

Chapter 6

Concurrency: Deadlock and Starvation

Deadlock

- Deadlock is the permanent blocking of a set of processes that either compete for resources or communicate with each other.
- Deadlocks involve conflicting needs for resources by two or more processes
- There is no efficient solution.

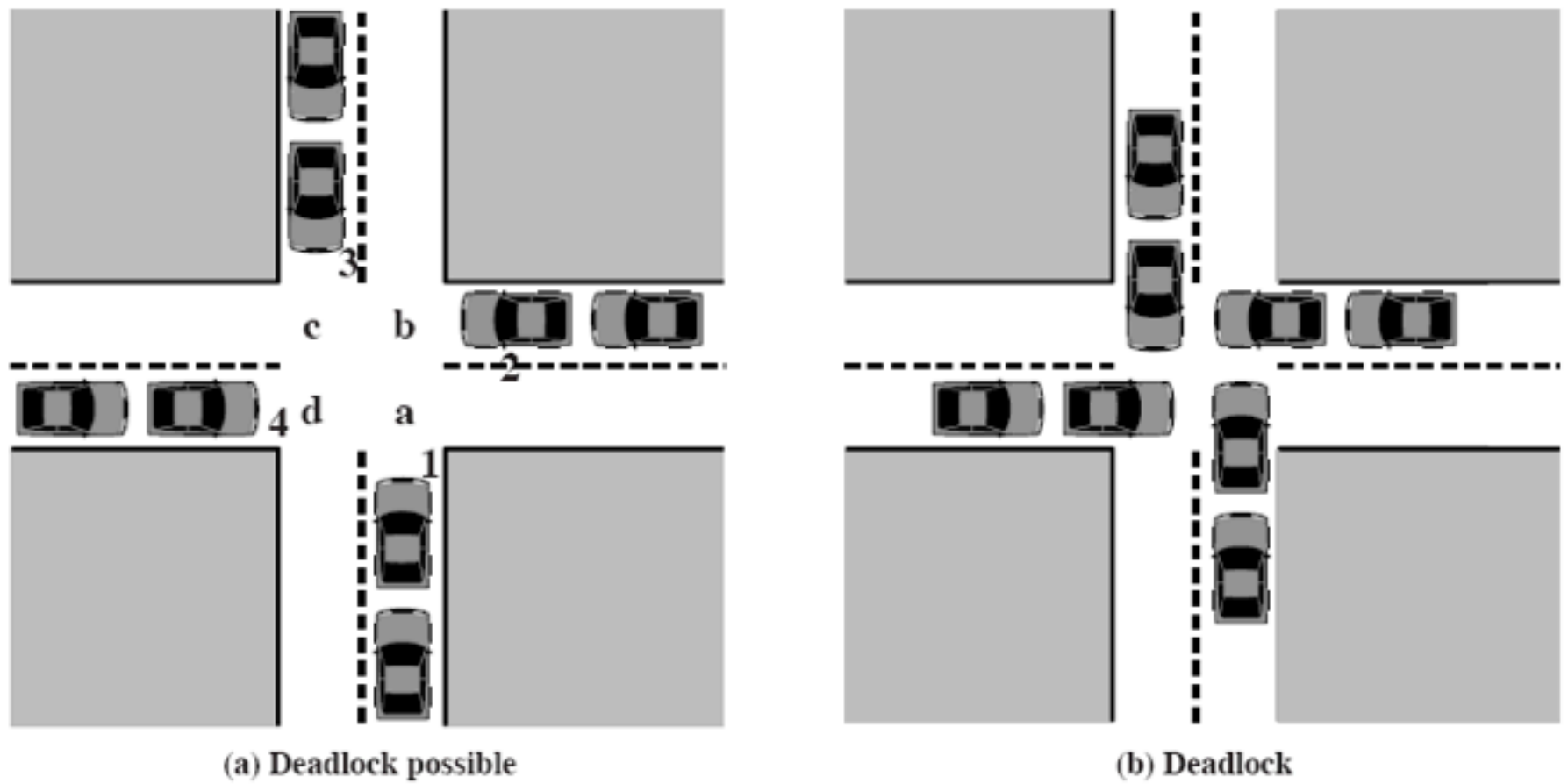


Figure 6.1 Illustration of Deadlock

Deadlock Example

Process P

Get A

Get B

Release A

Release B

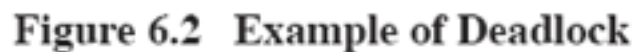
Process Q

Get B

Get A

Release B

Release A



No Deadlock Example

Process P

Get A

Release A

Get B

Release B

Process Q

Get B

Get A

Release B

Release A

Reusable Resources

Two categories of resources:

- Reusable
 - used by one process at a time and is not consumed.
 - For example, the processor
- Consumable
 - Can be created and destroyed.
 - For example, a message.

Deadlock of Reusable Resource

If only 200 Kbytes available, neither one can complete

Process P1

...

Request 80 Kbytes;

...

Request 60 Kbytes;

Process P2

...

Request 70 Kbytes;

...

Request 80 Kbytes;

Deadlock of Consumable Resource

P1

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

P2

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

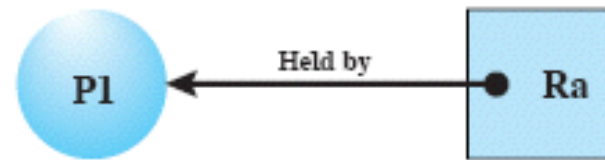
Deadlock occurs if receive calls are blocking calls, meaning it waits until it is finished.

Resource Allocation Graphs

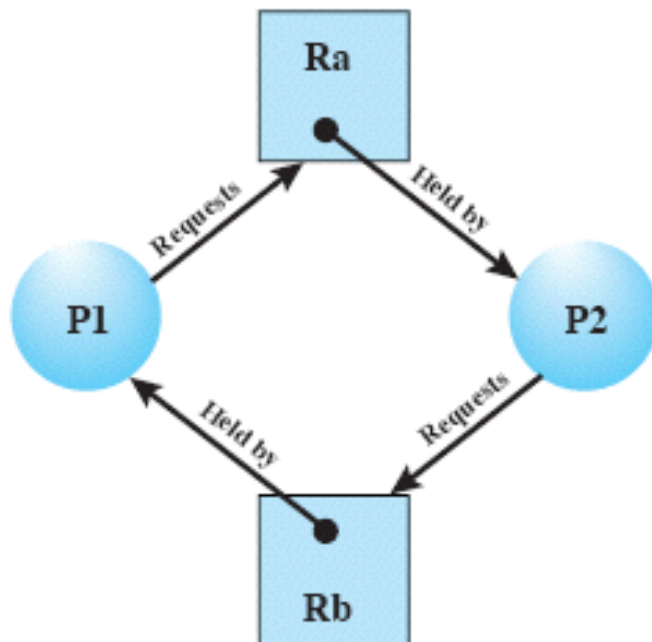
- A resource allocation graph is a directed graph that depicts a state of the system of resources and processes.
- Each resource and process is represented by a node.
- Dots within a resource node represent instances of that resource.
- An edge from a process to a resource represents a resource request.
- An edge from a dot within a resource node to a process indicates a resource has been assigned to the process.



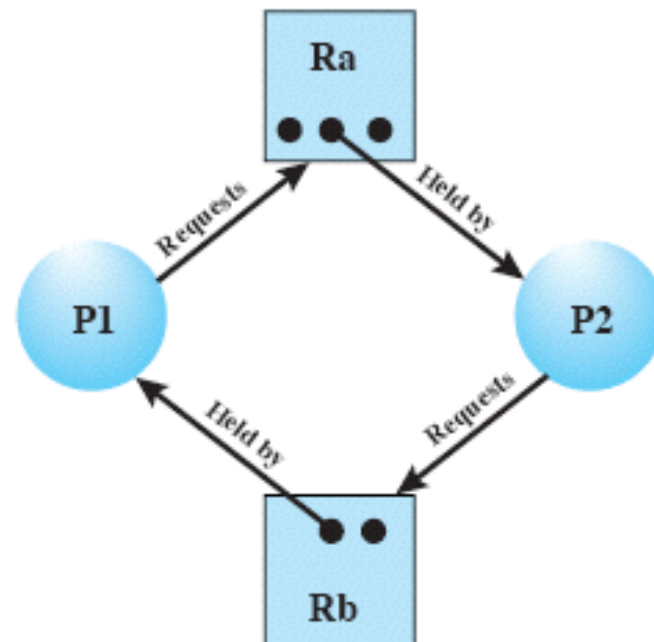
(a) Resource is requested



(b) Resource is held



(c) Circular wait



(d) No deadlock

Figure 6.5 Examples of Resource Allocation Graphs

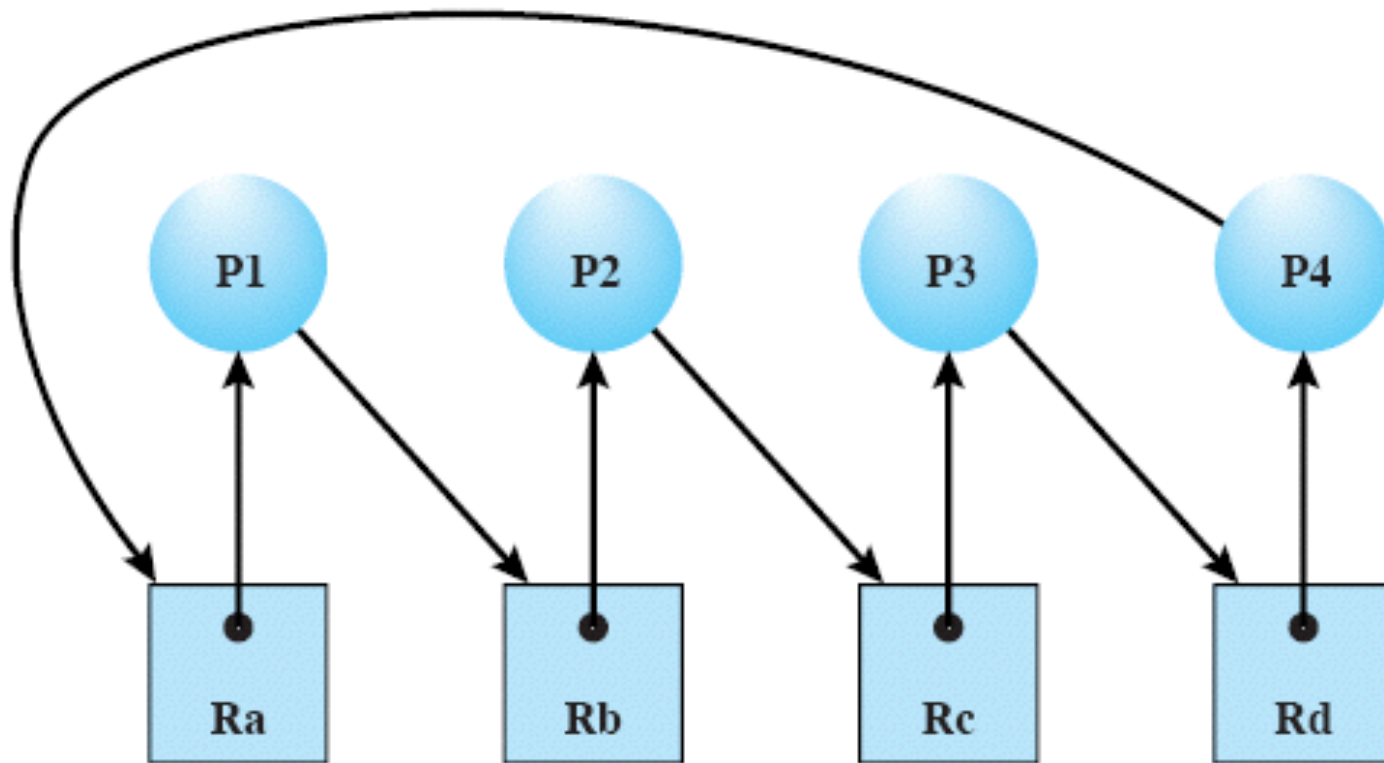


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Required Conditions for Deadlock

- Mutual exclusion:
 - only one process may use resource at a time
- Hold and wait:
 - a process may hold allocated resources while awaiting assignment of other resources.
- No preemption:
 - no resource can be forcibly removed from a process holding it.
- 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.

Deadlock Conditions

- The first three conditions are enough to allow deadlock to occur.
- The fourth condition depends on the existence of the first three conditions.
- The fourth condition is required for deadlock to actually occur. If it occurs, then a deadlock has occurred.

Dealing with Deadlocks

- There are three general approaches to dealing with deadlocks:
 - Prevention – prevent conditions necessary for deadlock.
 - Avoidance – allows first three conditions, avoids fourth condition dynamically.
 - Detection – detects if deadlock occurs, then deals with it.
- Note that there is no single effective strategy that can deal with all types of deadlock.

Deadlock Prevention

- Strategy is to prevent deadlock from occurring by denying the conditions:
- Two approaches:
 - Indirect: prevent one of the three preconditions.
 - Direct: prevent the fourth condition (circular wait).

Prevention: Mutual Exclusion

- Mutual Exclusion:
 - Hard to prevent since some resources require it.
 - For example, files should only be written by one writer at a time.

Prevention: Hold and Wait

- Hold and wait:
 - can be prevented by making process request all of its resources at one time and blocking it until all of the resources can be granted simultaneously.
- Drawbacks:
 - a process may have to wait a long time before all of its resources are free.
 - A process may acquire and keep resources a long time during which they are denied to other processes.
 - A process may not know in advance what resources it will need.

Prevention: No Preemption

Two possibilities:

- If a process holding resources cannot acquire a resource it needs, it must release the ones it has and request all of them again.
- If a process needs a resource held by another process, preempt the process and make it release its resources. This approach requires resources whose state can be easily saved and restored, such as the processor.

Prevention: Circular Wait

- Require an ordering of resource requests.
- Ordering prevents a circle by having all requests be forward in order, so that a process can't acquire a resource then request one prior to it which another process may be holding.
- As with preventing hold-and-wait, it may not be efficient to do this.

Deadlock Avoidance

- Similar to deadlock prevention.
- Allows the three preconditions but tries to prevent deadlock dynamically as resource requests are made.
- Allows more concurrency than prevention.
- Requires knowledge of future resource requests.

Deadlock Avoidance

Two approaches:

- **Process initiation denial**: constrains whether a new process is allowed to run.
- **Resource allocation denial**: lets a process run but constrains whether its resource requests are granted.

Process Initiation Denial

- Each process declares its maximum need for each resource.
- A process is only allowed to start if the maximum claim of all current processes plus the maximum claim of the new process can be met.
- This strategy doesn't work well because it assumes the worst case, that all processes will make their maximum resource requests at the same time, which is unlikely.
- So it may keep processes from starting that actually would have run successfully.

Resource Allocation Denial

- An algorithm for resource allocation denial is the “Banker’s Algorithm”, a term used by Dijkstra to compare processes and resources to bank reserves and loans.
- This algorithm defines the state of the system as either safe or unsafe.
- A safe state is one in which a sequence of allocations exists not leading to deadlock.
- An unsafe state is one in which there is no such sequence.
- The algorithm works by determining if the system would be in a safe state if the resource is granted. If so, grant the resource, if not, block the process until it is safe.

Banker's Algorithm

The banker's algorithm makes use of matrices and vectors:

- Claim matrix: requirement of each process for each resource.
- Allocation matrix: current allocation to each process of each resource.
- Resource vector: total amount of each resource in the system.
- Available vector: total amount of each resource not allocated to any process.

	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

Choose P2 which can acquire enough resources to finish.

	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

P2's resources now available.

	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

P1's resources now available.

(c) P1 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

P3's resources now available.

(d) P3 runs to completion

Now P4 can finish.

Figure 6.7 Determination of a Safe State

Choose P1 which can acquire enough resources to finish.

Choose P3 which can acquire enough resources to finish.

Safe State

- The previous slides represent a system that is in a “safe” state.
- This means that there is a sequence of resource allocation that does not result in deadlock.

Unsafe State

- An “unsafe” state is one that is not safe.
- That is, there is no sequence of resource allocation that will avoid deadlock.
- The next slides illustrate an example.

	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

- This is unsafe, because each process needs more of resource R1 and none are available.

Deadlock Avoidance

Advantages

- Less restrictive than deadlock prevention.
- Not necessary to preempt and rollback processes as in deadlock detection.

Disadvantages

- Maximum resource needs must be known in advance
- Process ordering needs to be independent (not constrained by any process synchronization).
- Must be a fixed number of resources.
- Processes can't exit while holding resources.

Deadlock Detection

- Deadlock prevention and avoidance strategies are conservative. They limit access to resources and impose restrictions on processes to solve the problem of deadlocks.
- At the opposite extreme is deadlock detection. It doesn't limit resource requests, which may lead to deadlock, but detects deadlocks and takes action if it occurs.

Deadlock Detection

- Periodically, the system performs an algorithm to detect deadlock.
- This can be done at each resource request, or less frequently due to the overhead involved.
- The algorithm uses an allocation matrix and available vector as in the banker's algorithm.
- A request matrix states the amount of resources being requested by each process.
- The algorithm is on the next slide...

Deadlock Detection Algorithm

1. Mark each process that has a row in the allocation matrix of all zeroes (holds no resources).
 2. For each remaining process see if its resource requests can be met by the current available resources.
 3. Mark all such rows and temporarily add their current allocation to the available list.
- If there are unmarked processes at the end of these steps, then these unmarked processes are deadlocked.

	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

P1 and P2 Deadlocked, because:

Mark P4 since no resources.

Mark P3 since its requests can be met by available resources.

Add P3's allocation temporarily to the available vector, giving us
0 0 0 1 1.

No other process requests can be met by this vector.

So, P1 and P2 remain unmarked, they are deadlocked.

Recovery

- **Abort** all deadlocked processes.
 - This is one of the **most common solutions**.
- Back up each deadlocked process to some previous **checkpoint** and restart all processes.
 - Could result in same deadlock again.
- Continue **aborting deadlocked** processes one by one until deadlock is resolved.
 - Abortion order should be intelligent.
- Successively **preempt resources** until deadlock is resolved.
 - Preemption choices should be intelligent. Requires rollback of process being preempted.

Recovery

Possible intelligent process choice criteria:

- Least CPU time so far.
- Least output so far.
- Most estimated remaining runtime.
- Least total resources allocated so far.
- Lowest priority.

Table 6.1 Summary of deadlock approaches

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommit resources	Requesting all resources at once	<ul style="list-style-type: none"> •Works well for processes that perform a single burst of activity. •No preemption necessary 	<ul style="list-style-type: none"> •Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none"> •Convenient when applied to resources whose state can be saved and restored easily 	<ul style="list-style-type: none"> •Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none"> •Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design 	<ul style="list-style-type: none"> •Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> •No preemption necessary 	<ul style="list-style-type: none"> •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	<ul style="list-style-type: none"> •Never delays process initiation •Facilitates online handling 	<ul style="list-style-type: none"> •Inherent preemption losses

Combined Approaches

- Another possibility is to use different approaches in different situations.
- For example, resources could be grouped into classes, with a linear ordering of requests required among classes.
- Within the class, the algorithm most appropriate for that class could be used.

Dining Philosophers Problem

- The Dining Philosophers problem is a classic concurrency problem.
- The problem involves five philosophers sitting down to eat spaghetti.
- The philosophers require two forks each to eat.
- A deadlock occurs if they all pickup their first fork before anyone can get a second fork.

Dining Philosophers Problem

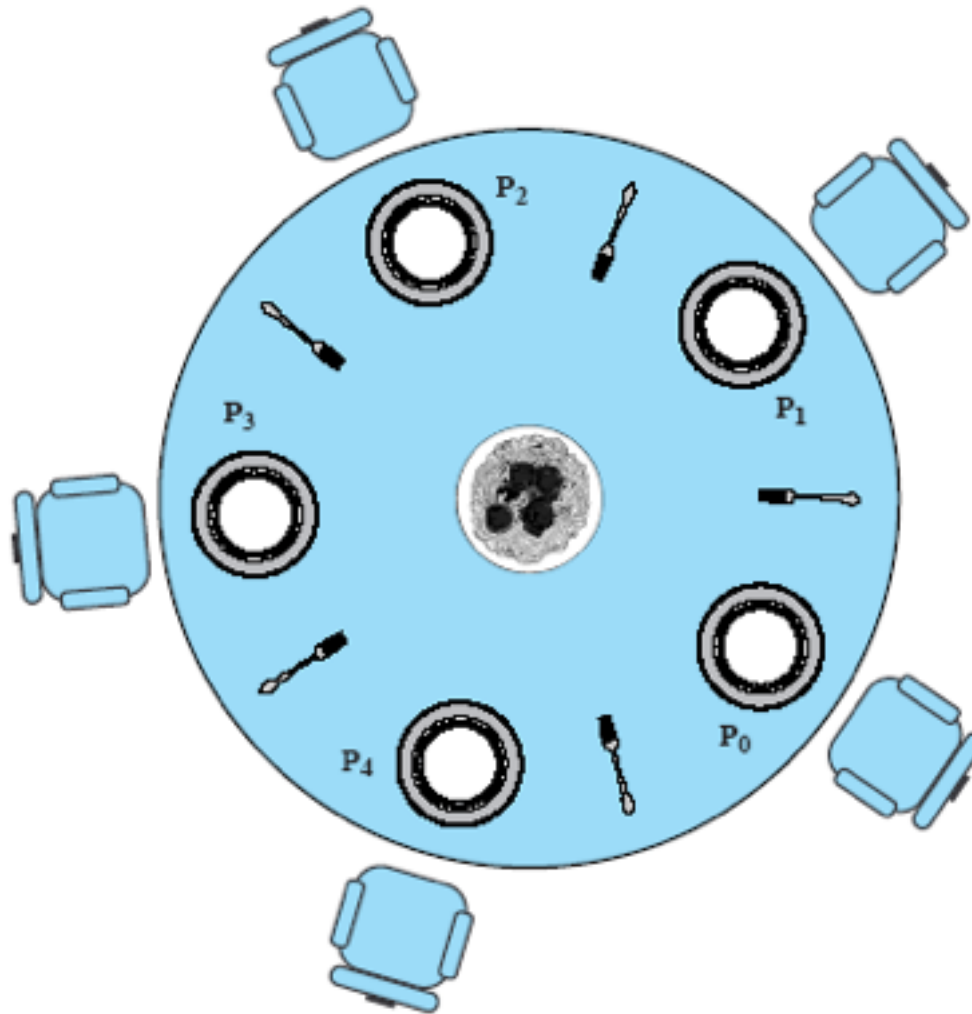


Figure 6.11 Dining Arrangement for Philosophers

```

/* program      diningphilosophers */
semaphore fork [5] = {1,1,1,1,1};
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));
}

```

Deadlock if
all processes
complete first
wait before
any completes
second wait.

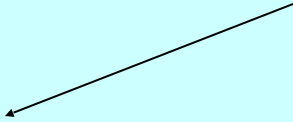


Figure 6.12 A First Solution to the Dining Philosophers Problem

```

/* program diningphilosophers */
semaphore fork[5] = {1,1,1,1,1};
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));
}

```

Now only 4 can enter at a time, so one fork is free so that one philosopher can have two forks.

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 */
    }
}
```

Doesn't deadlock since the first philosopher to get forks can get both forks before another philosopher can get forks.

UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

Pipes

- A pipe is a form of IPC where one process can send data to another.
- A process reading from a pipe blocks until data is present. Likewise a process writing is blocked until the pipe has room to write.
- The OS enforces mutual exclusion of pipe access, so only one process can access the pipe at a time.

Named versus Unnamed Pipes

- There are two kinds of Unix pipes, named and unnamed.
- Unnamed pipes are for related processes (parent / child).
- Named pipes are for unrelated processes.

Messages

- `msgsnd` and `msgrcv` are functions in Unix for sending and receiving messages.
- A message is a block of text with a type.
- Receiving processes can receive all messages or just a particular type.
- Senders and receivers block on full queues or empty queues.

Shared Memory

- This is the fastest form of IPC in Unix.
- Processes share a block of memory.
- Processes must enforce mutual exclusion themselves.

Semaphores

- The kernel supports the semaphore operations atomically.
- System calls are made to create and work with semaphores.

Signals

- For asynchronous events.
- A process can send a signal to another process, or the OS can send a signal to a process.
- A process may respond to a signal by executing a signal handler.

End of Slides