K. J. Somaiya College of Engineering, Mumbai
(A constituent College of Somaiya Vidyavihar University)

Operating System

Module 3. Process Concurrency

Deadlock and Starvation

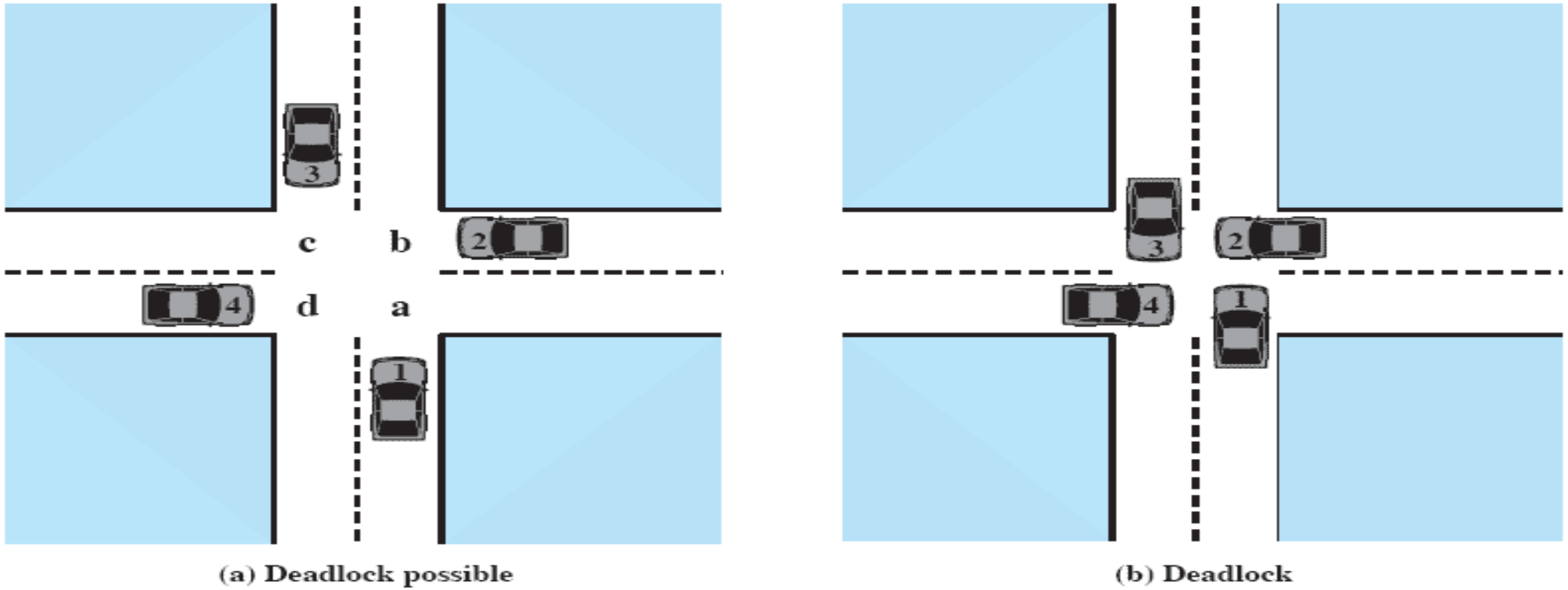
Dr. Prasanna Shete
Dept. of Computer Engineering
prasannashete@somaiya.edu
Mobile/WhatsApp 9960452937

Process Concurrency

- To understand the issues related to concurrent execution of processes and the solution/s -Principles of Concurrency

Deadlock

- 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
- No efficient solution
- Involve conflicting needs for resources by two or more processes



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

Traffic Deadlock

- Four quadrants of the intersection are the resources
- Traffic rules are that a vehicle at a 4 way intersection should defer to a vehicle immediately to its left (or right, abroad)
- Resource requirements for passing straight through the intersection?

Reusable Resources

- Safely used by only one process at a time and not depleted by that use
  - Processes obtain resources, use them and 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

Reusable Resources

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

Reusable Resources

- Space available for allocation is 200 Kbytes, 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
  - System design constraints may be imposed that take into account the order in which resources can be requested

Example of Deadlock

- Pair of processes; Each process attempts to receive message from other process
- Deadlock may occur if a Receiver is blocking
  - Cause: Design errors
    - does not occur always, May take a rare combination of events to cause deadlock

P1

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

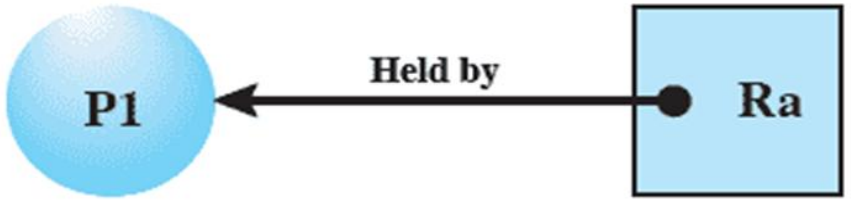
P2

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

Resource Allocation Graphs

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

(a) Resouce is requested



(b) Resource is held

Conditions for Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
  - No process is allowed to access a resource that has been allocated to another process
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of other resources
- No preemption
  - No resource can be forcibly removed form a process holding it

Conditions for Deadlock

- Deadlock can possibly occur if the first 3 conditions hold true
  - But might not always exist with just these 3 conditions
  - Fourth condition required →
- 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

Resource Allocation Graphs

Resource Allocation Graphs

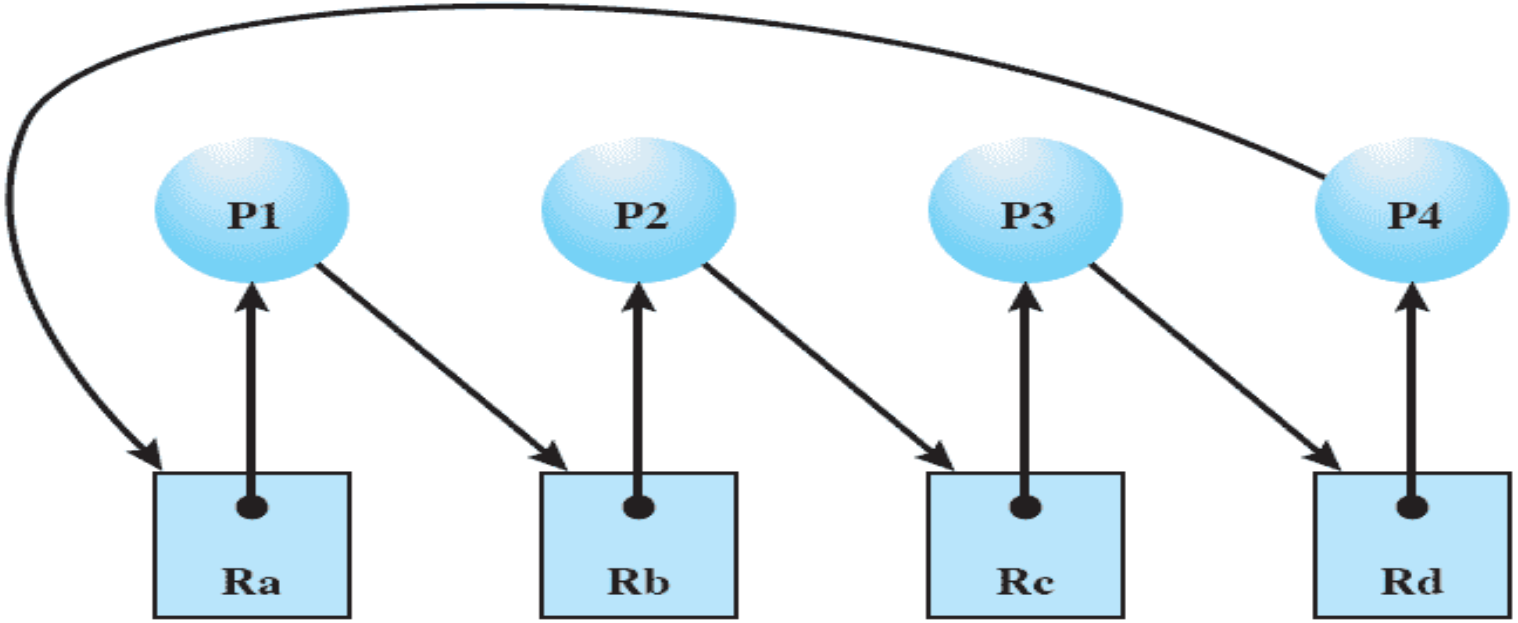


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Possibility of Deadlock

- Mutual Exclusion
- Hold and wait
- No preemption

Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

Deadlock Prevention

- 2 classes
- Indirect: prevent occurrence of one of the necessary condition
- Direct: prevent occurrence of circular wait

Deadlock Prevention

- Mutual Exclusion
  - Must be supported by the OS
- Hold and Wait
  - Can be prevented if a process requests all of its required resources at one time
- No Preemption
  - Can be prevented if a process holding some resource is denied access to its additional request
  - Process must release resource and request again
  - OS may preempt a process (low priority process holding a resource) to release its resources
- Circular Wait
  - Can be prevented by using a linear ordering of resource types
  - If a process has been allocated resources of type R, then it should subsequently request only those resources of type R, that follow the ordering

Deadlock Avoidance

- Allows 3 necessary conditions of deadlock, but assures that deadlock point is never met
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows more concurrency than prevention
- Requires knowledge of future process resource requests

Deadlock Avoidance: 2 approaches

- Do not start a process if its demands might lead to deadlock
- → Process Initiation Denial

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

Process Initiation Denial

- Consider a system of ‘n’ processes and ‘m’ different types of resources
- Following vectors and matrices are defined
  - Resource vector, $\mathbf{R_j = (R_1, R_2, \ldots, R_m)}$ - Total amount of each resource in the system
  - Available vector, $\mathbf{V_j = (V_1, V_2, \ldots, V_m)}$ - Total amount of each resource not allocated to any process
  - Claim Matrix, $\mathbf{[C_{ij}]}$ - Requirement of process *i* for resource *j* gives maximum requirements of each process for each resource
  - Allocation Matrix, $\mathbf{[A_{ij}]}$ - Current allocation of resource *j* to process *i* gives current allocation of each resource to each process
- Following relationship holds:

Process Initiation Denial

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$; for all j → All resources are either available or allocated
2. $C_{ij} \leq R_j$; for all j → No processes can claim more than the total amount of resources in the system
3. $A_{ij} \leq C_{ij}$; for all j → No processes is allocated more resources of any type than the process originally claimed

- Deadlock avoidance policy: Refuse to start a new process if its resource requirements may lead to deadlock
- Start a new process P_{n+1}, if and only if
  - $$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$$

A new process is only started if the max claim of current processes + those of a new process can be met

Resource Allocation Denial

- Referred to as Banker's algorithm
- State accounted in decision making
- *State* of the system reflects the current allocation of resources to processes
  - 2 vectors (resource vector, available vector)
  - 2 matrices (claim matrix, allocation matrix)
- ***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
- Safe state if → $C_{ij} - A_{ij} <= V_j$

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	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

	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

	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

	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

	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

Deadlock Avoidance Logic

```
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

Deadlock Avoidance Logic

```
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

- Deadlock Avoidance policies are conservative; deadlock is avoided by restricting processes to access the resources
- Deadlock detection→ no restriction; requested resources are granted whenever possible
- OS periodically detects the circular wait condition using an algorithm
- Algorithm marks the processes that are not deadlocked
  - Initially all processes are unmarked

Deadlock Detection Algorithm

- [A]- Allocation Matrix, (V) available vector and [Q]- request matrix
  - { Qij→ amount of resource of type '*j*' requested by process '*i*' }

1. Mark each process that has a row of all zeros in [A]
2. Initialize a temporary vector '*W*' that equals the Available vector *V* (W_k = V)
3. Find an index *i* such that process *i* is currently unmarked and i^{th} row of Q is less than or equal to W_k i.e $\boldsymbol{Q_{ik} <= W_k}$

   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 [A] to W.

   i.e. set $\boldsymbol{W_k = W_k + A_{ik}}$ ***(for 1<= k <= m);*** return to step 3

Deadlock exists iff, there are unmarked processes at the end of the algorithm→

Each unmarked process is deadlocked

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

Allocation 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

Advantages and Disadvantages

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 online handling	•Inherent preemption losses

Dining Philosophers Problem

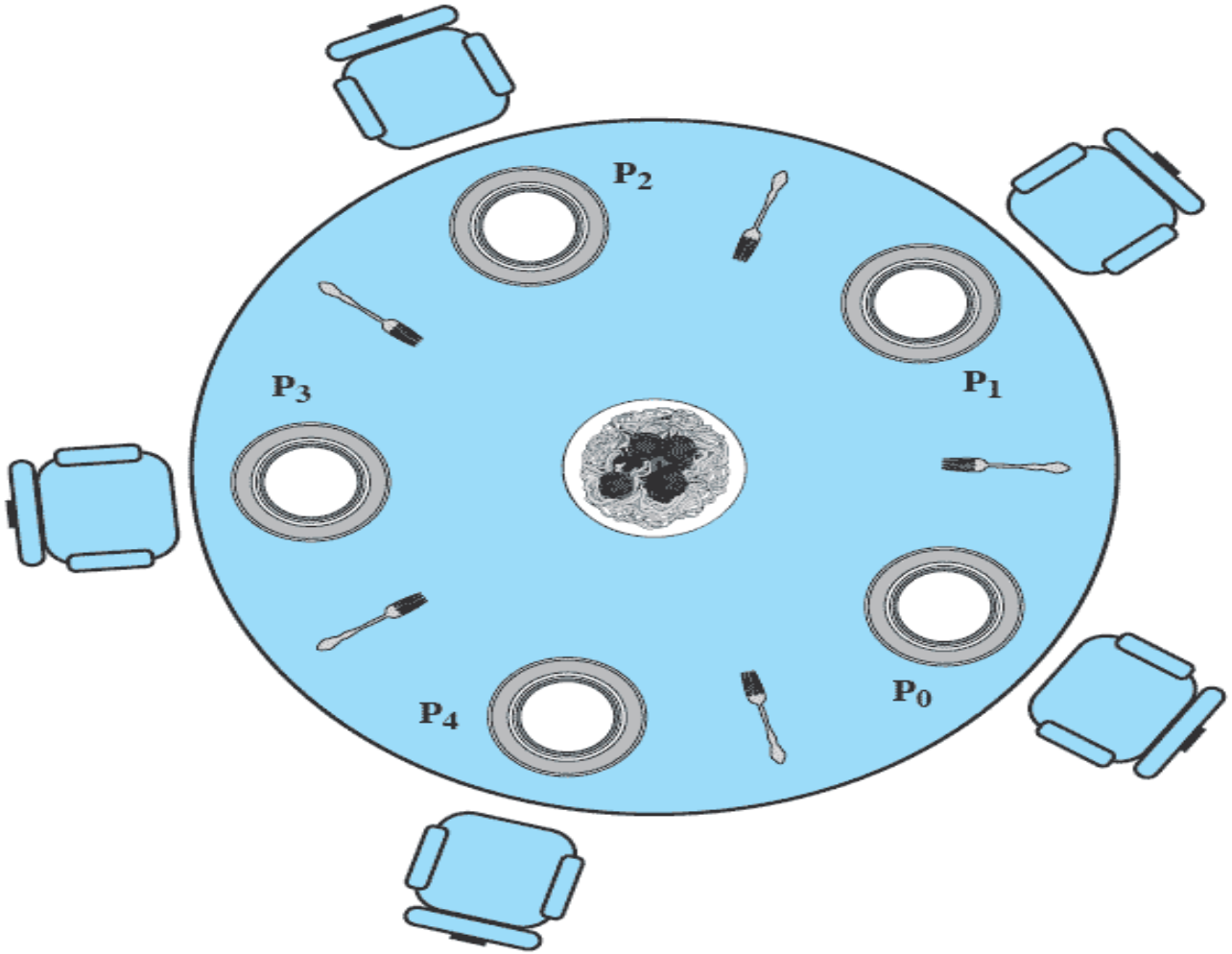


Figure 6.11 Dining Arrangement for Philosophers

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]);
}
```

Dining Philosophers Problem

```
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