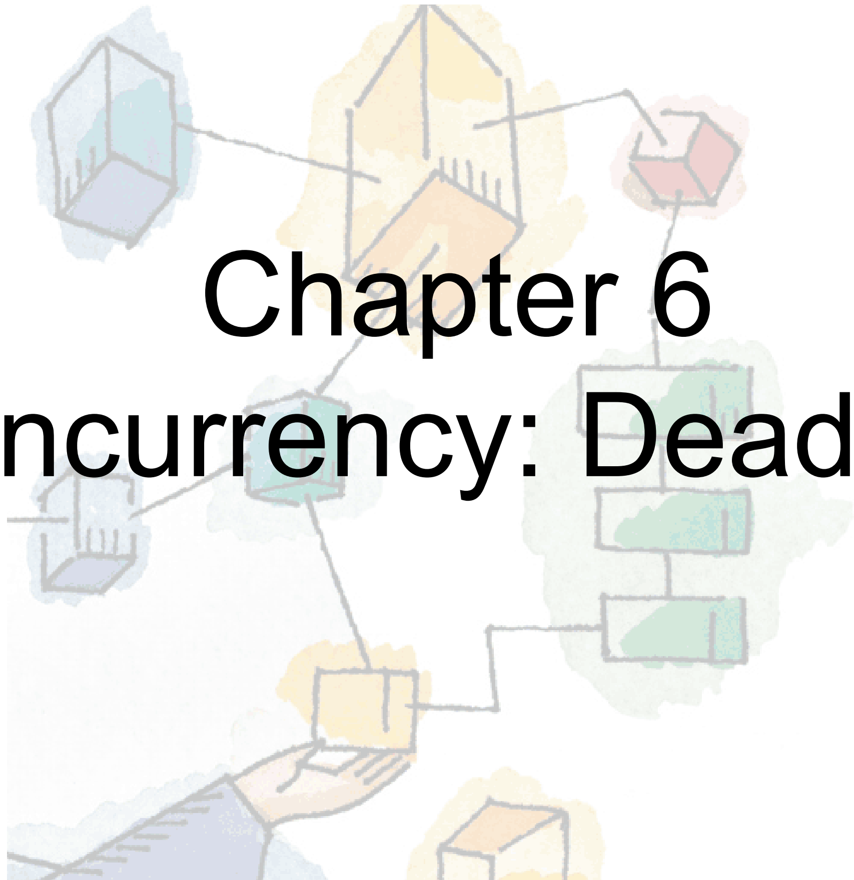
Operating Systems:
Internals and Design Principles
William Stallings



Chapter 6
Concurrency: Deadlock

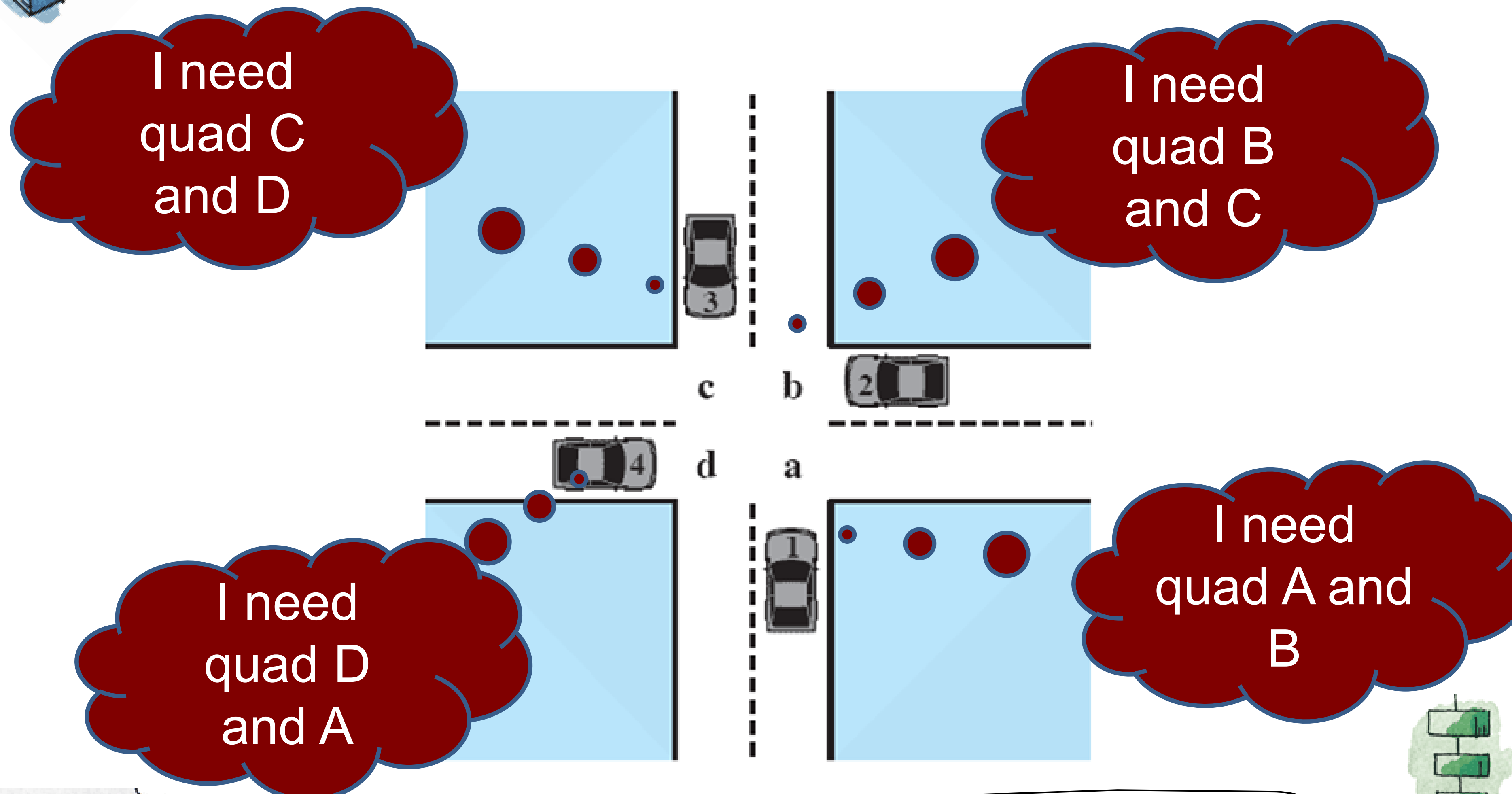
Outline

- Deadlock
- Resource allocation graphs
- Necessary conditions for deadlock
- Dealing with deadlock
  - Prevention
  - Avoidance
  - Discovery/recovery
- Dining philosophers problem

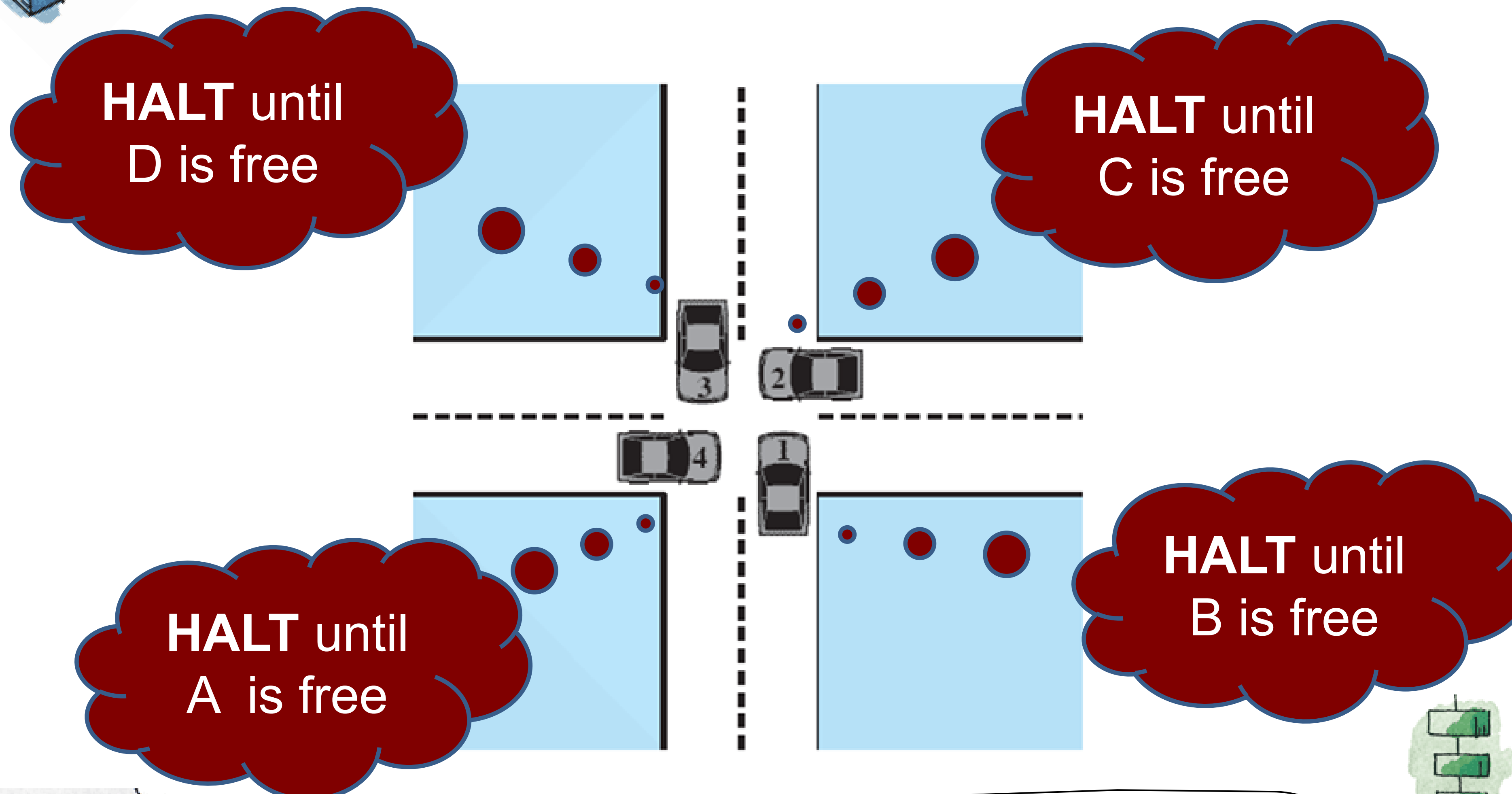
Deadlock

- Permanent blocking of a set of processes that 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
- No efficient solution

Potential Deadlock



Actual Deadlock



Reusable Resources

- Used by only one process at a time and not depleted by that use
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, and databases
- Processes obtain resources that they later release for reuse by other processes

Reusable Resources

- 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

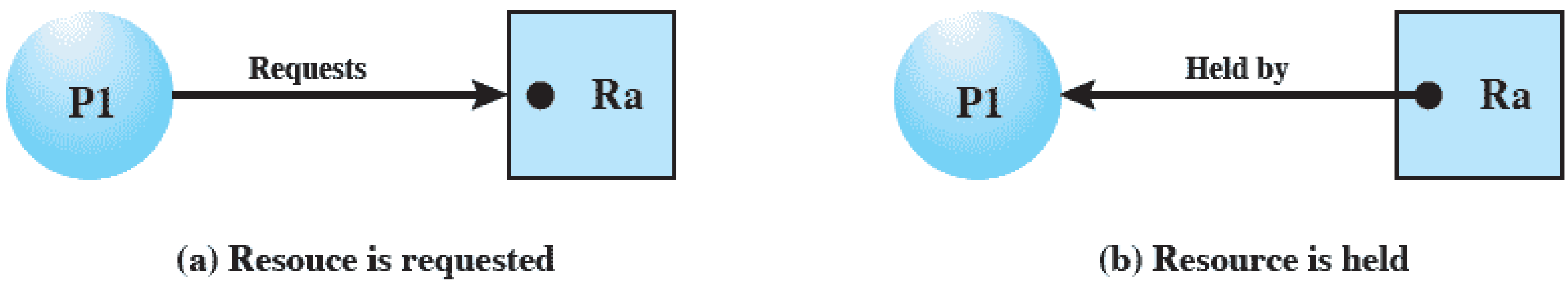
Consumable Resources

- Deadlock occurs if receives blocking

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

Resource Allocation Graphs

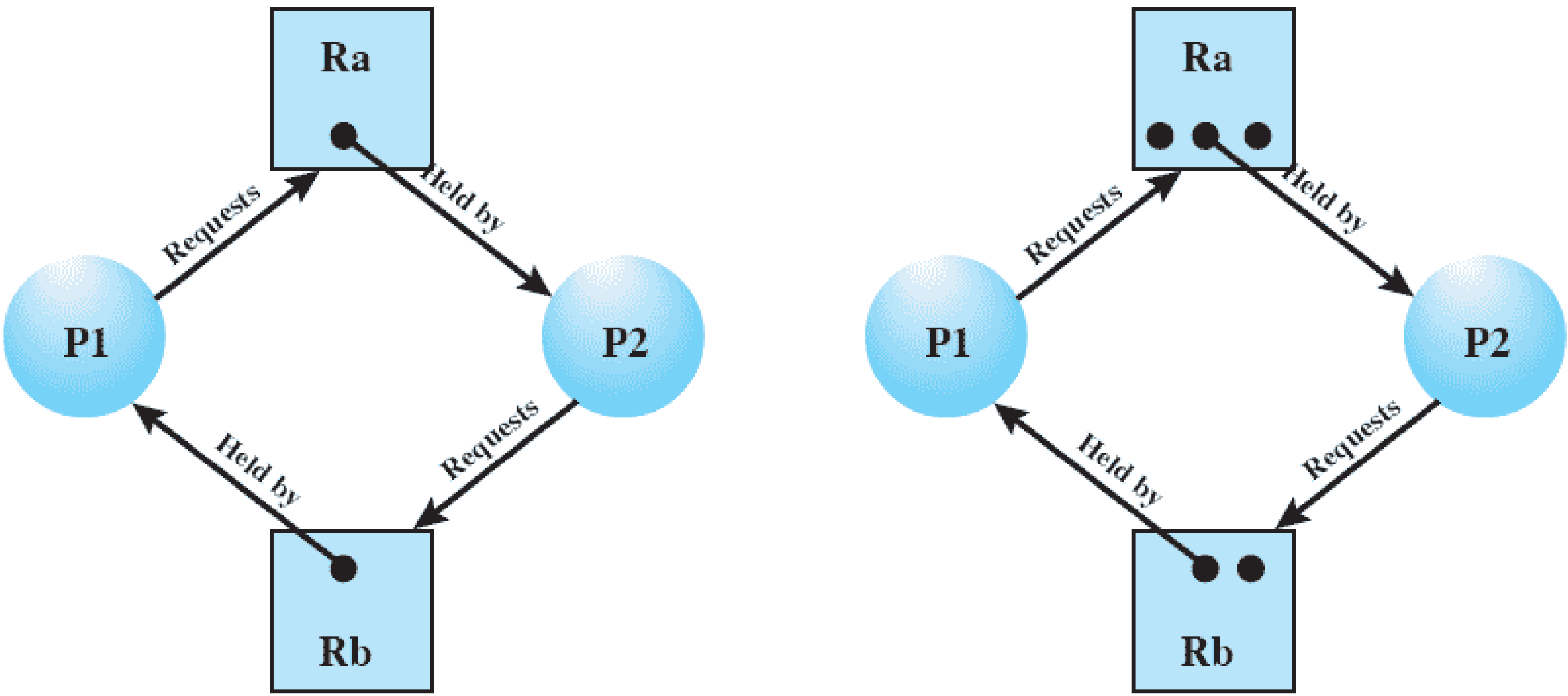
- Directed graph that depicts a state of the system of resources and processes



Resource Allocation Graphs

- A set of vertices *V* and a set of edges *E*.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graphs



(c) Circular wait

(d) No deadlock

Resource Allocation Graphs

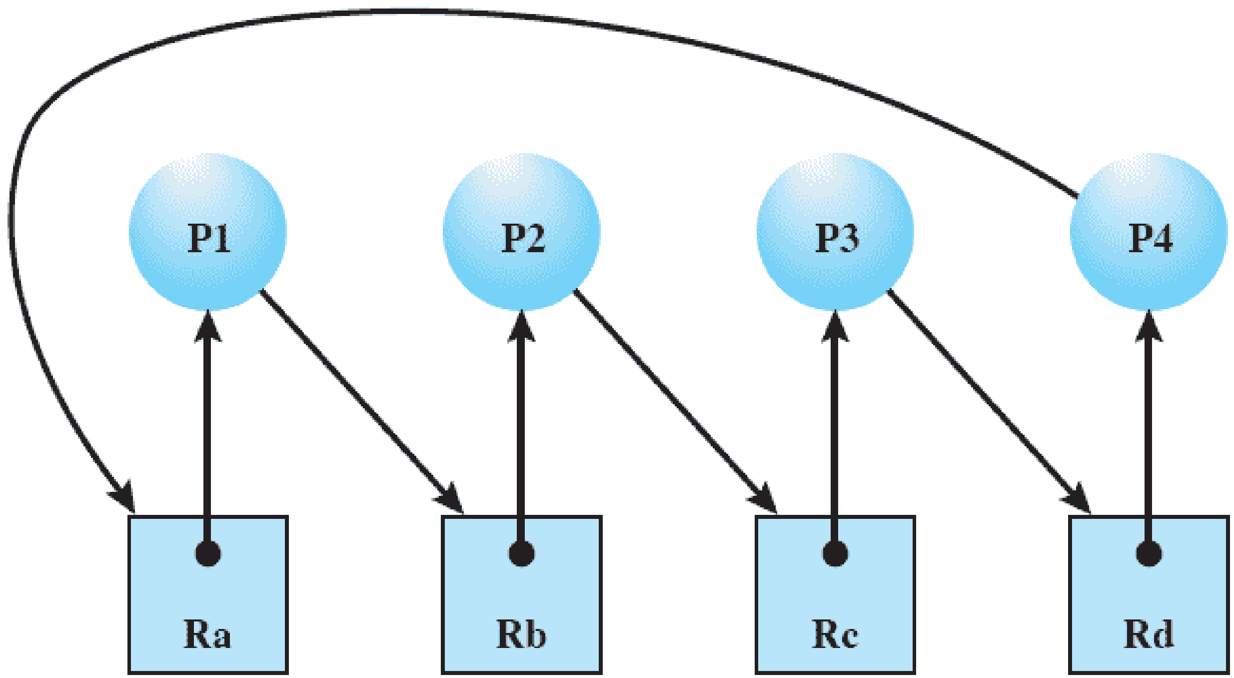


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock if no preemption.
  - if several instances per resource type, possibility of deadlock.

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• 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

Dealing with Deadlock

- Three general approaches exist for dealing with deadlock:

Prevent Deadlock

- adopt a policy that eliminates one of the conditions

Avoid Deadlock

- make the appropriate dynamic choices based on the current state of resource allocation

Detect Deadlock

- attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention

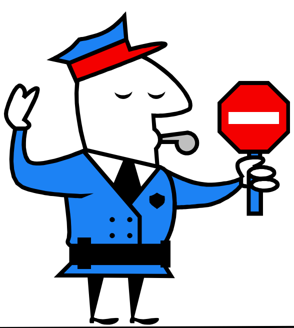
- Ensure that at least one of the necessary condition for deadlocks does not hold. Can be accomplished restraining the ways request can be made
  - Mutual Exclusion – Must hold for non-sharable resources that can be accessed simultaneously by various processes. Therefore cannot be used for prevention.
  - Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
    - Low resource utilization; starvation possible

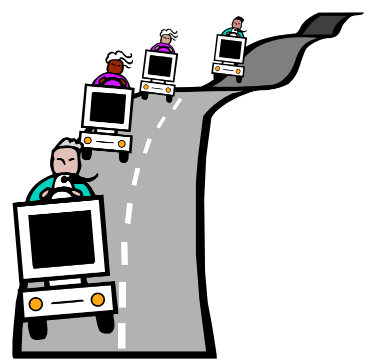
Deadlock Prevention

- No Preemption – not practical for many systems
  - If a process A that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by A are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- Circular Wait – Can be used in practice
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

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 requests





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
- No process may exit while holding resources

Deadlock Detection Algorithms

- 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
- Disadvantage
  - frequent checks consume considerable processor time

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)

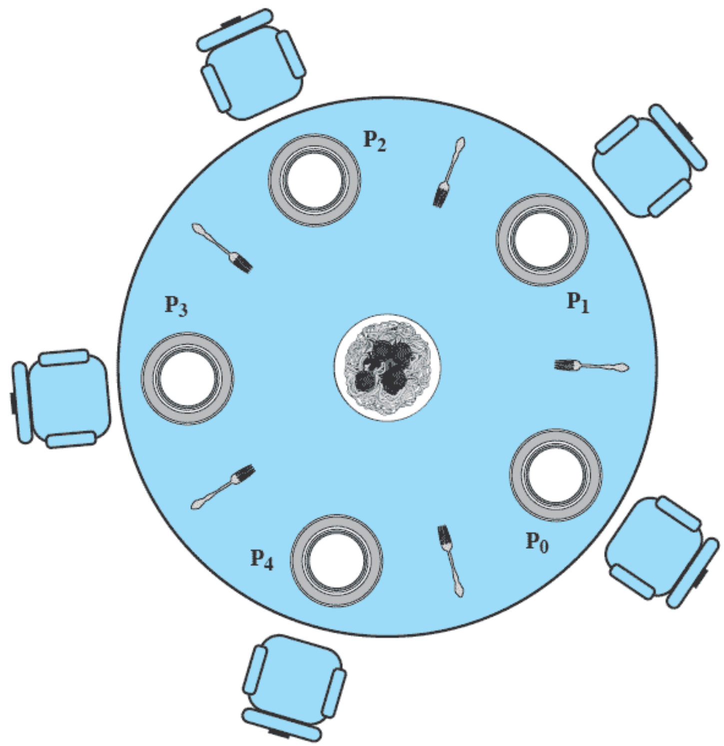


Figure 6.11 Dining Arrangement for Philosophers

Using Semaphores

```
/* 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));
     }
```

Warning: This solution could create a deadlock!

Figure 6.12 **A First Solution to the Dining Philosophers Problem**

A Second Solution

```
/* 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

Solution Using A Monitor

```
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]);
}
```

Solution Using A Monitor

```
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

- Suppose that at least one philosopher always picks up his left fork first (a “lefty”), and the others always pick up his right fork first (a “righty”). We can prove
  - Any seating arrangement of lefties and righties avoids a deadlock.
  - Any seating arrangement prevents starvation.
- Thus the deadlock situation can be prevented if one philosopher’s picking order is made different from others – No circular waiting!

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocess 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

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 6.1

UNIX Signals

(Table can be found on page 316 in textbook)

Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX plus:

Barriers

Spinlocks

Atomic Operations

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 317 in textbook)

Spinlocks

- 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

Windows 7 Concurrency Mechanisms

- Windows provides synchronization among threads as part of the object architecture

Important methods are:

- Executive dispatcher objects
- user mode critical sections
- slim reader-writer locks
- condition variables
- lock-free operations