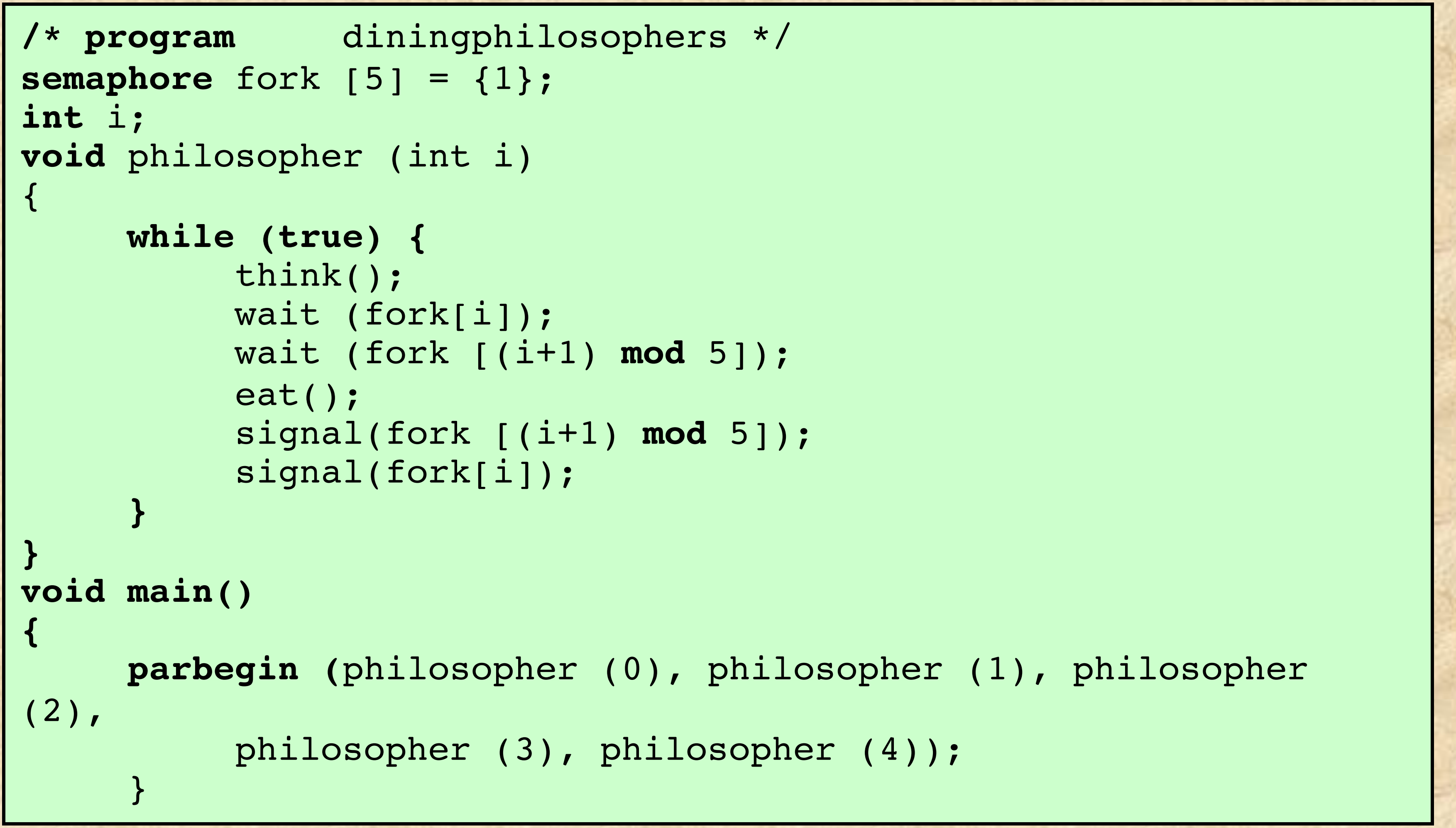


Figure 6.11 Dining Arrangement for Philosophers



```
/* 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

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Pipes

Messages

Shared memory

Semaphores

Signals

Pipes

- Circular buffers allowing two processes to communicate on the producer-consumer model
  - First-in-first-out queue, written by one process and read by another

Two types:

- Named
- Unnamed

Messages

- A block of bytes with an accompanying type
- UNIX provides ***msgsnd*** and ***msgrcv*** system calls for processes to engage in message passing
- Associated with each process is a message queue, which functions like a mailbox

Shared Memory

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

Semaphores

- Generalization of the `semWait` and `semSignal` primitives
  - No other process may access the semaphore until all operations have completed

Consists of:

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
    - Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
    - Performing some default action
    - Executing a signal-handler function
    - Ignoring the signal

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

Table 6.2

UNIX Signals

(Table can be found on page 288 in textbook)

Atomic Operations

- Atomic operations execute without interruption and without interference
- Simplest of the approaches to kernel synchronization
- Two types:

Integer Operations

- Operate on an integer variable
- Typically used to implement counters

Bitmap Operations

- Operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable

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
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

Table 6.2

Linux Atomic Operations

(Table can be found on page 289 in textbook)

Spinlocks

- Most common technique for protecting a critical section in Linux
- Can only be acquired by one thread at a time
    - Any other thread will keep trying (spinning) until it can acquire the lock
- Built on an integer location in memory that is checked by each thread before it enters its critical section
- Effective in situations where the wait time for acquiring a lock is expected to be very short
- Disadvantage:
    - Locked-out threads continue to execute in a busy-waiting mode

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 6.4 Linux Spinlocks

(Table can be found on page 291 in textbook)

Semaphores

- User level:
  - Linux provides a semaphore interface corresponding to that in UNIX SVR4
- Internally:
  - Implemented as functions within the kernel and are more efficient than user-visable semaphores
- Three types of kernel semaphores:
  - Binary semaphores
  - Counting semaphores
  - Reader-writer semaphores

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)	Initializes 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

(Table can be found on page 293 in textbook)

Readers/Writer Locks

- Allows multiple threads to have simultaneous read-only access to an object protected by the lock

- Allows a single thread to access the object for writing at one time, while excluding all readers
  - When lock is acquired for writing it takes on the status of `write lock`
  - If one or more readers have acquired the lock its status is `read lock`

Condition Variables

A condition variable is used to wait until a particular condition is true

Condition variables must be used in conjunction with a mutex lock

Summary

- Principles of deadlock
    - Reusable/consumable resources
    - Resource allocation graphs
    - Conditions for deadlock
- Deadlock prevention
    - Mutual exclusion
    - Hold and wait
    - No preemption
    - Circular wait
- Deadlock avoidance
    - Process initiation denial
    - Resource allocation denial
- Deadlock detection
    - Deadlock detection algorithm
    - Recovery
- Android interprocess communication
- Integrated deadlock strategy
- UNIX concurrency mechanisms
    - Pipes
    - Messages
    - Shared memory
    - Semaphores
    - Signals
- Linux kernel concurrency mechanisms
    - Atomic operations
    - Spinlocks
    - Semaphores
    - Barriers
- Solaris thread synchronization primitives
    - Mutual exclusion lock
    - Semaphores
    - Readers/writer lock
    - Condition variables
- Windows concurrency mechanisms
    - Wait functions
    - Dispatcher objects
    - Critical sections
    - Slim reader-writer locks
    - Lock-free synchronization