SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited with Grade "A" by NAAC | Approved by AICTE

SCS1301 - OPERATING SYSTEM

UNIT – III

SYNCHRONIZATION AND DEADLOCKS

Dr.S.PRAYLA SHYRY
Assoc. Professor
Dept. of Computer Science and Engineering
School of Computing

UNIT 3 SYNCHRONIZATION AND DEADLOCKS 9 Hrs

The critical section problem - Semaphores - Classic problems of synchronization - Critical regions - Monitors Deadlocks - Deadlock characterization - Prevention - Avoidance - Detection - Recovery.

UNIT 4 MEMORY MANAGEMENT 9 Hrs

Storage Management Strategies - Contiguous Vs. Non-Contiguous Storage Allocation - Fixed & Variable Partition Multiprogramming - Paging - Segmentation - Paging/Segmentation Systems - Page Replacement Strategies - Demand & Anticipatory Paging - File Concept - Access Methods - Directory Structure - File Sharing - Protection File - System Structure - Implementation.

COOPERATING PROCESSES

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.
- There are several reasons for providing an environment that allows process cooperation:
  - **INFORMATION SHARING**
  - **COMPUTATION SPEEDUP**
  - **MODULARITY**
  - **CONVENIENCE**
  - **BUFFER**

COOPERATING PROCESSES

- **INFORMATION SHARING:**
  - Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **COMPUTATION SPEEDUP:**
  - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
- **MODULARITY:**
  - We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads
- **CONVENIENCE**
  - Even an individual user may work on many tasks at the same time

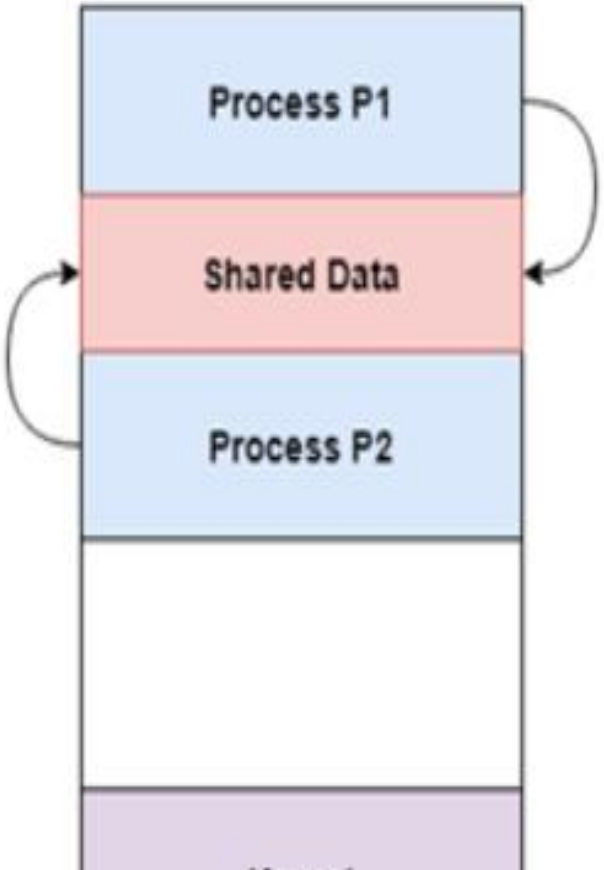
COOPERATING PROCESSES

- **A cooperating process is one that can affect or be affected by other process executing in the system**

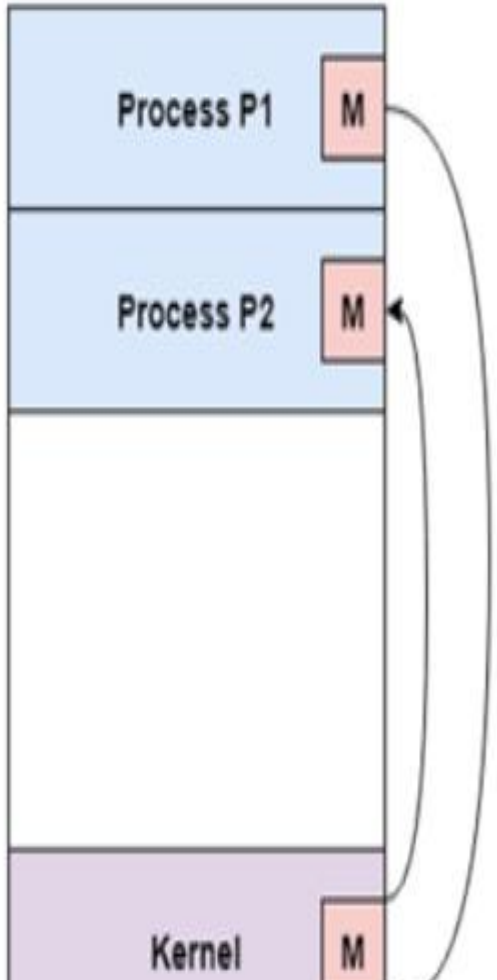
Cooperating process

- **Directly share a logical address data space (i.e. code & data)** - This may result in data inconsistency. It is implemented on threads.
- **Share data only through files/ messages** - So we will deal with various to order....orderly execution of cooperating process so that data consistency is maintained.

Cooperation by Sharing



Cooperation by Communication



Example- producer-consumer problem

- Also Known as bounded buffer problem.

(a print program produces characters that are consumed by the printer driver)

- Problem describes two processes

1. Producer

2. Consumer

They share a common fixed Size buffer

Producer: Generate a piece of data, put it into the buffer and start again

Consumer: Consuming the data(removing the data from the buffer)one piece at a time.

Problem?

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution?

Solution from producer side

- Producer goes to sleep mode or discard data if the **buffer is full**
- Consumer removes the data from the buffer and it notifies the producer

Solution from Consumer side

- Consumer goes to sleep mode if the **buffer is found to be empty**
- The very next time when the producer puts data in the buffer ,it wakes the consumer

Producer Side

```
BufferSize = 3;
 count = 0;

Producer()
 {
int item;
WHILE (true)
{
make_new(item);                    // create a new item to put in the buffer
IF(count==BufferSize)
Sleep();                           // if the buffer is full, sleep
put_item(item);                    // put the item in the buffer
count = count + 1;                 // increment count of items
IF (count==1)
Wakeup(Consumer);                  // if the buffer was previously empty, wake the consumer
}
}
```

Consumer Side

```
 Consumer()
{
Int item;
WHILE(true)
{
IF(count==0)
Sleep();                          // if the buffer is empty, sleep
remove_item(item);                // take an item from the buffer
count = count - 1;                // decrement count of items
IF(count==N-1)
Wakeup(Producer);                 // if buffer was previously full, wake the producer
Consume_item(item);               // consume the item
}
}
```

Race condition ….(leads to deadlock)

The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if-block.

Just before calling sleep, the consumer is interrupted and the producer is resumed.

The producer creates an item, puts it into the buffer, and increases itemCount.

Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.

Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when itemCount is equal to 1.

The producer will loop until the buffer is full, after which it will also go to sleep.

Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

Race condition

- When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called race condition

- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter.

- Hence processes must be synchronized.

Critical Section Problem

The Critical-Section

A code segment that accesses shared variables (or other shared resources) and that has to be executed as an atomic action is referred to as a critical section.

- *n* processes competing to use some shared data.
- No assumptions may be made about speeds or the number of CPUs.
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.

Critical Section Problem

- The critical section problem is to design a protocol that the processes can use to co- operate.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.

do
{
Entry Section
Critical Section
Exit Section
Remainder Section
}while(TRUE);

General Structure of a typical process Pi.

- ENTRY SECTION:- The process will request to enter the critical section then a part of code decide wheather process can enter in the **Critical Section** on not.
- CRITICAL SECTION:- A code segment that accesses shared resources and that has to be executed as an atomic action.
- EXIT SECTION:- Locking section can be undone which is done in **Critical Section**.
- REMAINDER SECTION:- Remaining part of the program .

- A critical section is a piece of code that only one thread can execute at a time.
- If multiple threads try to enter a critical section, only one can run and the others will sleep.
- Imagine you have three threads that all want to enter a critical section.
- Only one thread can enter the critical section; the other two have to sleep.
- When a thread sleeps, its execution is paused and the OS will run some other thread.
- Once the thread in the critical section exits, another thread is woken up and allowed to enter the critical section.

Requirements to the critical section

Three requirements:

1. **Mutual exclusion:**

   If a process is executing in critical section, then no other process can be executing in their critical section.

   Only one process can execute at a time

   This means access to the critical section must be mutually exclusive.

2. **Progress:**

   If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next and this selection cannot be postponed indefinitely.

If only one process wants to enter, it should be able to

If two or more wish to enter, one of them should succeed

Contd..,

- **Bounded waiting:**
- There exists a bound or limit on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before that request is granted.
- Assume that each process executes at a non zero speed

Critical Regions

- A critical region is a ***section of code*** that is always executed under mutual exclusion.
- Critical regions shift the responsibility for enforcing mutual exclusion from the programmer (where it resides when semaphores are used) to the compiler.

Two parts:

1. Variables that must be accessed under mutual exclusion.

2. A new language statement that identifies a critical region in which the variables are accessed.

Example:

```
Process A:
region V1 do
begin
{ Do some stuff. }
end;
region V2 do
begin
{ Do more stuff. }
end;
```

```
Process B:
region V1 do
begin
{ Do other stuff. }
end;
```

Here process A can be executing inside its V2 region while process Bis executing inside
its V1 region, but if they both want to execute inside their respective V1 regions only one
will be permitted to proceed.
Each shared variable (V1 and V2 above) has a queue associated with it. Once one process is executing code inside a region tagged
with a shared variable, any other processes that attempt to enter a region tagged with the same variable are blocked and put in the
queue.

Conditional Critical Regions

- Lack of condition synchronization
- Conditional critical regions provide condition synchronization for critical regions:

region v when B do

begin ... end; where B is a boolean expression (usually B will refer to v).

Conditional critical regions work as follows:

1. A process wanting to enter a region for v must obtain the mutex lock. If it cannot, then it is queued.
2. Once the lock is obtained the boolean expression B is tested. If B evaluates to true then the process proceeds, otherwise it releases the lock and is queued.
3. When it next gets the lock it must retest B. ndition synchronization

Explanation

- Each shared variable has *two queues* associated with it.
- The ***main queue*** is for processes that want to enter a critical region but find it locked.
- The ***event queue*** is for the processes that have blocked because they found the condition to be false.
- When a process leaves the conditional critical region the processes on the event queue join those in the main queue.

Limitations

- Conditional critical regions are still distributed among the program code.
- There is no control over the manipulation of the protected variables — no information hiding or encapsulation.
- Once a process is executing inside a critical region it can do whatever it likes to the variables it has exclusive access to.
- Conditional critical regions are more difficult to implement efficiently than semaphores.

Disabling Interrupts (Hardware Solution)

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section.

With interrupts turned off the CPU could not be switched to other process.

Hence, no other process will enter its critical and mutual exclusion achieved.

Lock Variable (Software Solution)

- In this solution, we consider a single, shared, (lock) variable, initially 0.
- When a process wants to enter in its critical section, it first test the lock.
- If lock is 0, the process first sets it to 1 and then enters the critical section.
- If the lock is already 1, the process just waits until (lock) variable becomes 0.
- Thus, a 0 means that no process in its critical section, and 1 means hold your horses - some process is in its critical section.

Semaphore

A semaphore –

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.

Accessed only through standard atomic operations: wait() and signal().

Semaphore

Semaphores

- A *semaphore* is an object that consists of a counter, a waiting list of processes and two methods (*e.g.*, functions): `signal` and `wait`.

1

Semaphore Method: wait

```
void wait(sem S)
{
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list;
        block();
    }
}
```

- After decreasing the counter by 1, if the counter value becomes negative, then
  - add the caller to the waiting list, and then
  - block itself.

Semaphore Method: signal

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume(P);
    }
}
```

- After increasing the counter by 1, if the new counter value is not positive, then
  - remove a process P from the waiting list,
  - resume the execution of process P, and return

Contd..,

- Modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly.
- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of semaphores

Binary semaphores

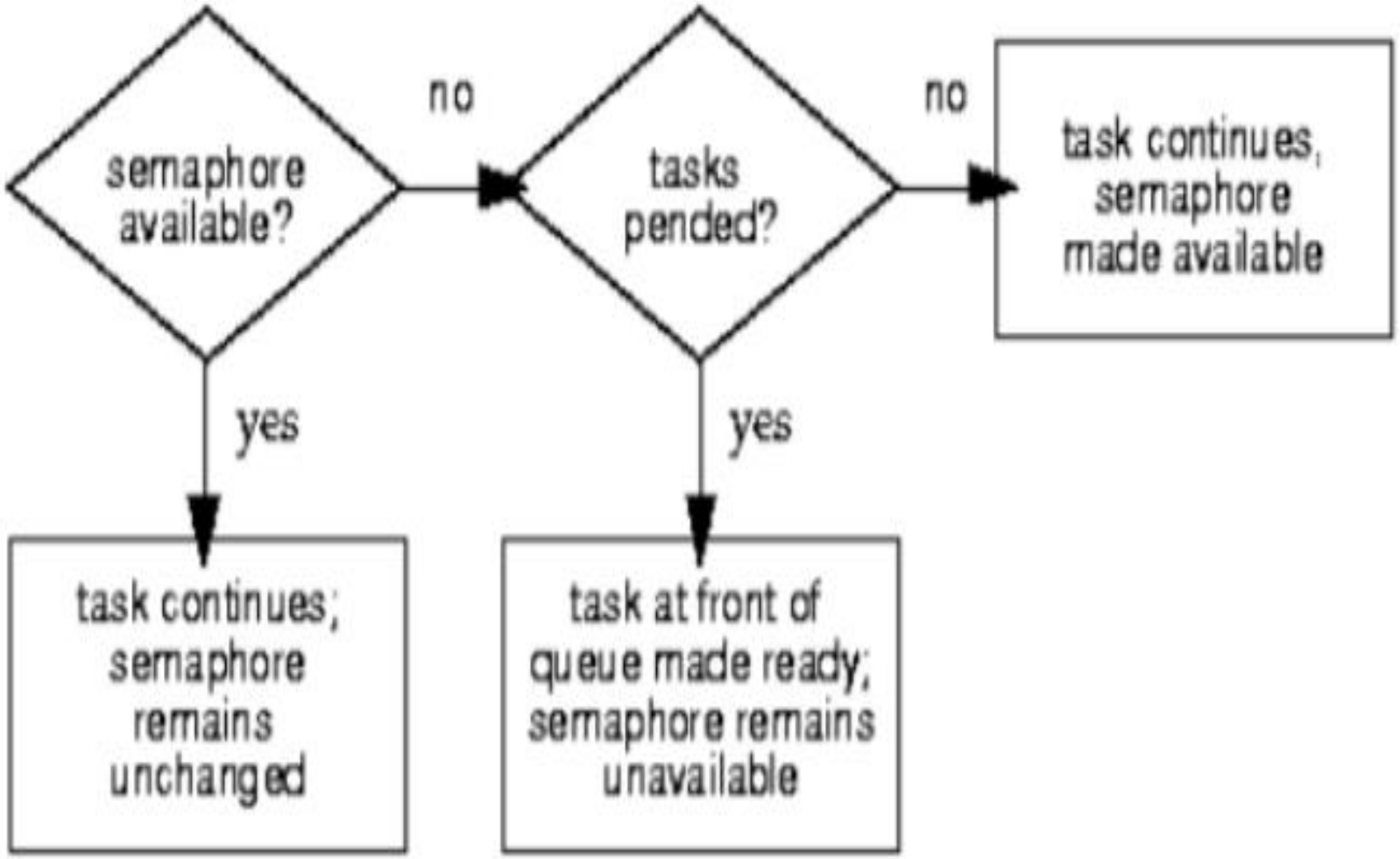
Binary semaphores have 2 methods associated with it. (up, down / lock, unlock)
Binary semaphores can take only 2 values (0/1). They are used to acquire locks. When a resource is available, the process incharge set the semaphore to 1 else 0.

Binary semaphores are known as mutex locks as they are locks that

provide mutual exclusion

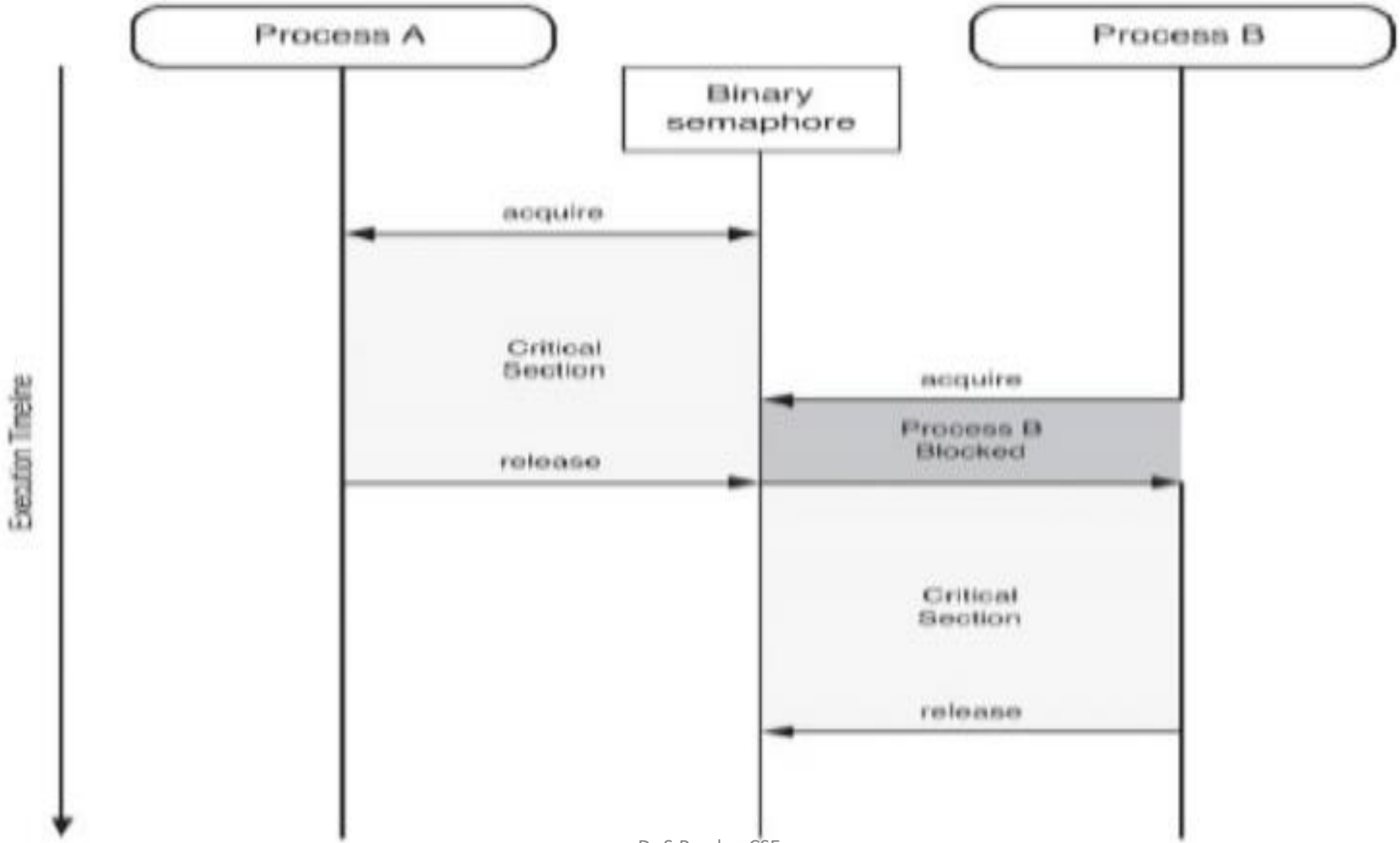
Counting semaphores

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


semaphore available?
no
tasks pended?
no
task continues, semaphore made available
yes
yes
task continues; semaphore remains unchanged
task at front of queue made ready; semaphore remains unavailable

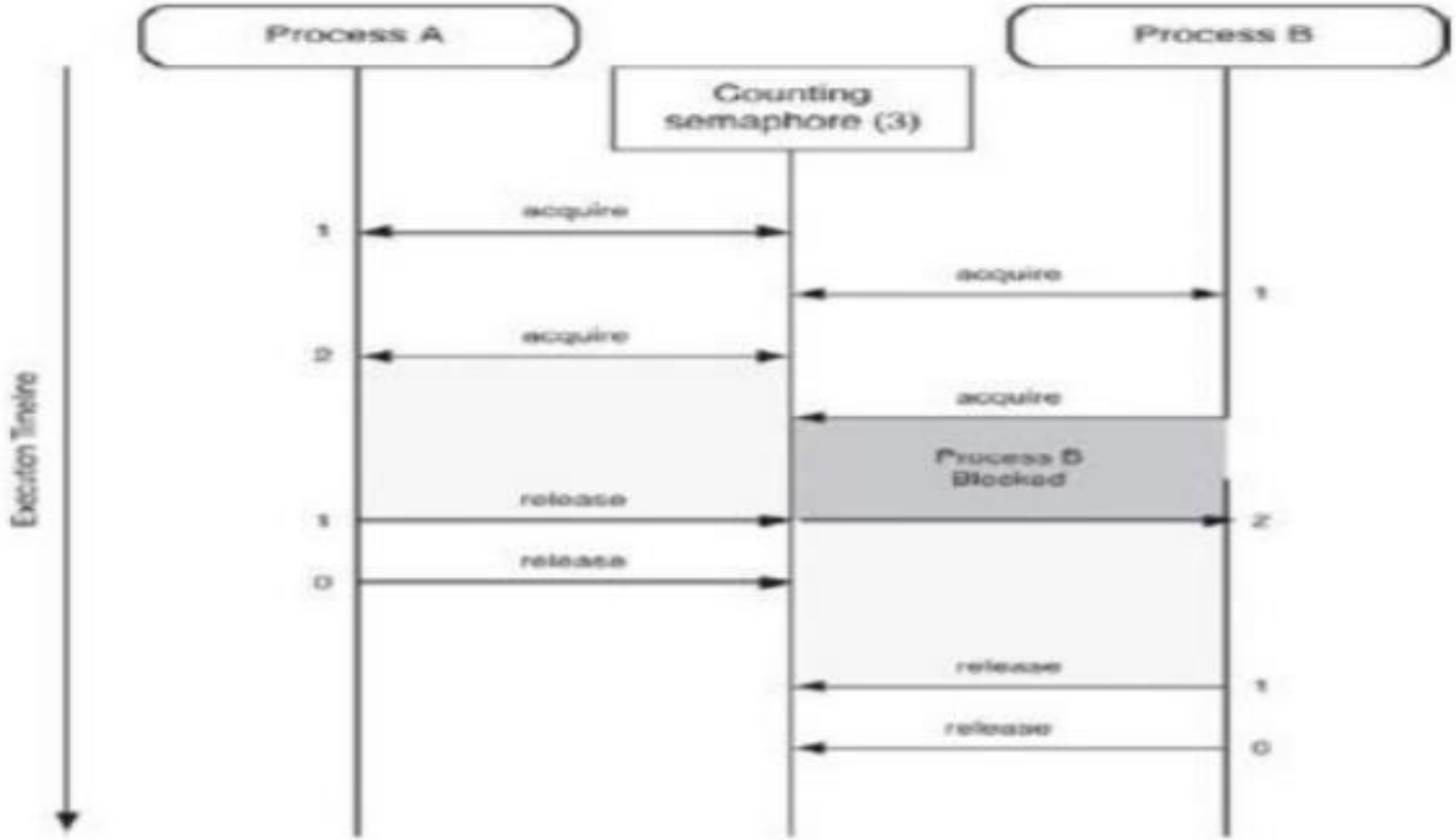
Binary Semaphore

- Process A performs the acquire first and therefore is provided with the semaphore.
- The period in which the semaphore is owned by the process is commonly called a critical section.
- The critical section can be performed by only one process, therefore the need for the coordination provided by the semaphore.
- While Process A has the semaphore, Process B is not permitted to perform its critical section.
- Note that while Process A is in its critical section, Process B attempts to acquire the semaphore.
- As the semaphore has already been acquired by Process A, Process B is placed into a blocked state. When Process A finally releases the semaphore, it is then granted to Process B, which is allowed to enter its critical section.
- Process B at a later time releases the semaphore, making it available for acquisition.


Process A
Process B
Binary semaphore
acquire
Critical Section
acquire
Process B Blocked
release
Critical Section
release
Execution Timeline

Counting Semaphores

- Each process requires two resources before being able to perform its desired activities.
- The value of the counting semaphore is 3, which means that only one process will be permitted to fully operate at a time.
- Process A acquires its two resources first, which means that Process B blocks until Process A releases at least one of its resources.


Process A
Counting semaphore (3)
Process B
Execution Timeline
acquire
1
acquire
1
2
acquire
acquire
Process B Blocked
release
1
2
release
0
release
1
release
0

Advantages of Semaphores

- Semaphores allow ***only one process*** into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is ***no resource wastage*** because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disdavantage:

Semaphores are impractical for last scale use as their use leads to ***loss of modularity.*** This happens because the wait and signal operations prevent the creation of a structured layout for the system.

Disadvantages of Semaphores

- While a process is in its critical section, other process that tries to enter its critical section must loop continuously in the entry code.
- ***Busy waiting wastes CPU cycles*** that some other process might be able to use productively. This type of semaphore is called a spinlock because the process spins while waiting for the lock

Solution:

Rather than engaging in busy waiting, ***the process can block itself***. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state.

Then control is transferred to CPU scheduler which selects another process to execute.

A process that is ***blocked waiting on a semaphore S, should be restarted when some other process executes a signal() operation.***

The process is restarted by a ***wakeup() operation*** which changes the process from the waiting state to the ready state. Process is then placed in the ready queue.

Functions

Signal() semaphore operation

```
Signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
}
```

Wait() semaphore operation

```
wait(semaphore *S)
{
S->value—;
if (S->value < 0)
{ add this process to S->list;
block();
}
}
```

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
- The event in question is the execution of a signal()operation. When such a state is reached, these processes are said to be deadlocked.

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose that *P0* executes wait (S) and then P1 executes wait (Q). When Po executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until Po executes signal(S). Since these signal () operations cannot be executed, Po and P1 are deadlocked.

Classic problems of synchronization

- In the Operating System, there are a number of processes present in a particular state.
- At the same time, we have a limited amount of resources present, so those resources need to be shared among various processes.
- Ensure that no two processes are using the same resource at the same time because this may lead to data inconsistency.
- So, synchronization of process should be there in the Operating System.(race condition, critical section)

Race Condition

```
SomeProcess(){
    ...
    read(a) //instruction 1
    a = a + 5 //instruction 2
    write(a) //instruction 3
    ...
}
```

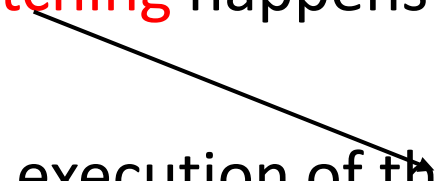
Classic problems of synchronization

Case 1

- Process P1 will be executed fully (i.e. all the three instructions) and after that, the process P2 will be executed.

Case 2

P1 starts executing. So, it reads the value of "a" from the memory and that value is 10(initial value of "a" is taken to be 10). Now, at this time, context switching happens between process P1 and P2



the execution of the process that is present in the running state is suspended by the kernel and another process that is present in the ready state is executed by the CPU.

Outcome of the execution depends on the particular order in which the access takes place(race condition)

Reader Writer –Problem statement

- If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state. Only one process should be allowed to change the value of the data present in the file at a particular instant of time.
- Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read because the process writing on the file will change the value of the file, but the process reading that file will read the old value present in the file. So, this should be avoided.

Solution is …..using semaphore

If a process is performing some write operation, then no other process should be allowed to perform the read or the write operation i.e. no other process should be allowed to enter into the critical section

If a process is performing some read operation only, then another process that is demanding for reading operation should be allowed to read the file and get into the critical section because the read operation doesn't change anything in the file. So, more than one reads are allowed. But if a process is reading a file and another process is demanding for the write operation, then it should not be allowed.

- Solution of readers writers problem using semaphore variable :
- There is two operation which we can perform on semaphore variable
  1) wait(S) = S – 1
  2) Signal(S) = S + 1
- Here we used three semaphore variable for solve this problem:

1) Mutex initialize to 1 (Binary Semaphore)
2) Wrt initialize to 1 (Binary Semaphore)
3) Read count initialize to 0 (Counting Semaphore)

➢Function of these variable:

- **Mutex** : Used to lock critical section for both readers and writers.
- **Wrt** : Used to block the writers from entering the critical section.
- **Read count** : Helps counting the number of readers in the critical section and permits the writer to enter when it become zero.

Structure of Reader –Writer Process

Structure of Readers process:

```
Wait (mutex);
Read count= Read count +1;
If (Read count == 1)
Then
Wait (wrt);
Signal(mutex);
. . . . . .
Reading is performed
. . . . . .
```

```
Wait (mutex)
Read count= Read count – 1;
If ( Read count == 0)
Then
Signal (wrt);
Signal (mutex);
```

Structure of Writers process:

```
Wait (Wrt);
. . . . . .
Writing is performed
. . . . . .
Signal (wrt);
```

Dining philosopher

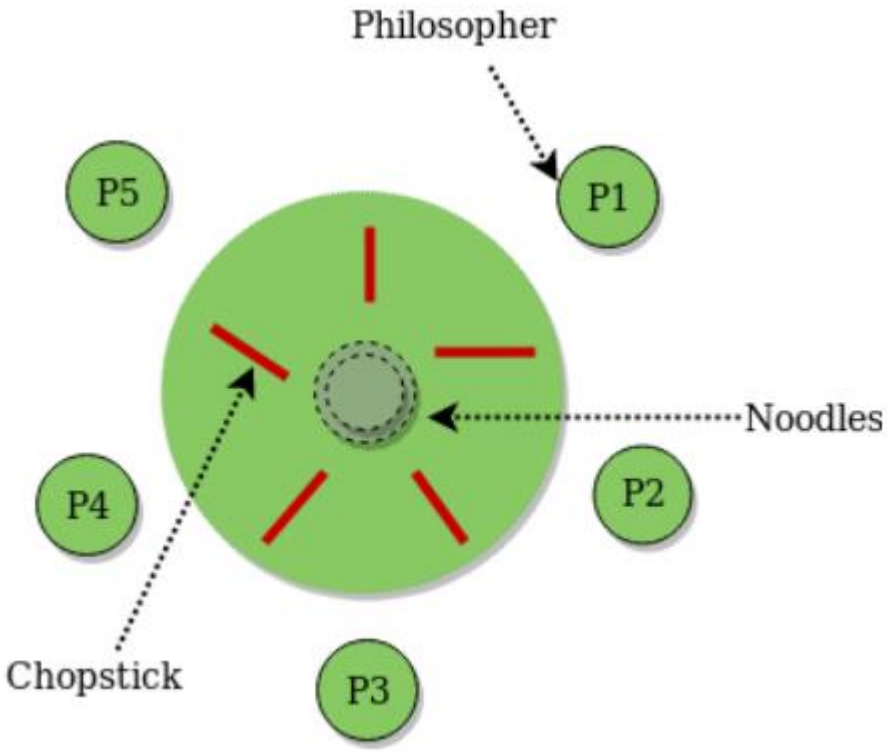
The **dining philosophers problem** is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

- **concurrency** is the decomposability property of a program, algorithm, or problem into order-independent or partially ordered components or units
- This means that even if the concurrent units of the program, algorithm, or problem are executed out-of-order or in partial order, the final outcome will remain the same.
- This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems

Problem Statement

- Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers
- Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks.
- Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher.

Explanation



At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

Solution with semaphore

```
while(TRUE) {
wait(stick[i]);
wait(stick[(i+1) % 5]);  // mod is used because if i=5, next
                    // chopstick is 1 (dining table is circular)
/* eat */
signal(stick[i]);
signal(stick[(i+1) % 5]);
/* think */
}
```

When a philosopher wants to eat the rice or spaghetti , he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a **deadlock situation** occurs because they will be waiting for another chopstick forever.

A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
Allow at most four philosophers to be sitting simultaneously at the table
Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick whereas an even philosopher picks up her right chopstick and then her left chopstick.

Problems in semaphore

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in ***timing errors that are difficult to detect*** since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

Case 1

Suppose that a process interchanges the order in which the wait () and signal () operations on the semaphore mutex are executed.

Signal (mutex); …….

Critical section ……..

Wait (mutex);

Here several processes may be executing in their critical sections simultaneously, violating the mutual exclusion requirement.

Problems in semaphore

Case 2

- Suppose that a process replaces signal (mutex) with wait (mutex) that is it executes
- Wait(mutex); ……
- Critical section …….
- Wait(mutex);
- Here a ***deadlock*** will occur.

Case 3:

Suppose that a process omits the wait(mutex), or the signal(mutex) or both. Here, either mutual exclusion is violated or a dead lock will occur

Monitor - high level synchronization

- Presents a set of ***programmer defined operations*** that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of the procedures or functions that operate on those variables.
- The representation of a monitor type cannot be used directly by the various processes.
- Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local procedures.

```
name: monitor
   ...local declarations
   ...initialize local data
   proc1 (...parameters)
      ...statement list
   proc2 (...parameters)
      ...statement list
   proc3 (...parameters)
      ...statement list
```

Ensures that only one process at a time can be active within the monitor. But this monitor construct is not powerful for modeling some synchronization schemes.

To make more powerful..we use condition construct.

The only operations that can be invoked on a condition variable are wait() and signal(). The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal().

The x.signal() operation resumes exactly one suspended process.

When x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x.

If suspended process Q is allowed to resume its execution, the signaling process P must wait.

Otherwise, both P and Q would be active simultaneously within the monitor

However, both processes can conceptually continue with their execution. Two possibilities exist:

1. Signal and wait – P either waits until Q leaves the monitor or waits for another condition.
2. Signal and condition – Q either waits until P leaves the monitor or waits for another condition.

Dining Philosophers Solution using Monitors

- Solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- Three states of philosopher: Hungry,Eating,Thinking
- **THINKING** – When philosopher doesn't want to gain access to either fork.
- **HUNGRY** – When philosopher wants to enter the critical section.
- **EATING** – When philosopher has got both the forks, i.e., he has entered the section.
- Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).

```
monitor DP
{
    status state[5];
    condition self[5];

    // Pickup chopsticks
    Pickup(int i)
    {
        // indicate that I'm hungry
        state[i] = hungry;

        // set state to eating in test()
        // only if my left and right neighbors
        // are not eating
        test(i);

        // if unable to eat, wait to be signaled
        if (state[i] != eating)
            self[i].wait;
    }

   Putdown(int i)
   {

        // indicate that I'm thinking
        state[i] = thinking;

        // if right neighbor R=(i+1)%5 is hungry and
        // both of R's neighbors are not eating,
        // set R's state to eating and wake it up by
        // signaling R's CV
        test((i + 1) % 5);
        test((i + 4) % 5);
```

```
    test(int i)
    {

        if (state[(i + 1) % 5] != eating
            && state[(i + 4) % 5] != eating
            && state[i] == hungry) {

            // indicate that I'm eating
            state[i] = eating;

            // signal() has no effect during Pickup(),
            // but is important to wake up waiting
            // hungry philosophers during Putdown()
            self[i].signal();
        }
    }

    init()
    {

        // Execution of Pickup(), Putdown() and test()
        // are all mutually exclusive,
        // i.e. only one at a time can be executing
for
    i = 0 to 4

        // Verify that this monitor-based solution is
        // deadlock free and mutually exclusive in that
        // no 2 neighbors can eat simultaneously
        state[i] = thinking;
    }
} // end of monitor
```

Philosopher i must
invoke the operations pickup() and putdown() in the following sequence:

We also need to declare

```
condition self[5];
```

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs. We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
                ...
                eat
                ...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that **no two neighbors** are eating simultaneously and that no deadlocks will occur.

Implementing a Monitor using Semaphores

- For each monitor, a semaphore mutex initialized to 1 is provided.
- A process must execute wait(mutex) before entering the monitor and must execute(signal) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore next initialized to 0, on which the signaling processes may suspend themselves.
- An integer variable next_count is used to count the number of processes suspended on next.
- Thus, each external procedure P is replaced by

wait(mutex);

body of F

if (next_count > 0)

signal(next);

else

signal(mutex);

Mutual exclusion within a monitor is thus ensured.

Resuming Processes Within a Monitor

- If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then for determining which suspended process should be resumed next, we use FCFS ordering so that the process waiting the longest is resumed first.
- Or conditional wait construct() can be used as x.wait(c); where c is an integer expression that is evaluated when the wait () operation is executed.
- The value of c, which is called a priority number, is then stored with the name of the process that is suspended.
- When x. signal () is executed, the process with the smallest associated priority number is resumed next.

A monitor to allocate a single resource

- ResourceAllocator monitor controls the allocation of a single resource among competing processes.
- Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource.
- The monitor allocates the resource to the process that has the shortest timeallocation request.
- A process that needs to access the resource in question must observe the following sequence:
- R.acquire(t);
- access the resource;
- R. release O ;

Problems:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.

A process might attempt to release a resource that it never requested.

- A process might request the same resource twice (without first releasing the resource).

Solutions:

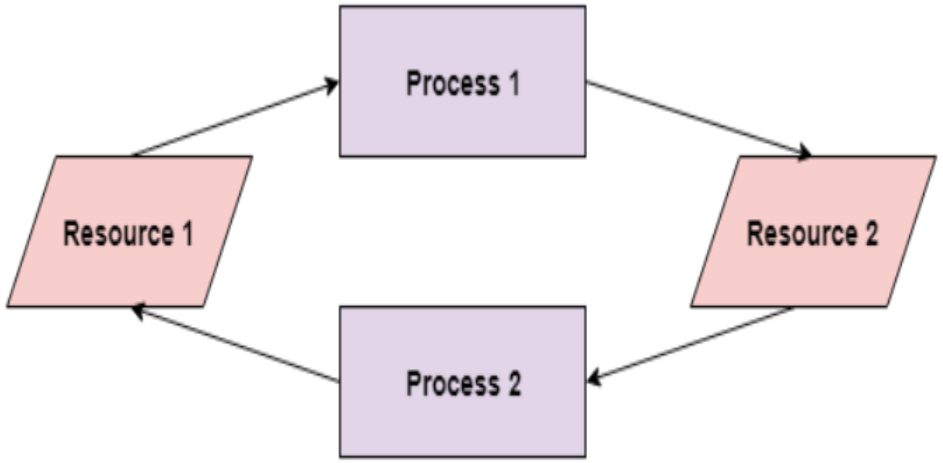
Inspect all the programs that make use of the ResourceAllocator monitor and its managed resource.

Check two conditions to establish the correctness of this system.

First, user processes must **always make their calls** on the monitor in a correct sequence.

Second, Ensure that an ***uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor*** and try to access the shared resource directly, without using the access protocols.

Deadlock



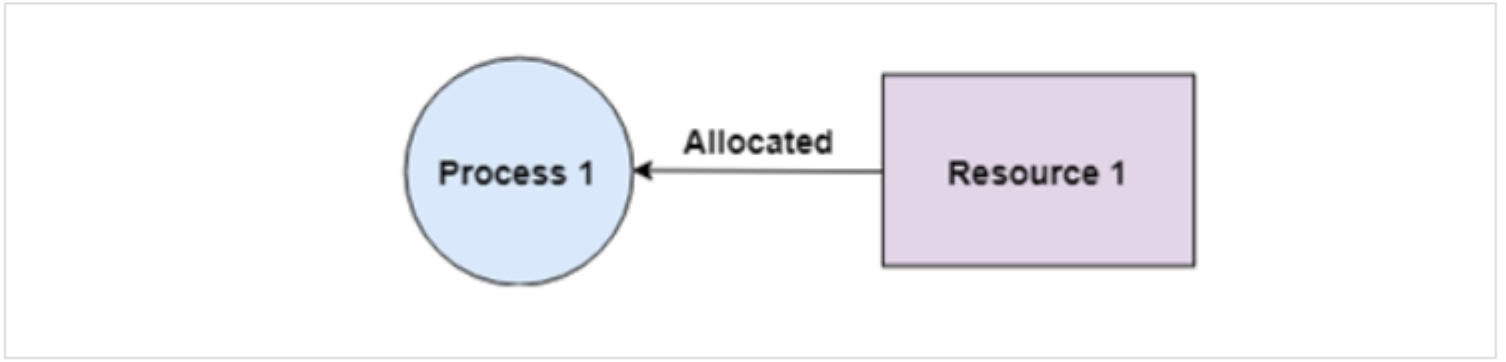
Deadlock in Operating System

- A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.

Coffman conditions

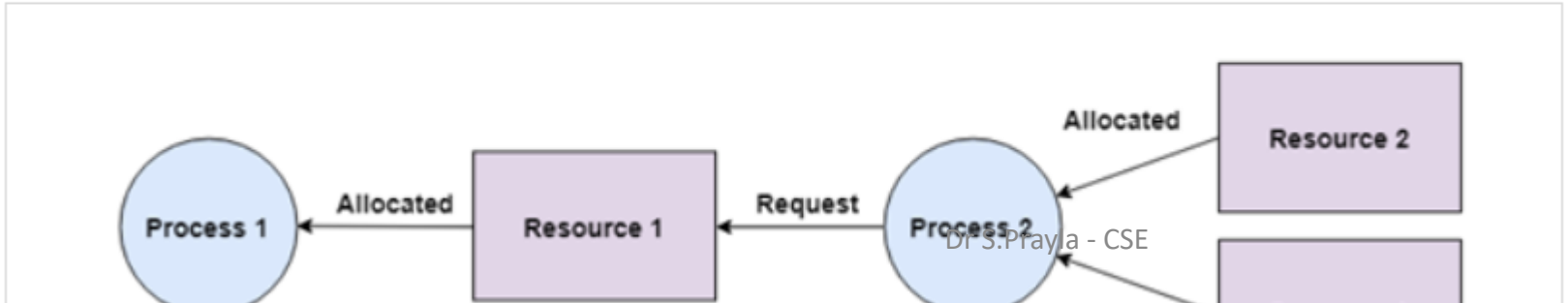
- **Mutual Exclusion**

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



Hold and Wait

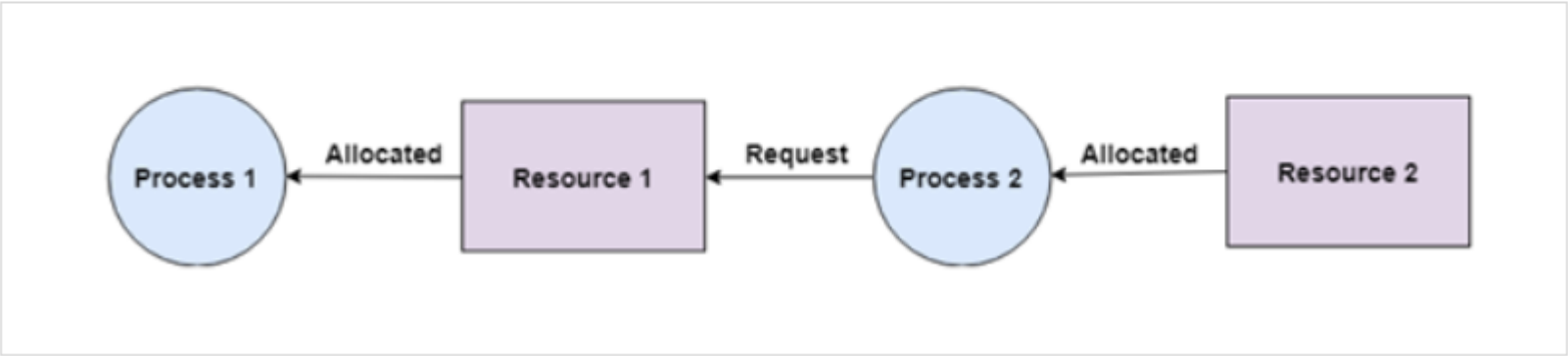
A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



Coffman conditions

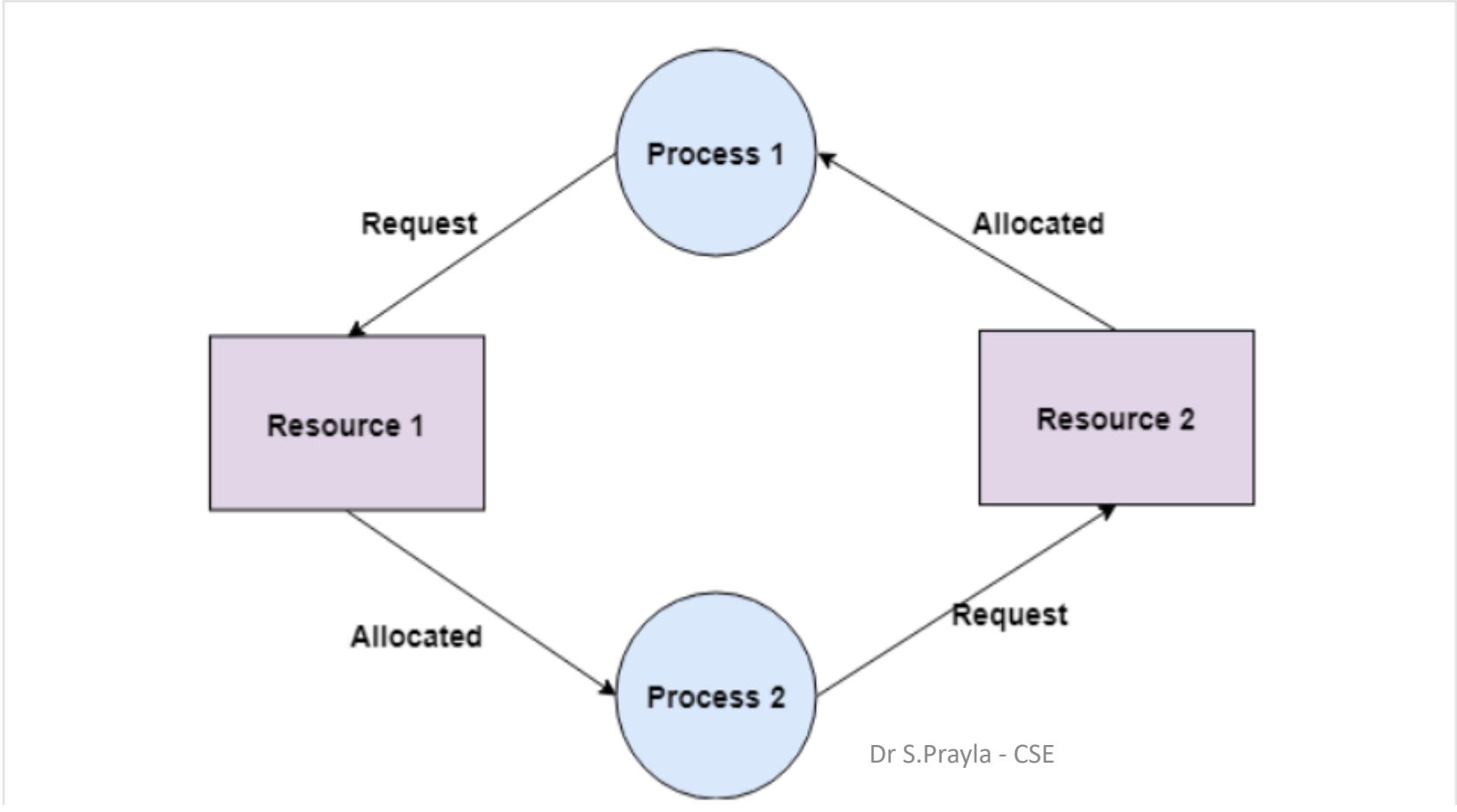
No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Coffman conditions

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Resource Allocation Graph (RAG) in Operating System

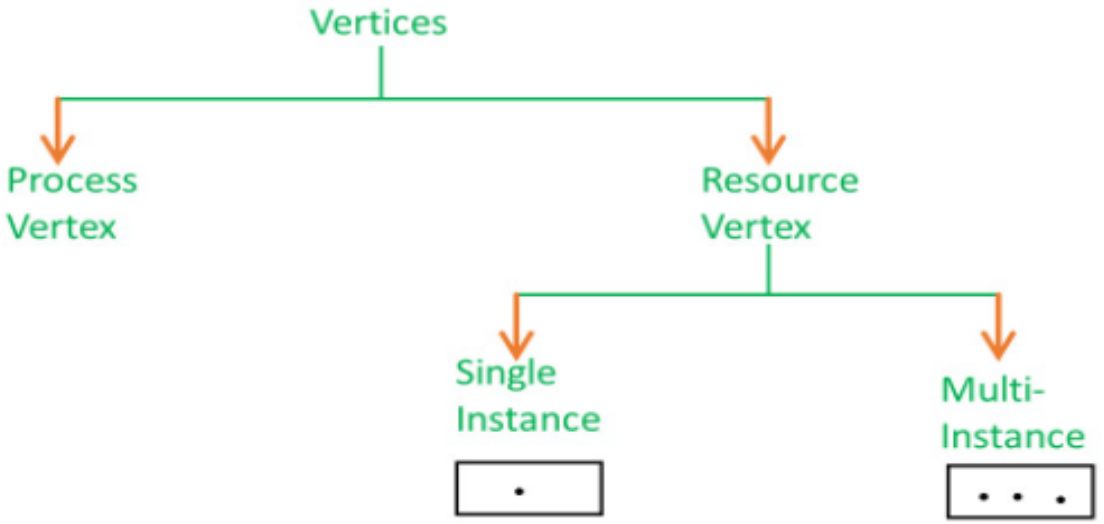
- what is the state of the system in terms of processes and resources.
- how many resources are available, how many are allocated and what is the request of each process.
- Everything can be represented in terms of the diagram.

RAG	Table
possible to see a deadlock directly by using RAG	Not easy as RAG
if the system contains less number of process and resource.	better if the system contains lots of process and resource

RAG

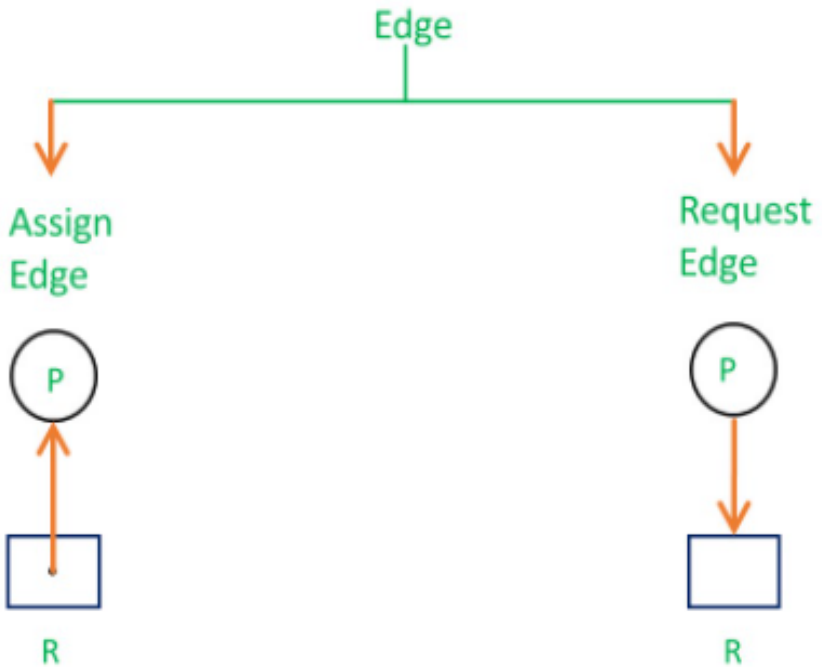
1. **Process vertex –** Every process will be represented as a process vertex.Generally, the process will be represented with a circle.
2. **Resource vertex –** Every resource will be represented as a resource vertex. It is also two type –

- **Single instance type resource –** It represents as a box, inside the box, there will be one dot.So the number of dots indicate how many instances are present of each resource type.
- **Multi-resource instance type resource –** It also represents as a box, inside the box, there will be many dots present.

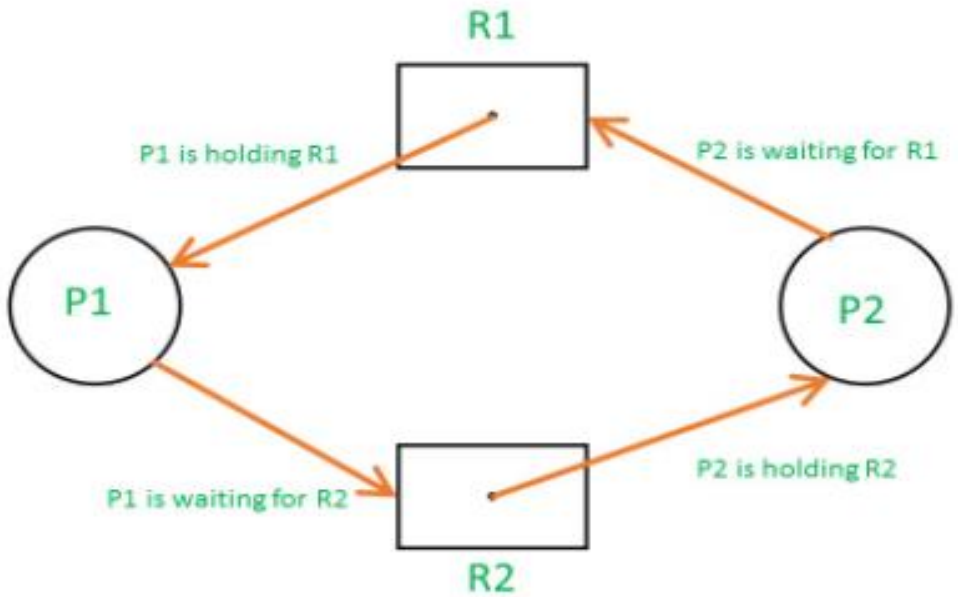


RAR-EDGES

1. **Assign Edge –** If you already assign a resource to a process then it is called Assign edge.
2. **Request Edge –** It means in future the process might want some resource to complete the execution, that is called request edge.



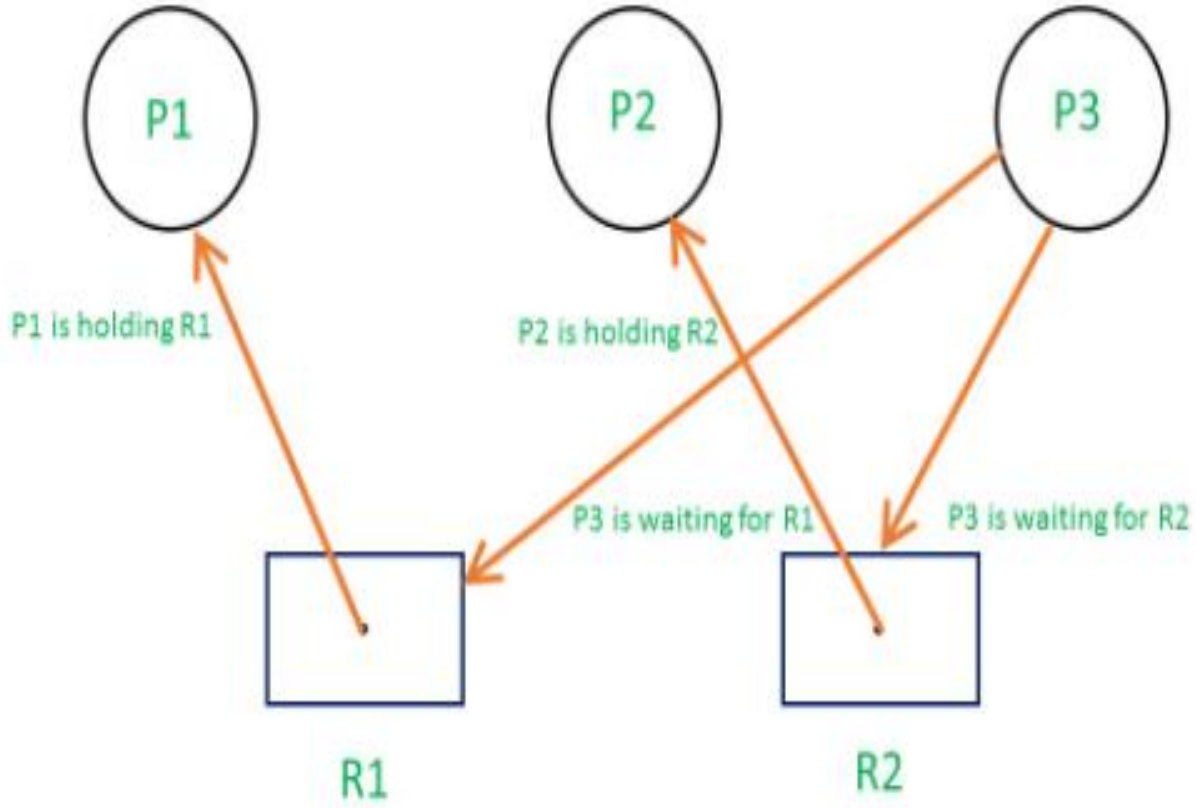
Example 1 (Single instances RAG) –



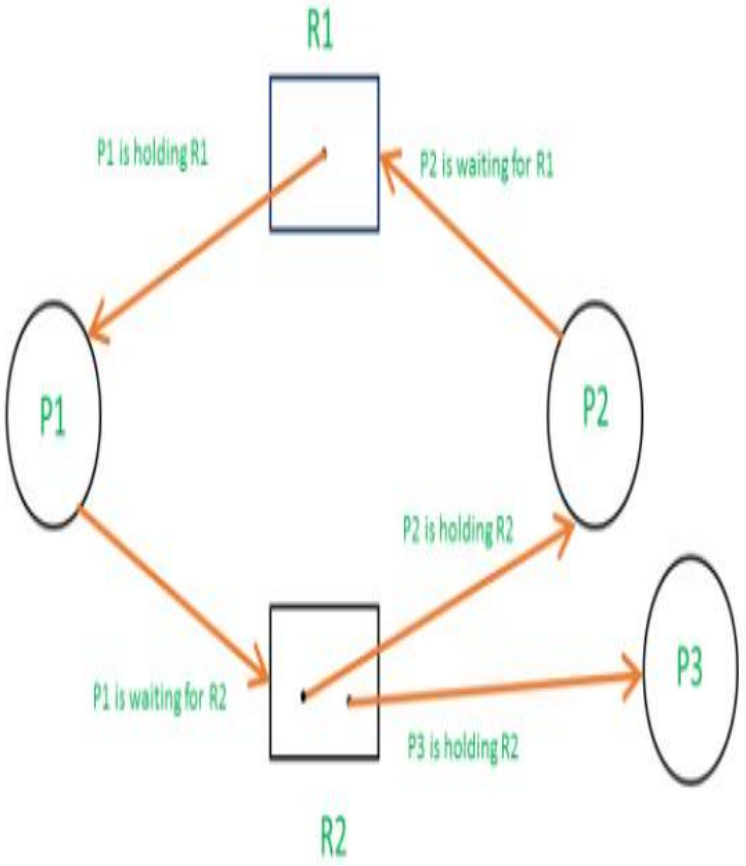
SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

Single instance resource type with deadlock



Multi instance resource type without deadlock



safe state or unsafe state

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

safe state or unsafe state

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

Allocation matrix –

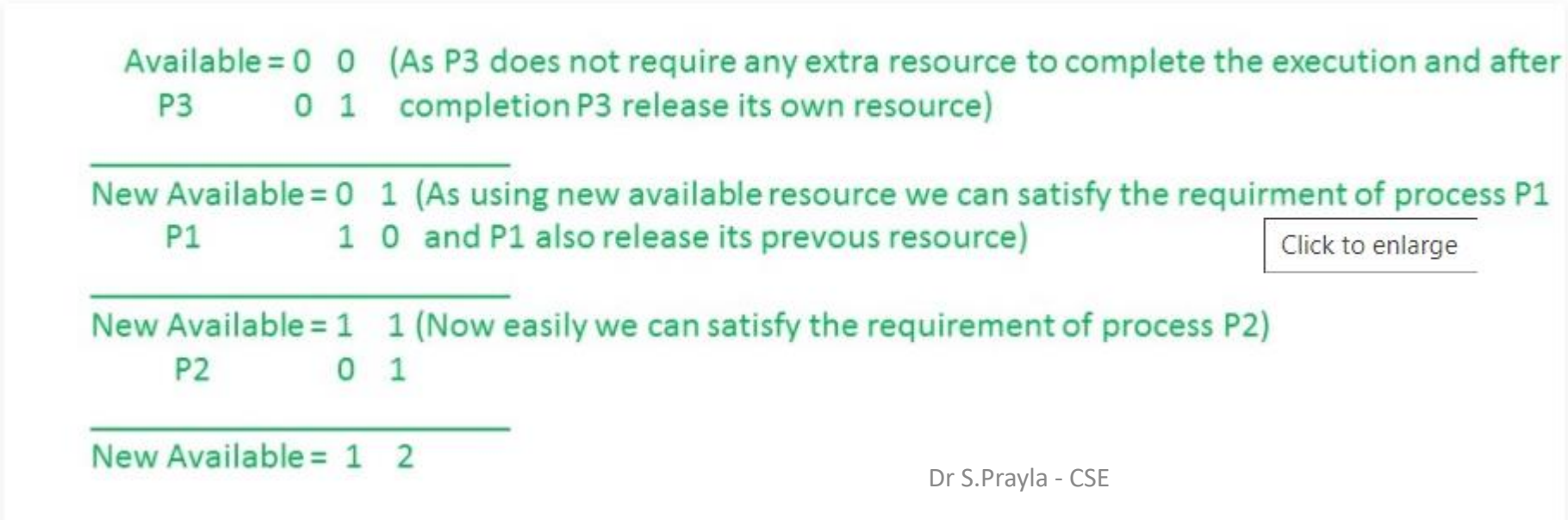
- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

Request matrix –

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.

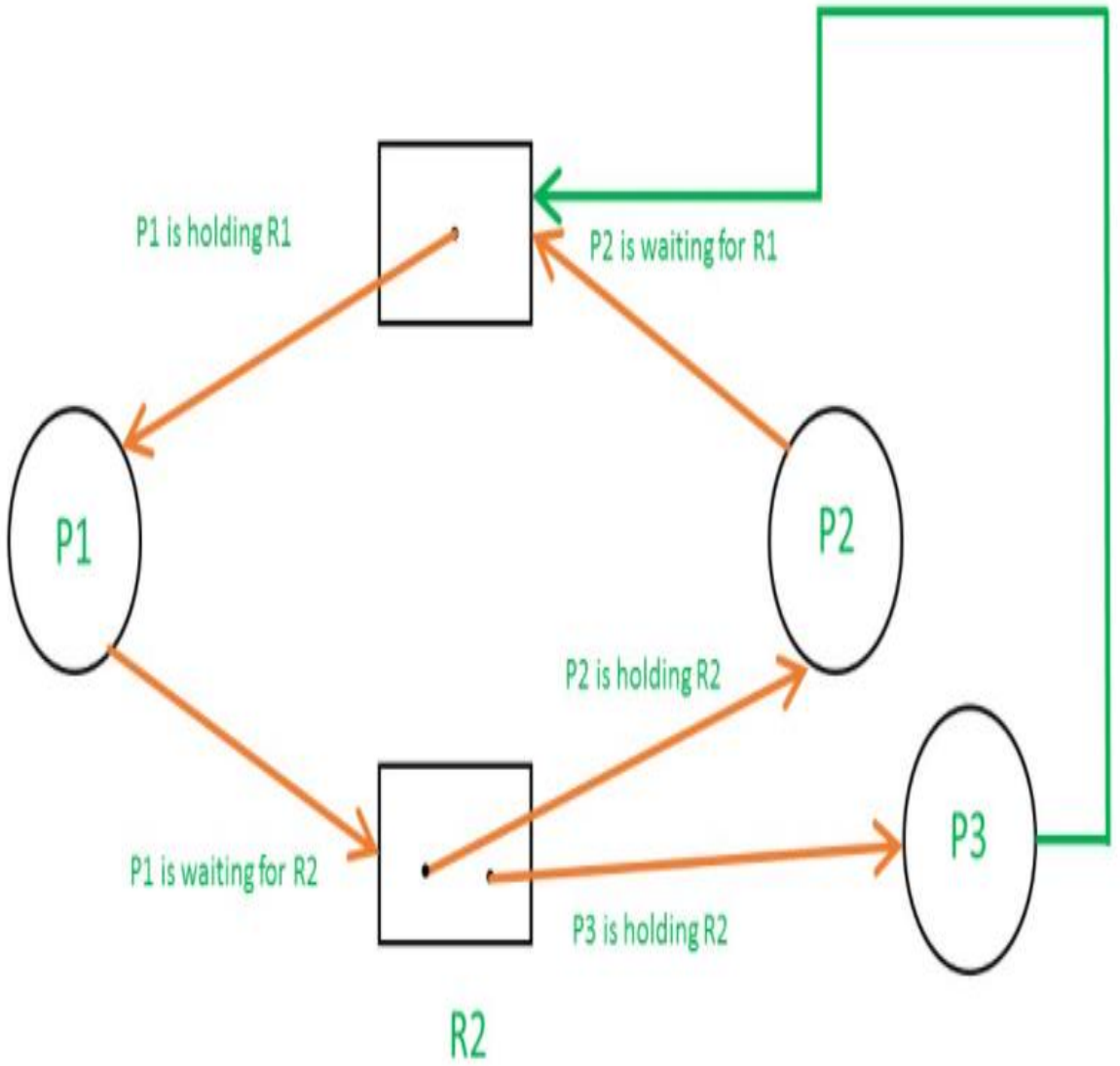
So now available resource is = (0, 0).

Checking deadlock (safe or not) –



There is no deadlock in this RAG.
Even though there is a cycle, still there is no deadlock.
Therefore in multi-instance resource cycle is not sufficient condition for deadlock.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0



So,the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement.Therefore, it is in deadlock.

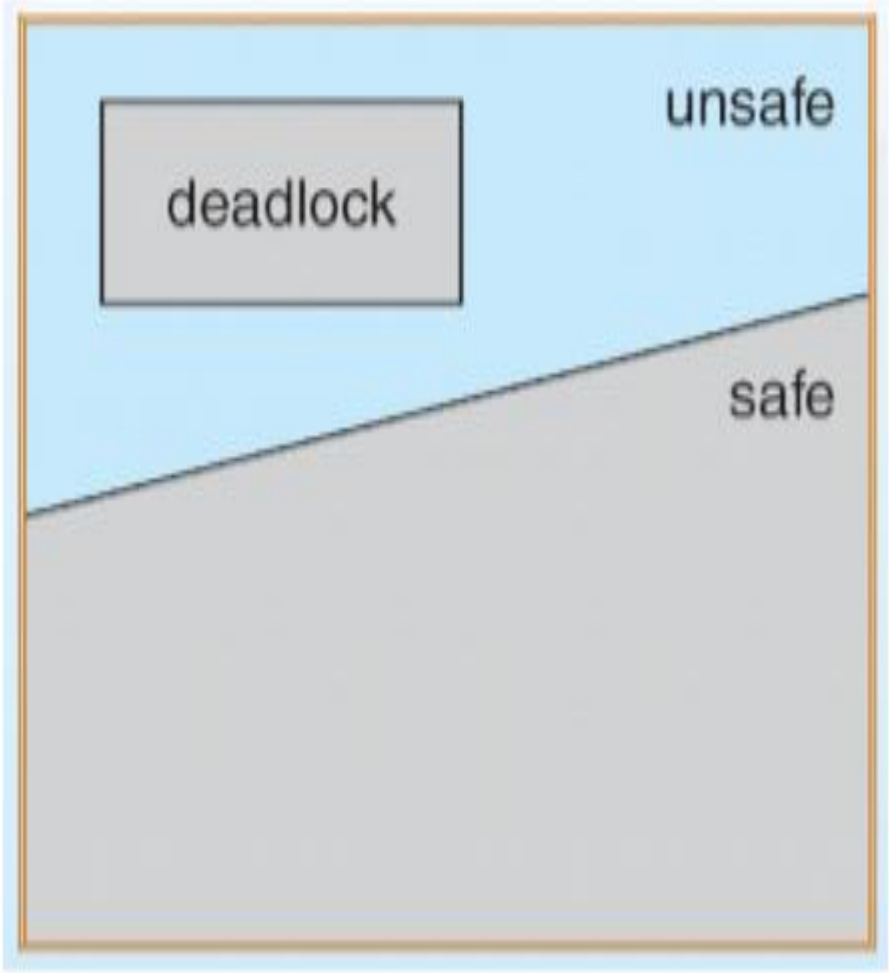
Deadlock Avoidance

- Requires that the system has some ***additional or priori information*** available.
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The *deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.*
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj,
- If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
- When Pi terminates, Pi+1 can obtain its needed resources, and so on.

Safe, Unsafe , Deadlock State

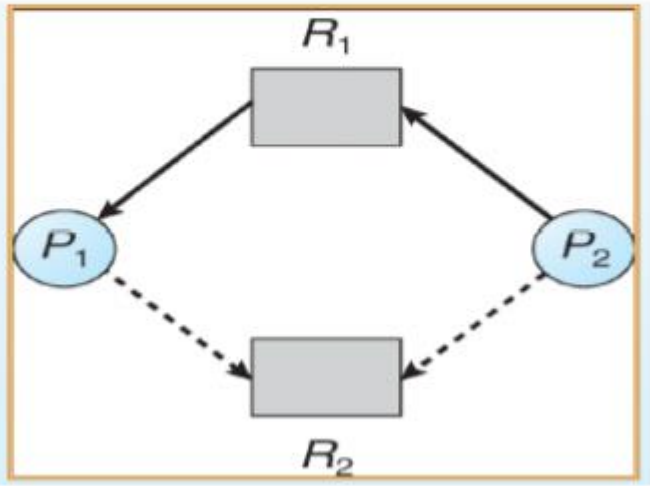


- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Resource-Allocation Graph Algorithm

- Claim edge Pi → Rj indicated that process Pi may request resource Rj; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

Resource-Allocation Graph For Deadlock Avoidance



Bankers algorithm

- Banker's Algorithm is used majorly in the banking system to avoid deadlock. It helps to identify whether a loan will be given or not.

- Used to test for safely simulating the allocation for determining the maximum amount available for all resources.
- It also checks for all the possible activities before determining whether allocation should be continued or not.

- For example, there are X number of account holders of a specific bank, and the total amount of money of their accounts is G.

- When the bank processes a car loan, the software system subtracts the amount of loan granted for purchasing a car from the total money (G+ Fixed deposit + Monthly Income Scheme + Gold, etc.) that the bank has.

- It also checks that the difference is more than or not G.
- It only processes the car loan when the bank has sufficient money even if all account holders withdraw the money G simultaneously.

Explanation

- I have $24 with me. And I want to share it with three of my friends(A,B,C), A needs $8 dollars, B needs $13, and C needs $10.

- I have already lent $6 to A, $8 to B, and $7 to C, So I am left with $24 – $21 (6+8+7) = $3

- Even after giving $6 to A, she still needs $2. Similarly, B needs $5 more and c $3.

- Until they get the amount they need, they can neither do whatever tasks they have to nor return the amount they borrowed.
- So I can pay $2 to A, and wait for her to get her work done and then get back the entire $8.

- Or, I can pay $3 to C and wait for her to pay me back after task is done.
- I can't pay B because she needs $5 and I don't have enough.

- I can pay her once A or C returns the borrowed amount after their work is done.

- This state is termed as the ***safe state***, where everyone's task is completed and, eventually, I will get all my money back.

Unsafe state or Deadlock state

- Knowing B needs $10 urgently, instead of giving $8, I end up giving $10.
- And I am left with only $1.
- In this state, A still needs $2 more, B needs $3 more, and still needs $3 more, but now I don’t have enough money to give them and until they complete the tasks they need the money for, no money will be transferred back to me.
- This kind of situation is called the ***Unsafe state or Deadlock state***, which is solved using Banker’s Algorithm.
- The goal of the Banker’s algorithm is to handle all requests without entering into the unsafe state, also called a deadlock.

1. How much of each resource each person could maximum request [MAX]
2. How much of each resource each person currently holds [Allocated]
3. How much of each resource is available in the system for each person [Available]

So we need MAX and REQUEST.

If REQUEST is given MAX = ALLOCATED + REQUEST

NEED = MAX – ALLOCATED

A resource can be allocated only for a condition.

REQUEST<= AVAILABLE or else it waits until resources are available.

Process	Allocated			Maximum			Available			Need (Maximum Allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2				1	2	2
P3	4	0	1	9	0	4				5	0	3
P4	2	1	1	2	2	2				0	1	1

Need P2<Available, so we allocate resources to P2 first.

After P2 completion the table would look as

Process	Allocated			Maximum			Available			Need (Maximum Allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	5	3	2	7	4	3
P3	4	0	1	9	0	4				5	0	3
P4	2	1	1	2	2	2				0	1	1

Need P4<Available, so we allocate resources to P4.

After P4 completion

Process	Allocated			Maximum			Available			Need (Maximum Allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	7	4	3	7	4	3
P3	4	0	1	9	0	4				5	0	3

And P3 will be allocated before P1, which gives us the sequence P2, P4, P3, and P1 without getting into deadlock.

Deadlock Detection

- A Deadlock detection algorithm, determines whether the system is in deadlock state or not?
- If yes, then the system uses a deadlock recovery method to break the deadlock
- ***Two deadlock detection methods depending upon the number of instances of each resource.***

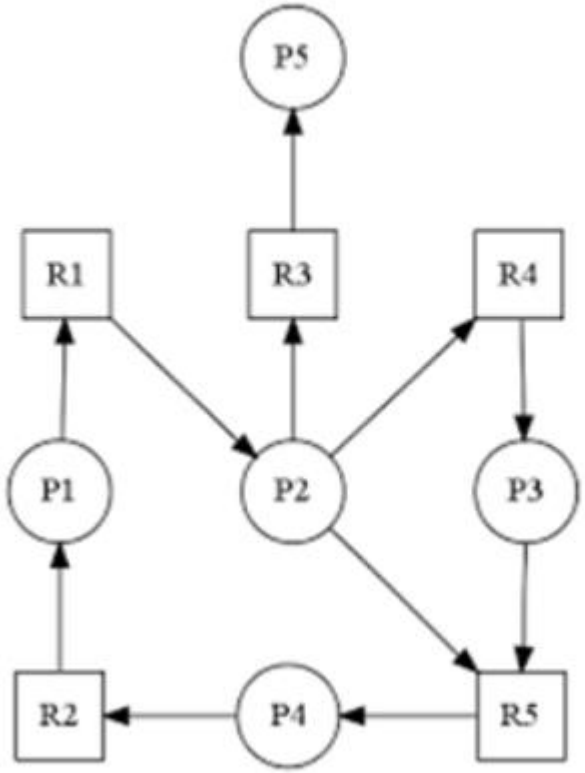
1. Single Instance of Each Resource: *wait-for-graph*
2. Multiple Instance of Each Resource

Single Instance of Each Resource: wait-for-graph

- Pi → Pj if Pi is waiting for Pj, Periodically invoke algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

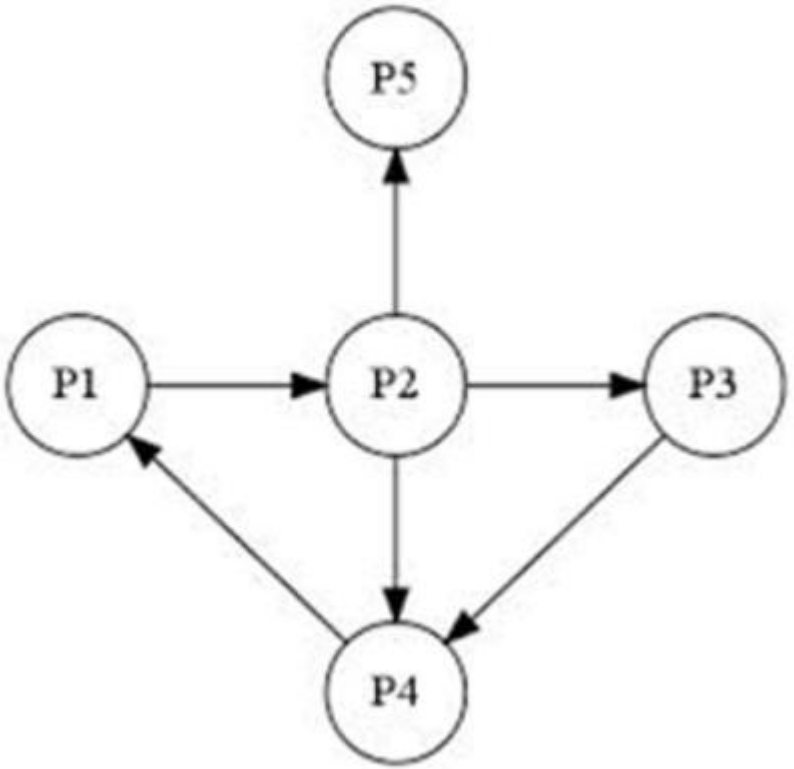
Working of wait-for-graph

Resource Allocation Graph



Resource Allocation Grsph

wait-for-graph



wait-for-graph

Multiple Instance of Each Resource

. *Available:* A vector of length *m* indicates the number of available resources of each type.

. *Allocation:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process

. *Request:* An *n* x *m* matrix indicates the current request of each process. For example, if *Request* [*ij*] = *k*, then process *Pi* is requesting *k* more instances of the resource type. *Rj*.

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively
Initialize:
(a) *Work* = *Available*
(b) For *i* = 1,2, ..., *n*, if $Allocation_i \neq 0$, then
Finish[i] = false; otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
(a) *Finish*[*i*] == *false*
(b) $Request_i <= Work$
If no such *i* exists, go to step 4.
3. *Work* = *Work* + $Allocation_i$
Finish[*i*] = *true*
go to step 2.
4. If *Finish*[*i*] == false, for some *i*, 1 <= *i* <= *n*, then the system is in deadlock state.
Moreover, if *Finish*[*i*] == *false*, then *Pi* is deadlocked.

Algorithm requires an order of O(*m* x *n*2) operations to detect whether the system is in deadlocked state.

Example for deadlock detection

Process	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
Po	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
p4	0	0	2	0	0	2			

P0 request can be satisfied with the Available resources. Hence, the first process in safe sequence is P0.

Next, update the Available using

Available=Available + Allocation of P0

So, the new Available is <0,1,0>

The next process whose Request can be satisfied with updated Available is P2. So, the updated safe frequencies is <P0,P2>

Available=Available + Allocation of P2

So, the new Available is <3,1,3>

Next, in the same manner find a process whose request can be satisfied. So, the final sequence is <P0,P2,P3,P1,P4>

Invoke detection algorithm?

1. Every time a process requests for a resource
    - Advantages
        - It is easy to identify the processes involved in the deadlock
        - Also, the process which caused the deadlock is known. This is possible because the system invokes the algorithm after each request.
    - Disadvantage
        - Overhead in computation time
2. After fixed-interval of time
    - Disadvantage
        - Can not tell which process caused the deadlock

Deadlock Recover

Process termination

- Terminate all the deadlocked processes.(killing the Process)
- Terminate processes one by one until the deadlock is broken. However, the issue is to decide which process to terminate.

Certain factors that can be used are:

- How long the process has computed?
- How much the process has finished its working?
- What is the priority of the process?
- How many resources are being used by the process?
- How many total processes will be required to be terminated?

Deadlock Recover

- Resources are preempted from certain processes and these resources can then be allocated to other process.

Issues

- Select a victim – which process to select for preempting the resources.
- Rollback – the process whose resources are preempted, can not continue execution. So, a process
  - must be rolled back to some safe state.
  - Or abort the process and restart
- Starvation – it must be ensured that the same process does not get selected for resource preemption, as this will lead to starvation.