Today's class

- Deadlock

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

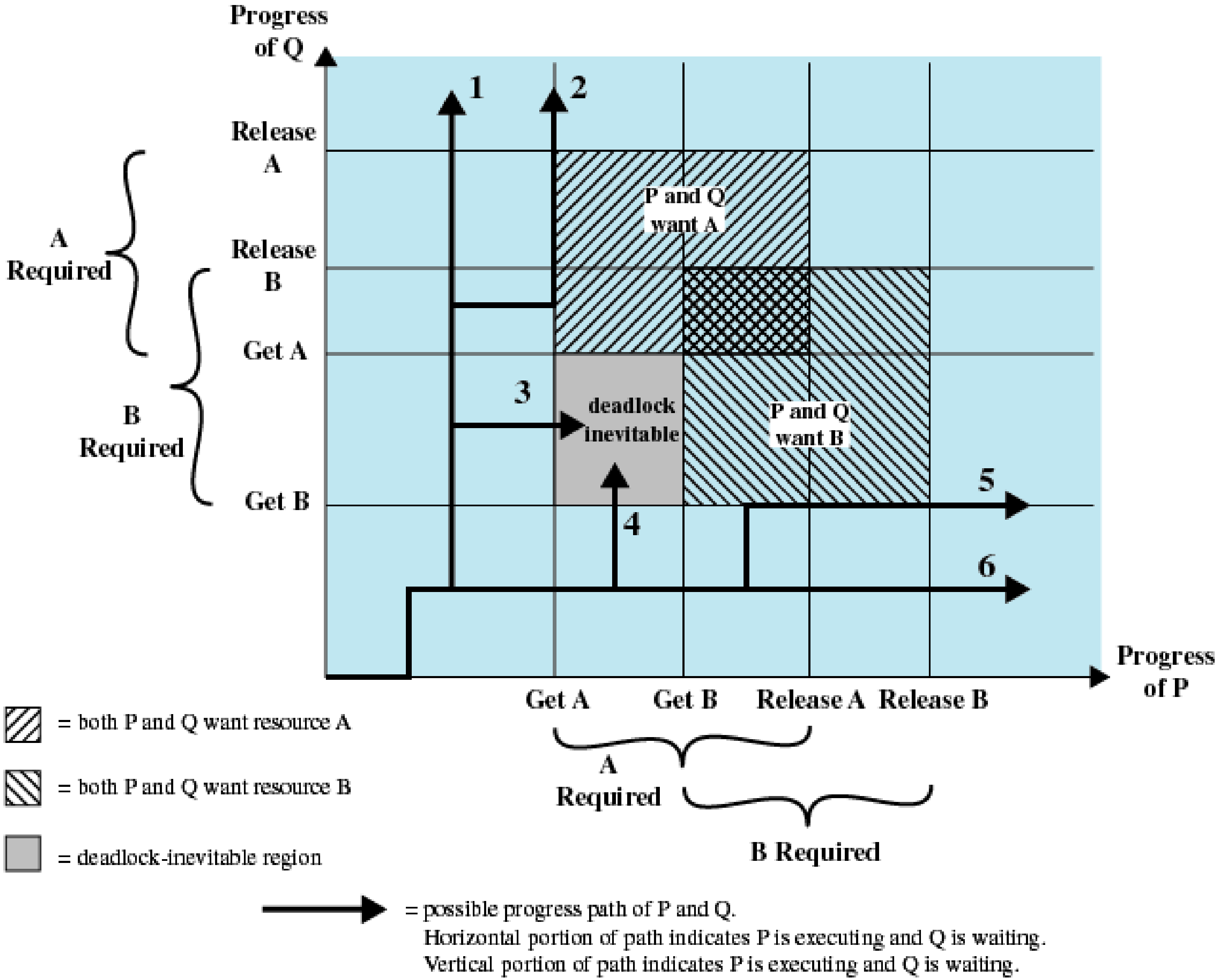


Figure 6.2 Example of Deadlock

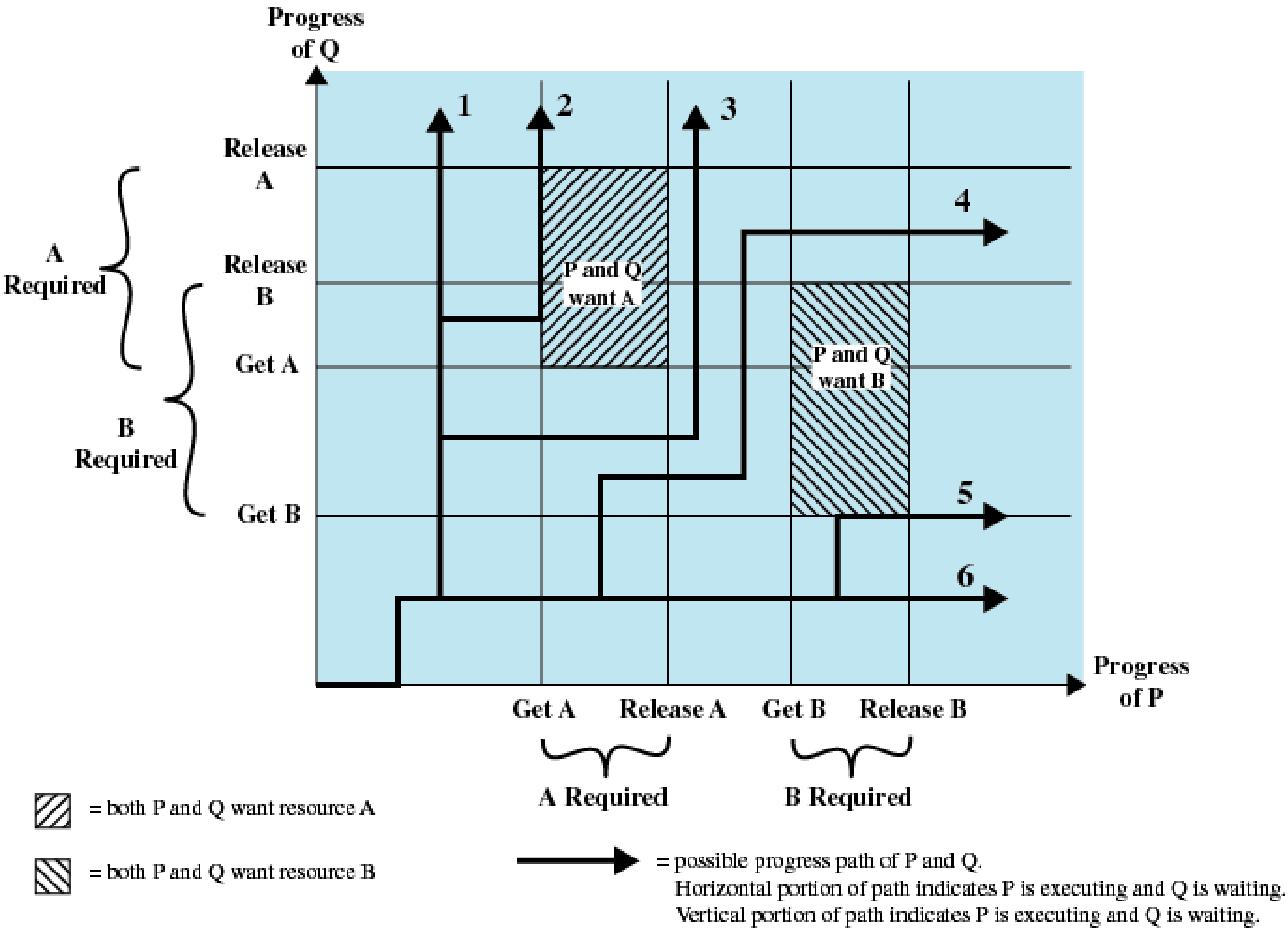


Figure 6.3 Example of No Deadlock [BACO03]

UPPSALA UNIVERSITET

Informationsteknologi

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
- 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

UPPSALA UNIVERSITET

Informationsteknologi

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

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

UPPSALA UNIVERSITET

Informationsteknologi

Example of Deadlock

- Deadlock occurs if Receive is blocking

P1

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

P2

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

UPPSALA UNIVERSITET

Informationsteknologi

Resource Allocation Graphs

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

(a) Resouce is requested

(b) Resource is held

UPPSALA
UNIVERSITET

Informationsteknologi

Resource Allocation Graphs

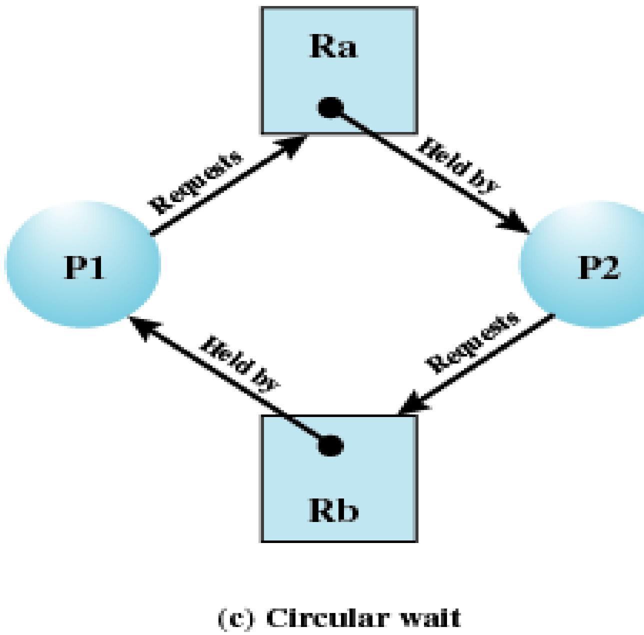


Figure 6.5 Examples of Resource Allocation Graphs

UPPSALA UNIVERSITET

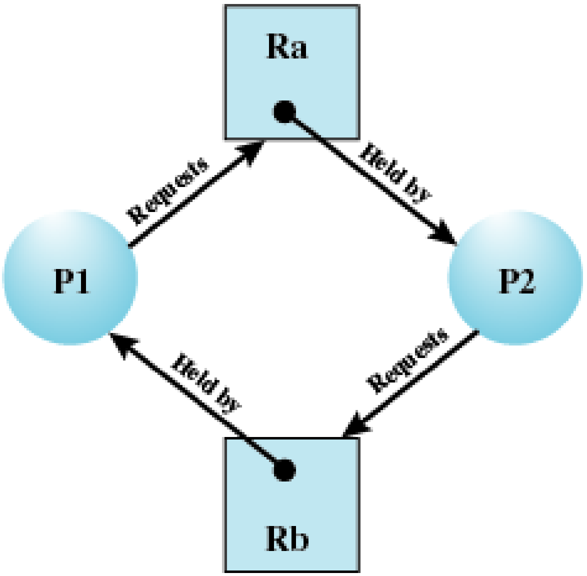
Informationsteknologi

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 form 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

UPPSALA UNIVERSITET

Informationsteknologi

Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

UPPSALA UNIVERSITET

Informationsteknologi

Existence of Deadlock

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

UPPSALA UNIVERSITET

Informationsteknologi

Deadlock Prevention

- Strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is excluded
- Indirect method – prevent the occurrence of one of the three necessary conditions mentioned earlier
- Direct method – prevent the occurrence of a circular wait

UPPSALA UNIVERSITET

Informationsteknologi

Deadlock Prevention

- Mutual Exclusion
  - Must be supported by the operating system, so it's hard to not allow this
- Hold and Wait
  - Require a process request all of its required resources at one time
  - Block the process until all requests can be granted simultaneously
  - Inefficient
    - May wait a long time for all resources to become available
    - Resources allocated to a process may be unused for a long period of time

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

Deadlock avoidance

- In deadlock prevention, resource requests are restrained to prevent one of the four conditions of deadlock
- This leads to inefficient use of resources and inefficient execution of processes
- Deadlock avoidance allows the three necessary conditions for deadlock but makes judicious choices to assure that the deadlock point is never reached

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

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 where there is at least one sequence of resource allocation to processes that does not result in deadlock
- Unsafe state is a state that is not safe

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

Determination of a Safe State P1 Runs to Completion

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Allocation matrix **A**

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

C – A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
7	2	3

(c) P1 runs to completion

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

Deadlock Detection

- Deadlock prevention strategies solve the problem of deadlock by limiting access to resources and imposing restrictions on processes. They are conservative in nature.
- Deadlock detection approaches grant resource requests whenever possible. Periodically, an algorithm that detects the circular wait condition is performed and recovery is attempted.

UPPSALA UNIVERSITET

Informationsteknologi

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

Available vector

Figure 6.10 Example for Deadlock Detection

UPPSALA UNIVERSITET

Informationsteknologi

Strategies once Deadlock Detected

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

UPPSALA UNIVERSITET

Informationsteknologi

Selection Criteria Deadlocked Processes

- 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

Dining Philosophers Problem

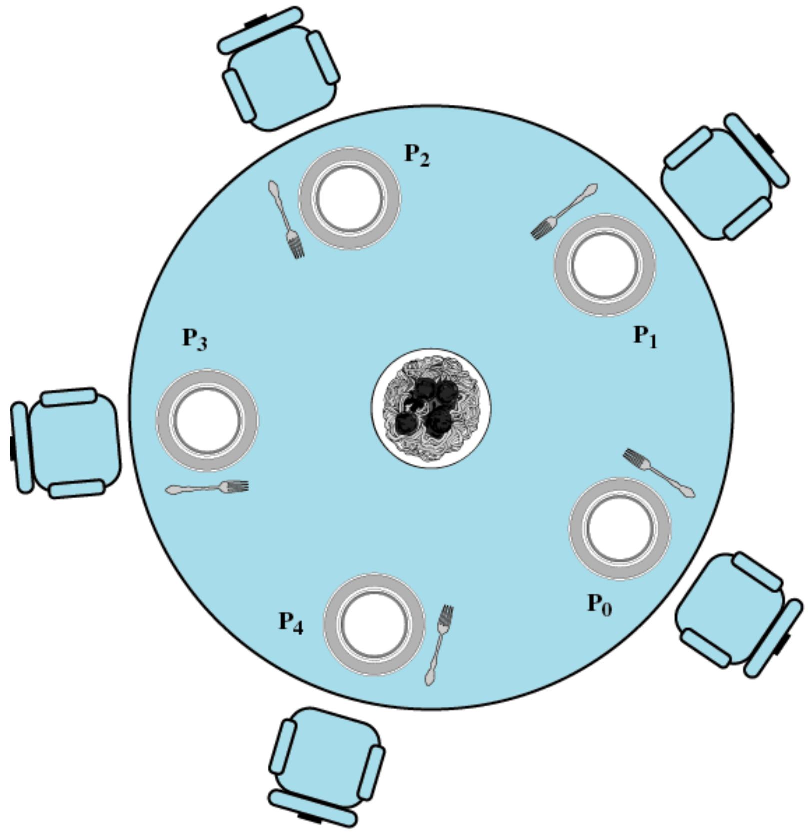


Figure 6.11 Dining Arrangement for Philosophers

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

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