Unit III (8 Hrs)

Concurrency Control : Concurrency and Race Conditions, Mutual exclusion requirements

Software and hardware solutions, Semaphores, Monitors, Classical IPC problems and solutions.

Deadlock : Characterization, Detection, Recovery, Avoidance and Prevention.

What is Concurrency?

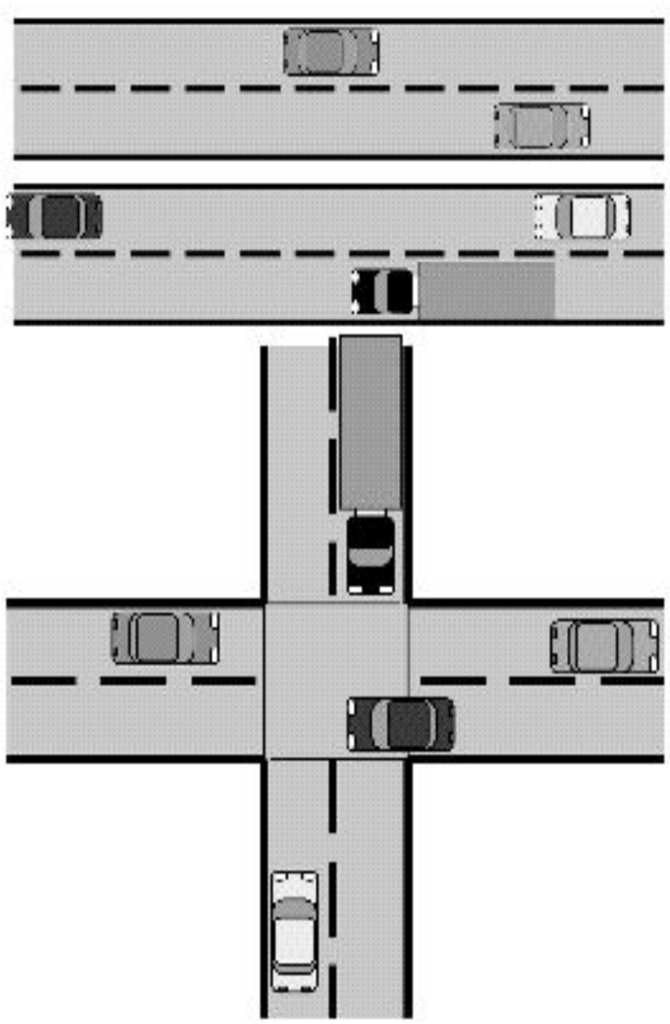
Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon.

In the real world, at any given time, many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency.

When dealing with concurrency issues in software systems, there are generally two aspects that are important: **being able to detect and respond to external events occurring in a random order**, and **ensuring that these events are responded to in some minimum required interval.**

If each concurrent activity evolved independently, in a truly parallel fashion, this would be relatively simple: we could simply create separate programs to deal with each activity.

The challenges of designing concurrent systems arise mostly because of the interactions which happen between concurrent activities. When concurrent activities interact, some sort of coordination is required.



e.g. - parallel activities that do not interact have simple concurrency issues. It is when parallel activities interact or share the same resources that concurrency issues become important.

Vehicular traffic provides a useful analogy. Parallel traffic streams on different roadways having little interaction cause few problems. Parallel streams in adjacent lanes require some coordination for safe interaction, but a much more severe type of interaction occurs at an intersection, where careful coordination is required.

Process management in operating systems can be classified broadly into three categories:

Multiprogramming involves *multiple processes* on *a system* with a *single processor*.

Multiprocessing involves *multiple processes* on *a system* with *multiple processors*.

Distributed processing involves multiple processes on *multiple systems*.

All of these involve cooperation, competition, and communication between processes that either run simultaneously or are interleaved in arbitrary ways to give the appearance of running simultaneously.

Concurrent processing is thus central to operating systems and their design.

Mutual Exclusion

Mutual exclusion is in many ways the fundamental issue in concurrency. It is the requirement that when a process P is accessing a shared resource R, no other process should be able to access R until P has finished with R. Examples of such resources include files, I/O devices such as printers, and shared data structures.

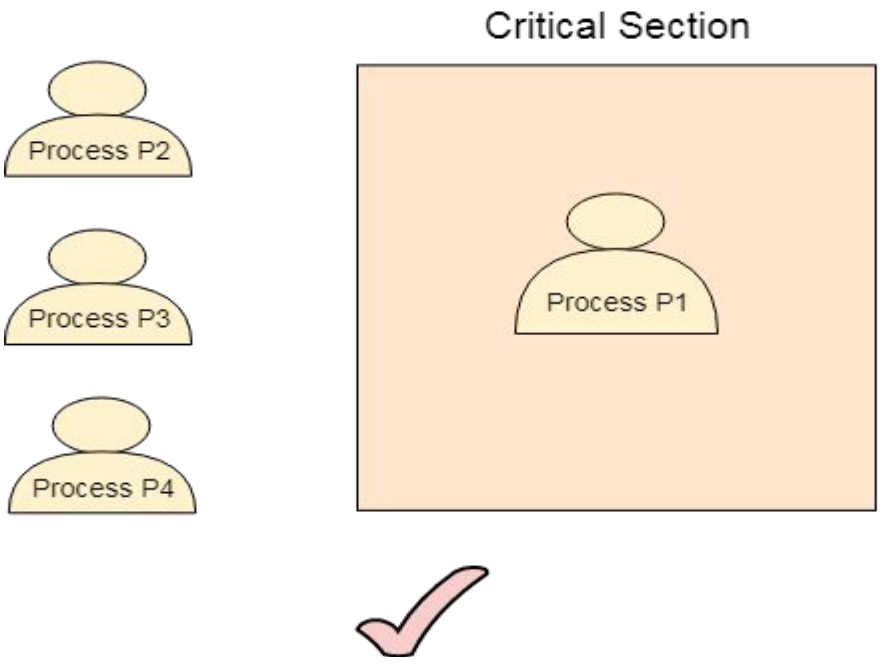
There are essentially three approaches to implementing mutual exclusion.

• Leave the responsibility with the processes themselves: this is the basis of most software approaches. These approaches are usually highly error-prone and carry high overheads.

• Allow access to shared resources only through special-purpose machine instructions:

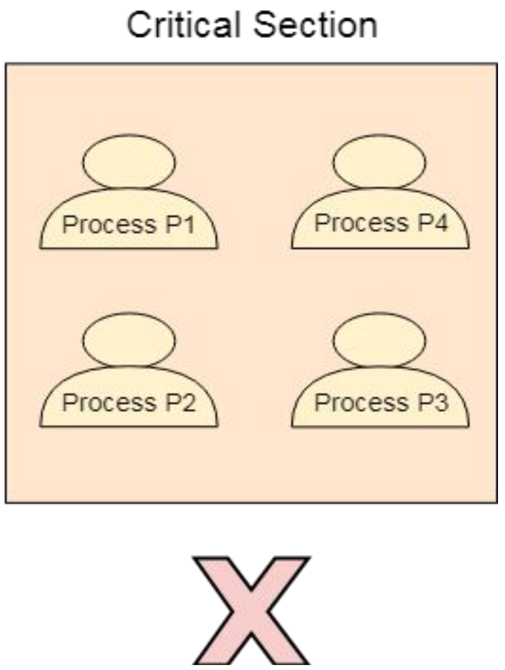
i.e. a hardware approach. These approaches are faster but still do not offer a complete solution to the problem, e.g. they cannot guarantee the absence of deadlock and starvation.

- Provide support through the operating system, or through the programming language. We shall outline three approaches in this category: semaphores, monitors, and message passing.



Means when turn = 1
loop
When turn = 0 then
P0 in CS

Means when turn = 0
loop
When turn = 1 then
P1 in CS

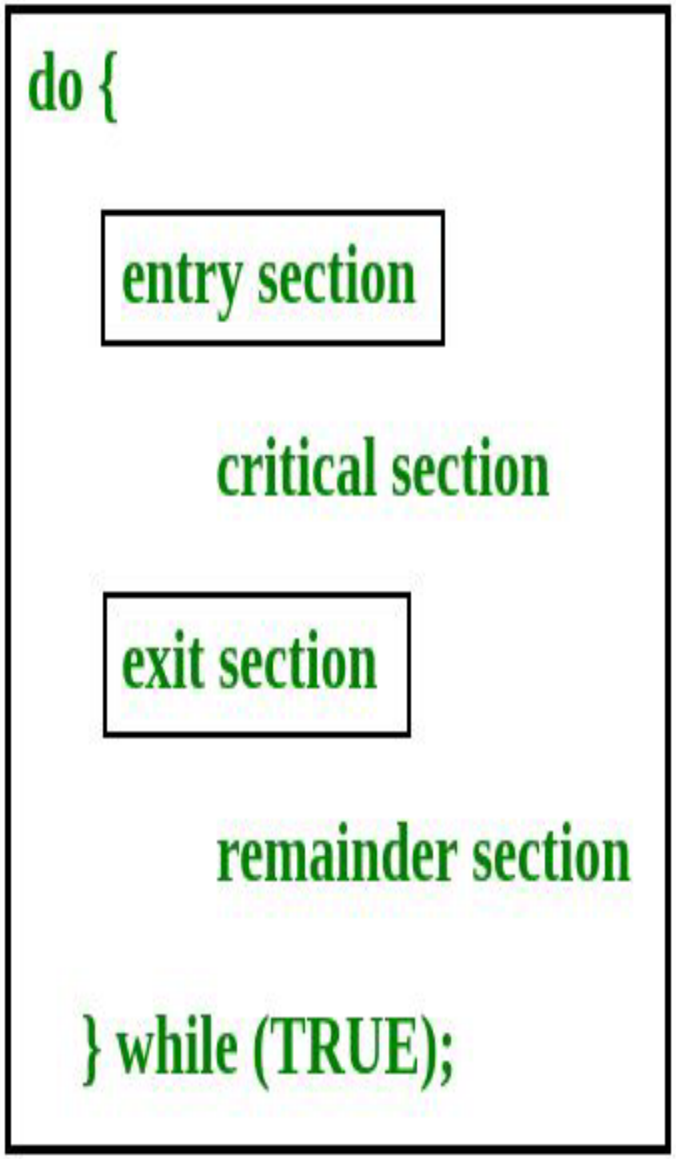


when turn = 0 (P0)   turn = 1 (P1)

Mutual Exclusion :- It implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

Progress :- Means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

Bounded Waiting :- It means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.



```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

P0

```
while (1)
{
    while(turn!=0);
    Critical Section
    turn = 1
    Reminder section
}
```

P1

```
while (1)
{
    while(turn!=1);
    Critical Section
    turn =0
    Reminder section
}
```

turn = 0	turn = 0
Or	**Or**
turn = 1	turn = 1

P0	**P1**
F	**F**

turn = 1	turn = 0

```
P0
while (1)
{
    flag[0]= T;
    while(flag[1]);
    Critical Section
    flag[0]=F;
    Reminder section
}
```

```
P1
while (1)
{
    flag[1]= T;
    while(flag[0]);
    Critical Section
    flag[1]=F;
    Reminder section
}
```

P0	**P1**
F	**F**
flag[0]	**flag[1]**

```
P0
while (1)
{
      flag[0]= T;
      turn = 1;
      while(turn==1 && flag[1]==T);
      Critical Section
      flag[0]=F;
      Reminder section
}
```

```
P1
while (1)
{
      flag[1]= T;
      turn=0;
      while(turn==0  &&  flag[0]==T);
      Critical Section
      flag[1]=F;
      Reminder section
}
```

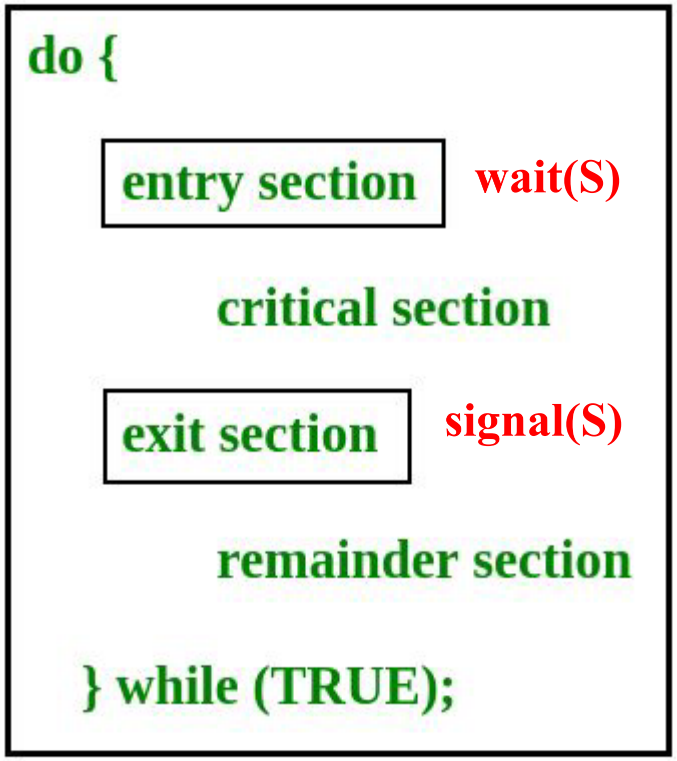
P0 **flag[0]**	**P1** **flag[1]**
T / F	**T / F**

Turn
F / T

Semaphore – Integer Variable (int S)

Which restrict access to shared resources in a multi processing environment. The two most common kinds of semaphore are counting semaphore and binary semaphore.

It is assessed through standard atomic operations - wait () & signal()



```
do {

    entry section    wait(S)

        critical section

    exit section    signal(S)

        remainder section

} while (TRUE);
```

```
wait(S)
{
     while(S<=0);
      S = S – 1 ;
}
```

```
signal(S)
{
     S = S + 1 ;
}
```

Producer & Consumer Problem

S = semaphore = 1 E = Empty Slots = n F = Full Slots = 0

1	2	3	4	5	6	7	8	9	10

Now

S = 1 E = 10 F = 00

Conditions :-

1) Consumer / producer should not work at a time.

2) Consumer should consume if any product is available.

3) Producer should produce if empty cell is there to store.

4) https://www.google.com/search?q=mutex+in+os&rlz=1C1FKPE_en-GB&tbm=vid&ei=dTiAZMWzAfWKseMPn_Kr2Aw&start=10&sa=N&ved=2ahUKEwiFwt7r17D_AhV1RWwGHR_5CssQ8NMDegQIGBAW&biw=1024&bih=657&dpr=1#fpstate=ive&vld=cid:3a2ccbb1,vid:8wcuLCvMmF8

1	2	3	4	5	6	7	8	9	10

```
void Producer()
{
    while(T)
    {
        produce();
        wait(E);
        wait(S);
        append();
        signal(S);
        signal(F);
    }
}
```

```
void Consumer()
{
    while(T)
    {
        wait(F);
        wait(S);
        take();
        signal(S);
        signal(E);
        Use();
    }
}
```

Reader Writer Problem

red = 0
wrt = 0
readcount=0

For Reader

```
wait (red)
read-count ++
if (readcount==1)
    wait (wrt)
signal red

    Read Operation

wait(red)
 read-count—
 if(readcount==0)
        signal (wrt)
 signal (red)
```

For Writer

```
wait (wrt)
        Write Operation
signal (wrt)
```

Reader – Reader Writer

red=1 wrt=1 wrt = 0

readcount=0

red=0 red=0

rc=1 rc=2

wrt=0

red=1 red=1

R1 R2

Exit Part

red=0

rc=1

red=1

Semaphores

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource.

Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding.

In multitasking operating systems, the activities are synchronized by using the semaphore techniques.

A semaphore is a variable. There are 2 types of semaphores:

1) Binary semaphores   2) Counting semaphores

Binary semaphores have 2 methods associated with it. (up, down / lock, unlock)

Binary semaphores can take only 2 values (0/1).

They are used to acquire locks.

When a resource is available, the process in charge set the semaphore to 1 else 0.

Counting Semaphore may have value to be greater than one, typically used to allocate resources from a pool of identical resources.

The fundamental idea of semaphores is that processes “communicate” via global counters that are initialized to a positive integer and that can be accessed only through two atomic operations (many different names are used for these operations: the following names are those used in Stallings Page 216):

semSignal(x) increments the value of the semaphore x.

semWait(x) tests the value of the semaphore x: if $x > 0$, the process decrements x and continues; if $x = 0$, the process is blocked until some other process performs a semSignal, then it proceeds as above.

A critical code section is then protected by bracketing it between these two operations:

```
semWait (x);        <critical code section>        semSignal (x);
```

In general the number of processes that can execute this critical section simultaneously is determined by the initial value given to x. If more than this number try to enter the critical section, the excess processes will be blocked until some processes exit. Most often, semaphores are initialized to one.

MUTEX

A mutex and the binary semaphore are essentially the same. Both can take values: 0 or 1. However, there is a significant difference between them that makes mutexes more efficient than binary semaphores.

A mutex can be unlocked only by the thread that locked it. Thus a mutex has an owner concept. Mutex is the short form for ‘Mutual Exclusion object’. A mutex allows multiple threads for sharing the same resource. The resource can be file. A mutex with a unique name is created at the time of starting a program. A mutex must be locked from other threads, when any thread that needs the resource. When the data is no longer used / needed, the mutex is set to unlock.

Monitors

The principal problem with semaphores is that calls to semaphore operations tend to be distributed across a program, and therefore these sorts of programs can be difficult to get correct, and very difficult indeed to prove correct!

Monitors address this problem by imposing a higher-level structure on accesses to semaphore variables. A monitor is essentially an object (in the Java sense) which has the semaphore variables as internal (private) data and the semaphore operations as (public) operations. Mutual exclusion is provided by allowing only one process to execute the monitor's code at any given time.

Monitors are significantly easier to validate than "bare" semaphores for at least two reasons:

- all synchronization code is confined to the monitor; and
- once the monitor is correct, any number of processes sharing the resource will operate correctly.

Concurrency issues (expressed using processes):

- **Atomic.** An operation is atomic if the steps are done as a unit. Operations that are not atomic, but interruptible and done by multiple processes can cause problems. For example, an lseek followed by a write is not atomic. A process is likely to lose its time quantum between the lseek (a slow operation if the distance seeked is large!) and the write. If another process has the file open and does a write then the result is not what is intended.
- **Race conditions.** A race condition occurs if the outcome depends on which of several processes gets to a point first. For example, fork( ) can generate a race condition if the result depends on whether the parent or the child process runs first. Other race conditions can occur if two processes are updating a global variable.

- **Blocking** and **starvation**. While neither of these problems is unique to concurrent processes, their effects must be carefully considered. Processes can *block* waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable. *Starvation* occurs when a process does not obtain sufficient CPU time to make meaniful progress.
- **Deadlock**. Deadlock occurs when two processes are blocked in such a way that neither can proceed. The typical occurrence is where two processes need two non-shareable resources to proceed but one process has acquired one resource and the other has acquired the other resource. Acquiring resources in a specific order can resolve some deadlocks.

Deadlock

Deadlock is defined as the permanent blocking of a set of processes that either compete for global resources or communicate with each other. It occurs when each process in the set is blocked awaiting an event that can be triggered only by another blocked process in the set.

Consider Figure 6.2 (all figures are taken from Stallings' web-site), in which both processes P and Q need both resources A and B simultaneously to be able to proceed. Thus P has the form get A, ... get B, ..., release A, ..., release B, and Q has the form get B, ... get A, ..., release B, ..., release A.

Banker's Algorithm – Deadlock Avoidance

Allocation Matrix	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Maximum Resource Types	A	B	C
	10	5	7

Currently Available	A	B	C
	3	3	2

MAX Required	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

1) Find the need

(Max- Allocation)

2) Find safe sequence

Need Matrix	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

P0

Need > Available

743 > 332 (True)

Don't Execute

P1

Need > Available

122 > 332 (False)

Executed

New available resources after P1

332+ 200 = 532

P2

Need > Available

600 > 532 (True)

Don't Execute

P3

Need > Available

011 > 532 (False)

Executed

New available resources after P3

532+ 211 = 743

P4

Need > Available

431 > 743 (False)

Executed

New available resources after P4

743+ 002 = 745

Again P0 (Second Try)

Need > Available

743 > 745 (False)

Executed

New available resources after P0

745+ 010 = 755

Again P2 (Second Try)

Need > Available

600 > 755 (False)

Executed

New available resources after P2

755+ 302 = 10 5 7

Maximum Resource Types	**A**	**B**	**C**
	10	5	7

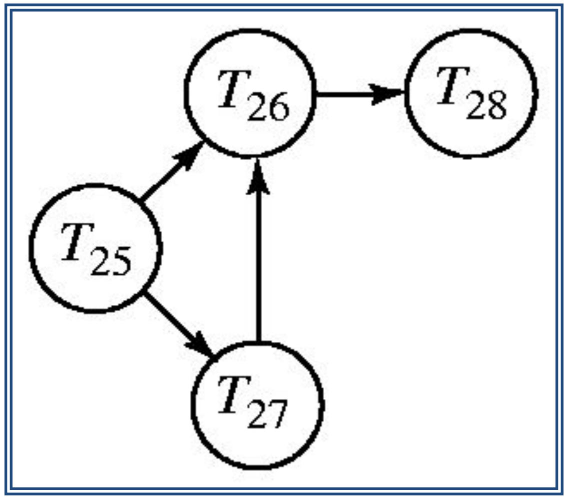
Lastly available resources should be match with Resources Given in example.

So safe sequence of processes to be adopted to avoid deadlock is

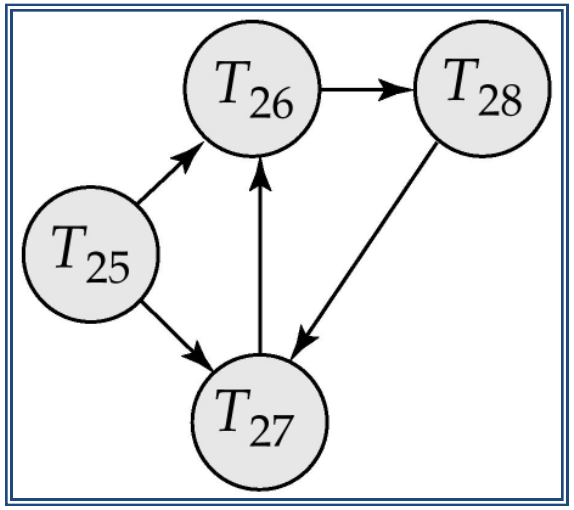
< P1 – P3 - P4 – P0 – P2 >

Deadlock Detection

- Deadlocks can be described as a *wait-for graph* where:
  - vertices are all the transactions in the system
  - There is an edge $T_i \rightarrow T_k$ in case T_i is waiting for T_k
- When T_i requests a data item currently being held by T_k, then the edge $T_i \rightarrow T_k$ is inserted in the wait-for graph. This edge is removed only when T_k is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

The six paths depicted are as following.

1. Q acquires both resources, then releases them. P can operate freely later.
2. Q acquires both resources, then P requests A. P is blocked until the resources are
   released, but can then operate freely.
3. Q acquires B, then P acquires A, then each requests the other resource.
   Deadlock is now inevitable. **(expected)**
4. P acquires A, then Q acquires B, then each requests the other resource.
   Deadlock is now inevitable.
5. P acquires both resources, then Q requests B. Q is blocked until the resources
   are released, but can then operate freely.
6. P acquires both resources, then releases them. Q can operate freely later.

The differences between binary semaphore and mutex are:

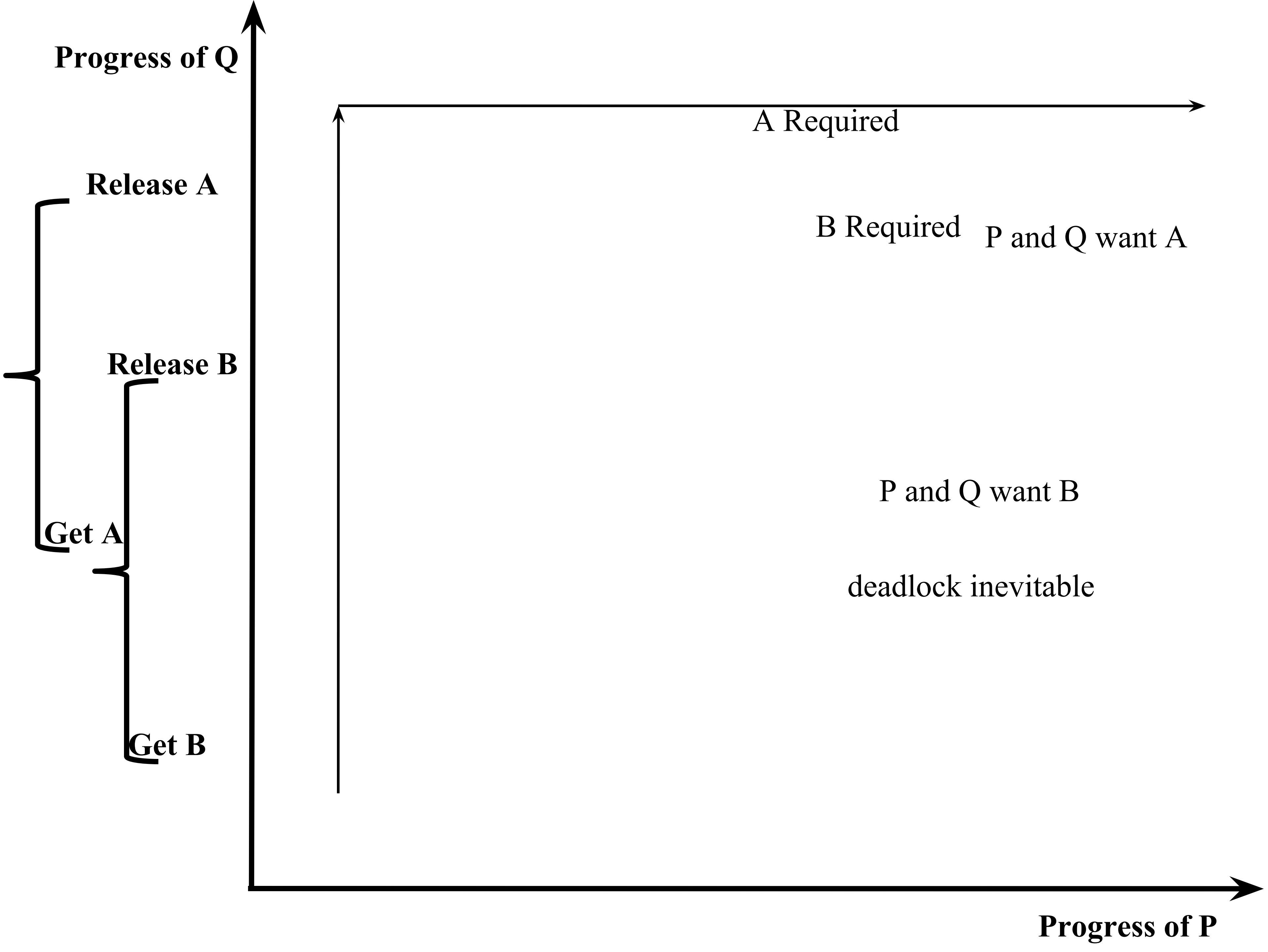
Mutex is used exclusively for mutual exclusion. Both mutual exclusion and synchronization can be used by binary.

A task that took mutex can only give mutex.

From an ISR a mutex can not be given.

Recursive taking of mutual exclusion semaphores is possible. This means that a task that holds before finally releasing a semaphore, can take the semaphore more than once.

Options for making the task which takes as DELETE_SAFE are provided by Mutex, which means the task deletion is not possible when holding the mutex.


Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
P and Q want A
P and Q want B
deadlock inevitable
Progress of P