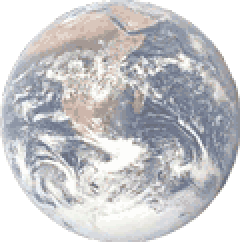


Concurrency:
Deadlock and Starvation

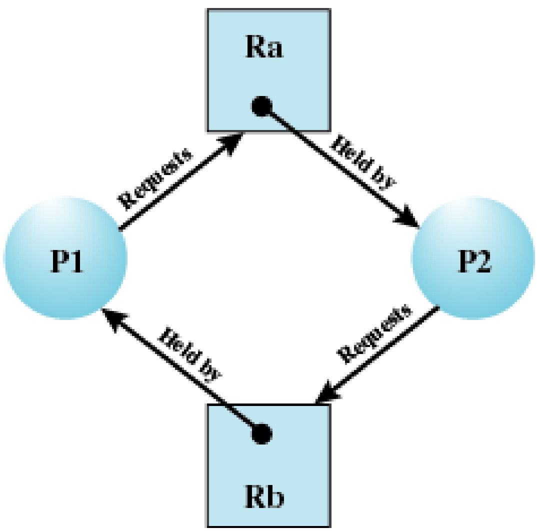
Agenda

- ✓ Principles of Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- An Integrated Deadlock Strategy
- Dining Philosophers Problem
- Linux Kernel Concurrency Mechanisms

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

- 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

Possibility of Deadlock	Existence of Deadlock
1. Mutual exclusion 2. No preemption 3. Hold and wait	1. Mutual exclusion 2. No preemption 3. Hold and wait 4. Circular wait

Three general approaches exist for dealing with deadlock.
First, one can **prevent deadlock by adopting a policy that eliminates one of the conditions** (conditions 1 through 4).

Second, one can **avoid deadlock by making the appropriate** dynamic choices based on the current state of resource allocation.

Third, one can attempt to **detect the presence of deadlock (conditions 1 through 4 hold) and take action** to recover.

Deadlock Prevention

- We can view deadlock prevention methods as falling into two classes.
- An **indirect** method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously (items 1 through 3).
- A **direct** method of deadlock prevention is to prevent the occurrence of a circular wait (item 4).

Deadlock Prevention

- Mutual Exclusion
  - Must be **supported** by the **operating system**
- Hold and Wait
  - Require a process request **all of its required** resources at one time

Deadlock Prevention

- No Preemption
  - Process <u>must release resource and request again</u>
  - Operating system <u>**may preempt** </u>a process to require it releases its resources
- Circular Wait
  - The circular-wait condition can be prevented <u>by</u>

  <u>Defineing a **linear ordering** of resource </u>types.

  (If a process has been allocated resources of type *R*, then it may subsequently request only those resources of types following *R* in the ordering.)

Deadlock Prevention

To see that this strategy works, let us associate an index with each resource type. Then resource R_i precedes R_j in the ordering if $i < j$. Now suppose that two processes, A and B, are deadlocked because A has acquired R_i and requested R_j, and B has acquired R_j and requested R_i. This condition is impossible because it implies $i < j$ and $j < i$.

As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

Deadlock Avoidance

- In **deadlock prevention , we constrain resource** requests to prevent at least one of the four conditions of deadlock.

- This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption),

  or directly, by preventing circular wait.

- This leads to <u>**inefficient**</u> use of resources and <u>inefficient execution</u> of processes.

Deadlock Avoidance

- **Deadlock avoidance , on the other hand, allows the three necessary conditions but**

  makes **judicious choices** to assure that the deadlock point is never reached.

- As such, **avoidance** allows more concurrency than **prevention**.
- With 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

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

Resource Allocation Denial

- The strategy of resource allocation denial, referred to as the **banker's algorithm**.

- Let us begin by defining the concepts of **state** and **safe state**.
- Consider a system with a **fixed** number of processes and a **fixed** number of resources.

- At any time a process may have zero or more resources allocated to it.

- The **state of the system** reflects the current allocation of resources to processes.

- **Thus,** the state consists of the two vectors, **Resource** and **Available**, and the two **matrices**, **Claim** and **Allocation**, defined earlier.

- A **safe state is one in which there is at least** one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion).

- An **unsafe state is, of course, a** state that is not safe.

Consider a system of **n** processes and **m** different types of resources.

Let us define the following vectors and matrices:

Resource $= \mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available $= \mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim $= \mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation $= \mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

The following relationships hold:

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 i,j	No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$, for all i,j	No process is allocated more resources of any type than the process originally claimed to need.

With these quantities defined, we can define a deadlock avoidance policy that **refuses to start a new process if its resource requirements might lead to deadlock**.

Start a new process P_{n+1} *only if*

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij} \quad \text{for all } j$$

- **The following example** illustrates these concepts. Next Figure-a shows the state of a system consisting of **four** processes **P** and **three** resources **R**.

- The total amount of resources **R1**, **R2**, and **R3** are **9**, **3**, and **6** units, respectively.

- **In the current state** allocations have been made to the four processes, **leaving**

1 unit of R2 and 1 unit of R3 available.

Is this a safe state?

Determination of a Safe State Initial State

Claim matrix **C**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix **A**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
0	1	1

(a) Initial state

<u>**To answer**</u> this question, we ask an intermediate question: Can any of the four processes be run to completion with the <u>resources available</u>?

That is, can the <u>**difference**</u> between the **maximum requirement** and **current allocation** for <u>**any**</u> process be met with the available resources?

$$C_{ij} - A_{ij} \leq V_j, \quad \text{for all } j$$

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

Determination of a Safe State P1 Runs to Completion

	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 P3 Runs to Completion

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
9	3	4

(d) P3 runs to completion

Determination of an Unsafe State

Consider the state defined in Figure a.

	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

Suppose that <u>**P1**</u> makes the request for one additional unit each of R1 and R3; if we assume that the request is granted, we are left in the state of Figure b.

	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

Is this a safe state?
The answer is no, because each process will need at least one additional unit of R1, and there are none available.

Thus, **<u>on the basis of deadlock avoidance</u>**, the request by P1 should be denied and P1 should be blocked

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)