Resource Sharing & Management

P.C.P Bhatt

Introduction

Some of the resources connected to a computer system (*image processing resource*) may be expensive.

- These resources may be *shared among users or processes.*

Dead-Lock Prevention

We have *multiple resources* and *processes* that can request *multiple copies of each resource.*

It is difficult modeling this as a *graph*.

We use the *matrix method* to model this scenario.

Assume n processes and m kinds of resources.

We denote the i^{th} resource with r_i.

We define 2 vectors each of size m,

Vector $R = (r_1, r_2, \ldots\ldots, r_m)$

Vector $A = (a_1, a_2, \ldots\ldots, a_m)$ where a_i is the resource of type i available for allocation.

We define 2 matrices for *allocations made* (AM) and the *requests pending for resources* (RM).

Matrix model of Requests and Allocation

RESOURCE VECTOR : $R = [r_1, r_2, \ldots\ldots r_m]$ and AVAILABILITY VECTOR $A = [a_1, a_2 \ldots a_m]$

$$
\begin{array}{cc}
 & \text{Resources} \longrightarrow \\
\begin{array}{c}\text{Processes} \\ \downarrow\end{array} &
\begin{bmatrix}
c_{11} & c_{12} & \cdots\cdots & c_{1m} \\
c_{21} & c_{22} & \cdots\cdots & c_{2m} \\
c_{n1} & c_{n2} & \cdots\cdots & c_{nm}
\end{bmatrix}
\end{array}
\qquad
\begin{array}{c}
\text{Resources} \longrightarrow \\
\begin{bmatrix}
q_{11} & q_{12} & \cdots\cdots & q_{1m} \\
q_{21} & q_{22} & \cdots\cdots & q_{2m} \\
q_{n1} & q_{n2} & \cdots\cdots & q_{nm}
\end{bmatrix}
\end{array}
$$

THE ALLOCATION MATRIX AM

THE REQUEST MATRIX RM

Clearly, we must have

$$\left[\sum_{i=1}^{n} \mathbf{c}_{i,j} + \mathbf{a}_j\right] <= \mathbf{r}_j$$

$$\left[\sum_{i=1}^{n} \mathbf{q}_{i,j}\right] >= \mathbf{r}_j$$

Bankers Algorithm

This is a *deadlock prevention* algorithm based on *resource denial* if there is a suspected risk of a deadlock.

A request of a process is assessed if the process resources can be met from the available resources RMi,j <= ai for all j.

Once the process is run, it shall return all the resources it held.

Note that Banker's Algorithm makes sure that *only processes that will run to completion are scheduled to run.*

However, if there are deadlocked processes, the will *remain deadlocked.*

Banker's Algorithm *does not eliminate a deadlock.*

Banker's Algorithm makes some unrealistic assumptions –*resource requirements for processes is known in advance.*

The algorithm requires that *there is no specific order* in which the processes should be run.

It assumes that there is a *fixed number* of resources available on the system.

A Graph Based Detection Algorithm

In the digraph model with one resource of one kind, we are required to detect a *directed cycle* in a processor *resource digraph.*

For each process, use the process node as root and traverse the digraph in *depth first mode* marking the nodes. *If a marked node is revisited, deadlock exists*.

Consider a process P_i and its corresponding row in matrix RM.

If vector $RM <= A$ then every resource request of process P_i can be met from the available set of resources.

On completion, this process can return its current allocation in row AM_i for another process.

Deadlock detection algorithm is in the following steps :

Step 0 : Assume that all processes are unmarked initially.

Step 1: While there are unmarked processes, choose an unmarked process with RMi <= A. process Step 2 else go to Step 3.

Step 2 : Add row AM_i to A and mark the process.

Step 3 : If there is no such process the algorithm terminates.

If all processes are marked, no deadlock.

If there is a set of processes that remain unmarked, then this set of processes have a deadlock.

Note that notwithstanding the non-deterministic nature of the algorithm it *always detects a deadlock.*

The method *detects a deadlock if present; it does not eliminate a deadlock.*

Deadlock elimination may require *preemption* or *release of resources.*

Mutual Exclusion Revisited : Critical Sections

Mutual exclusion is required for *memory.*

Mutual exclusion must be ensured whenever there is a *shared area of memory* and *processes writing to it.*

The main motivation is to avoid *race condition* among processes.

Critical Section is the section of code that is executed exclusively and *without any interruptions* – none of its operations can be *annulled.*

Unix provides a facility called *semaphore* to allow processes to use critical sections mutually exclusive of each other.

A semaphore is essentially a *variable which is treated in a special way*.

Access and operations on a semaphore is permitted only when it is in a *free state*.

If a process *locks* a semaphore, others cannot get access to it.

When a process enters a critical section, other processes are *prevented from accessing this shared variable*.

A process *frees* the semaphore on exiting the critical section.

To ensure this working, a notion of *atomicity* or *indivisibility* is invoked.

Basic Properties of Semaphores

- A semaphore takes only *integer values*.
- There are only *two operations* possible on a semaphore:

A *wait* operation on a semaphore decreases its value by 1.

wait(s) : while s < 0 do noop; s := s-1;

A *signal* operation increments its value

signal(s) : s := s + 1;

- A semaphore operation is *atomic.*

A process is *blocked* if its wait operation evaluates a *negative semaphore value*.

- A blocked process can be *unblocked* when some other process executes a *signal* operation.

Usage of Semaphore

Suppose two processes *P1* and *P2* use a semaphore variable *use* with initial value *0*.

We assume both processes have a program structure as:

```
repeat
   some process code here
   wait(use);
   enter the critical section the process
                    manipulates a shared area);
  signal(use);
  rest of the process code;
until false;
```

We have here an *infinite loop* for both processes.

Either P1 or P2 can be in its critical section.

The following is a representative operational sequence.

- Initially neither process is in critical section and

  use = 0.

- P1 arrives at critical section *first* and calls

  wait(use).

- It succeeds and enters the critical section setting

  use = -1.

- P2 wants to enter its critical section. Calls wait procedure.

- As *use* < *0*, *P2* busy waits.
- *P1* executes *signal* and exits its critical section, *use = 0* now.
- *P2* exits busy wait loop. It enters critical section *use = -1*.

The above sequence continues.

Semaphore is also used to *synchronize amongst processes.* A process may have a *synchronizing event.*

Suppose we have 2 processes P_i and P_j, P_j can execute some statement s_j only after statement s_i in P_i has been executed.

This can be achieved with semaphore s_e initialized to -*1* as follows:

- In P_i, execute sequence s_j ; *signal*(s_e);
- In P_j execute wait(s_e); s_j;

Now, P_j must wait completion of s_j before it can execute s_j.

These resources are not all used all the time.

In case of a *printer* - output resource is *used once in a while.*

This printer must be used amongst *multiple users* - because

the printer is expensive and because it is *sparingly used.*

Resources may be categorized depending upon the *nature of their use.*

OS needs a policy to schedule its use - dependant on *nature of use, frequency* and *context of use.*

For a printer, OS can *spool the data* to the printer the printer requests.

Each printer job must have *exclusive* use of it till it finishes.

Print-outs would be garbled otherwise.

Some times processes may require more than one resource.

A process *may not be able to proceed* till it gets all the resource.

Consider a process *P1* requiring resources *r1* and *r2*.

Consider process P2 requiring resources r2 and r3.

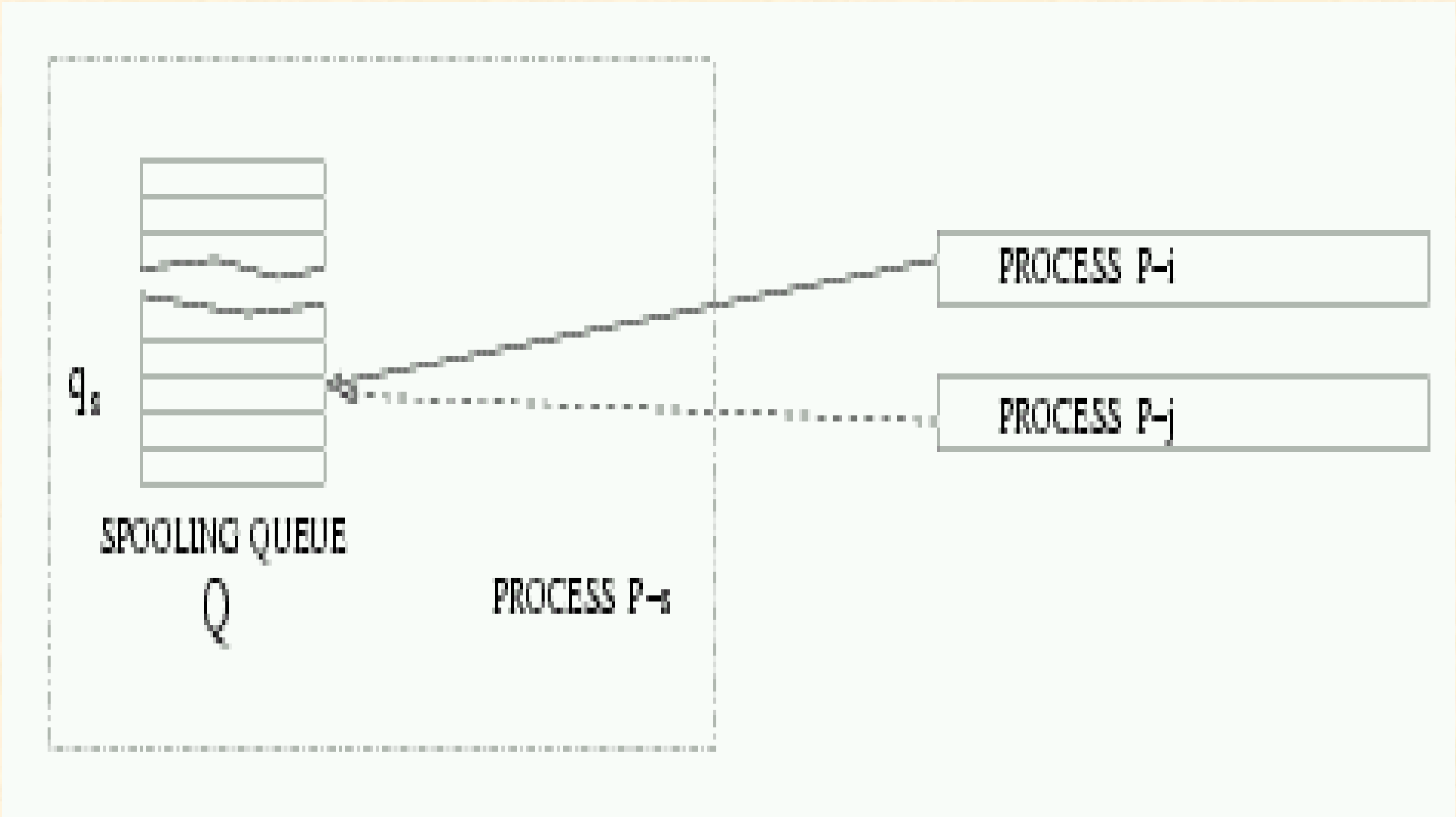
P1 will proceed only when it has both *r1* and *r2*. *P2* needs both *r2* and *r3*. If *P2* has *r2*, then *P1* has to wait until *P2* releases *r2* or terminates.

Mutual Exclusion

Mutual Exclusion is required in many situations in the OS design.

Consider the context of *management of a print request queue*.

Processes that need to print a file, deposit the *file address* into this queue. Printer spooler process picks the file address from this queue to print files.


PROCESS P-i
PROCESS P-j
SPOOLING QUEUE
Q
PROCESS P-s

Both processes P_i and P_j think their print jobs are spooled.

Q can be considered as a shared memory area between processes P_i, P_j and P_s.

Inter Process Communication can be established between processes that need printing and that which does printing.

Deadlocks

Consider an example in which process *P1* needs 3 resources *r1*, *r2* and *r3* to make any progress.

Similarly, *P2* needs resources *r2* and *r3*.

Suppose *P1* gets *r1* and *r3*; *P2* gets *r3*.

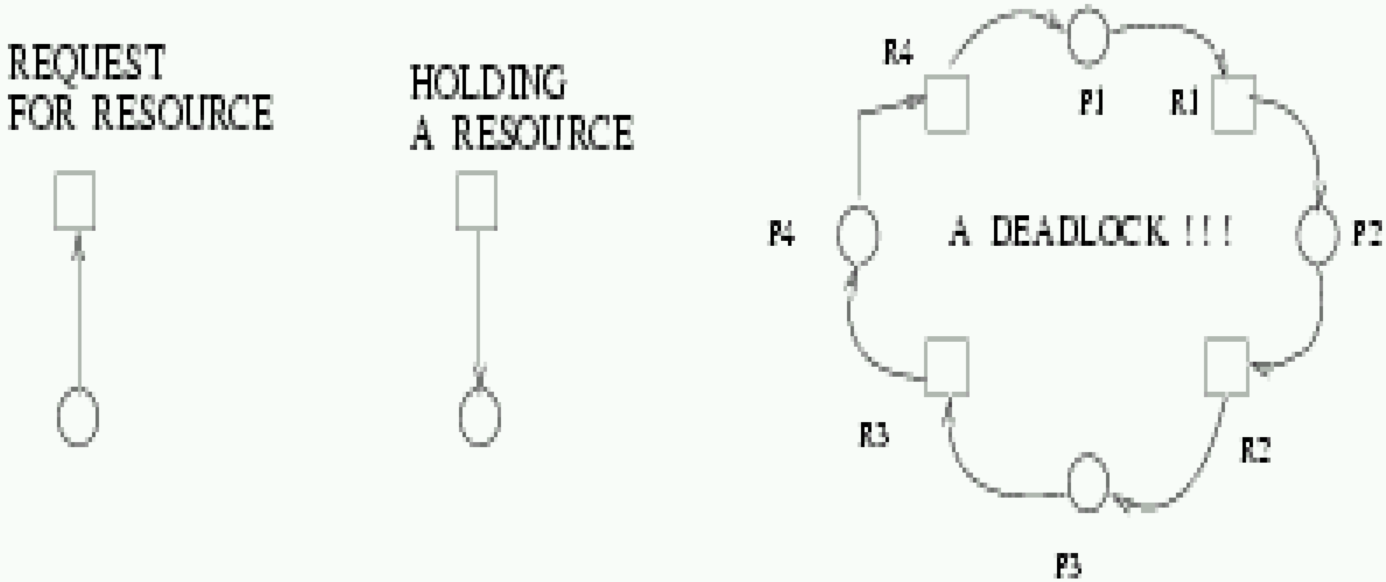
P2 is waiting for *r2* to be released; *P1* is waiting for *r3* to be released …… *deadlock.*

A *dead-lock* is a condition that may involve two or more processes in a state such that *each is waiting for release of a resource currently held by some other process*.

A PROCESS IS DENOTED BY A CIRCLE   A RESOURCE IS DENOTED BY A SQUARE

AN EDGE FROM A PROCESS TO A RESOURCE DENOTES A REQUEST FOR A RESOURCE

AN EDGE FROM A RESOURCE TO A PROCESS DENOTES THAT PROCESS HOLDS THE RESOURCE



Formally, a deadlock occurs when the following conditions are present simultaneously

- *Mutual Exclusion*
- *Hold and Wait*
- *No preemption*
- *Circular Wait*

Dead-lock Avoidance

Conditions for dead-lock to occur are *mutual exclusion, hold and wait, no preemption* and *circular wait*.

The first 3 conditions for dead-lock are *necessary conditions*. Circular Wait implies Hold and Wait.

How does one avoid having a dead-lock??

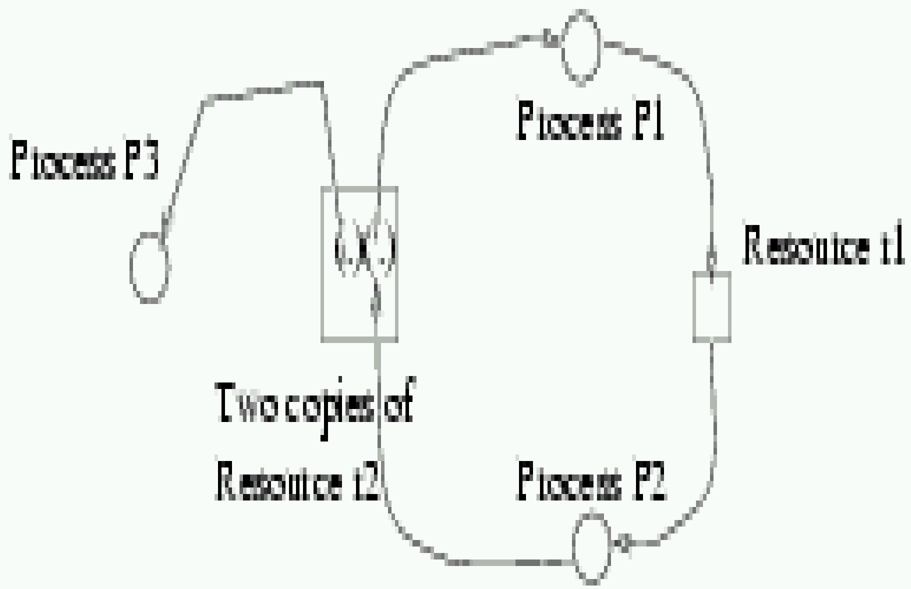
Infinite Resource Argument

One possibility is to have *multiple resources of the same kind.*

Sometimes, we may be able to break a dead-lock by having a *few additional copies of a resource.*

When one copy is taken, there is always another copy of that resource.

A PROCESS IS DENOTED BY A CIRCLE ◯ A RESOURCE IS DENOTED BY A SQUARE □



Process P1 has one r2 and requests r1

Process P2 has r1 and request r2

Process P3 has r2, which it will release on completion

The deadlock is broken when P3 terminates.

The pertinent question is,

how many copies of each resource do we need??

Unfortunately, theoretically, we need *infinite* number of resources!!!

In the example, if P3 is deadlocked, the deadlock between P1 and P3 cannot be broken.

Never let the conditions occur

It takes 4 conditions for dead-lock to occur.

This dead-lock avoidance simply states do not let conditions occur:

Mutual exclusion - unfortunately many resources require many exclusion!!

Hold and Wait - since this is implied by *Circular Wait*, we may possibly avoid Circular Wait.

Preemption - may not be the best policy to avoid dead-lock but works and is clearly enforceable in many situations.

Dead-Lock Prevention

We have *multiple resources* and *processes* that can request *multiple copies of each resource.*

It is difficult modeling this as a *graph.*

We use the *matrix method* to model this scenario.

Assume n processes and m kinds of resources.

We denote the i^{th} resource with r_i.

We define 2 vectors each of size m,

Vector $R = (r_1, r_2, \ldots\ldots, r_m)$

Vector $A = (a_1, a_2, \ldots\ldots, a_m)$ where a_i is the resource of type i available for allocation.

We define 2 matrices for *allocations made* (AM) and the the *requests pending for resources* (RM).

RESOURCE VECTOR : $R = [r_1, r_2, \ldots\ldots r_m]$ and AVAILABILITY VECTOR $A = [a_1, a_2 \ldots a_m]$

$$\begin{array}{cc} & \text{Resources} \longrightarrow \\ \text{Processes} \downarrow & \begin{bmatrix} c_{11} & c_{12} & \cdots\cdots & c_{1m} \\ c_{21} & c_{22} & \cdots\cdots & c_{2m} \\ c_{n1} & c_{n2} & \cdots\cdots & c_{nm} \end{bmatrix} \end{array} \qquad \begin{array}{c} \text{Resources} \longrightarrow \\ \begin{bmatrix} q_{11} & q_{12} & \cdots\cdots & q_{1m} \\ q_{21} & q_{22} & \cdots\cdots & q_{2m} \\ q_{n1} & q_{n2} & \cdots\cdots & q_{nm} \end{bmatrix} \end{array}$$

THE ALLOCATION MATRIX AM

THE REQUEST MATRIX RM

Bankers Algorithm

This is a *deadlock prevention* algorithm based on *resource denial* if there is a suspected risk of a deadlock.

A request of a process is assessed if the process resources can be met from the available resources RMi,j <= ai for all j.

Once the process is run, it shall return all the resources it held.

Note that Banker's Algorithm makes sure that *only processes that will run to completion are scheduled to run.*

However, if there are deadlocked processes, the will *remain deadlocked.*

Banker's Algorithm *does not eliminate a deadlock.*