KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

Module 3

Process synchronization

CONTENTS

Process synchronization- Race conditions – Critical section problem – Peterson's solution, Synchronization hardware, Mutex Locks, Semaphores, Monitors – Synchronization problems -Producer Consumer, Dining Philosophers and Readers-Writers.

Deadlocks: Necessary conditions, Resource allocation graphs, Deadlock prevention, Deadlock avoidance – Banker's algorithms, Deadlock detection, Recovery from deadlock.

Process are classified into two category :

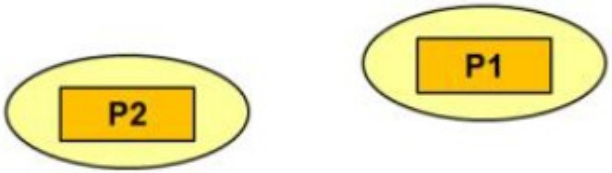
- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperating Process:** Execution of one process affects the execution of other processes

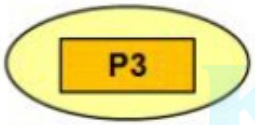
(or)

Any updating in one process may affect the other process executing in the system.

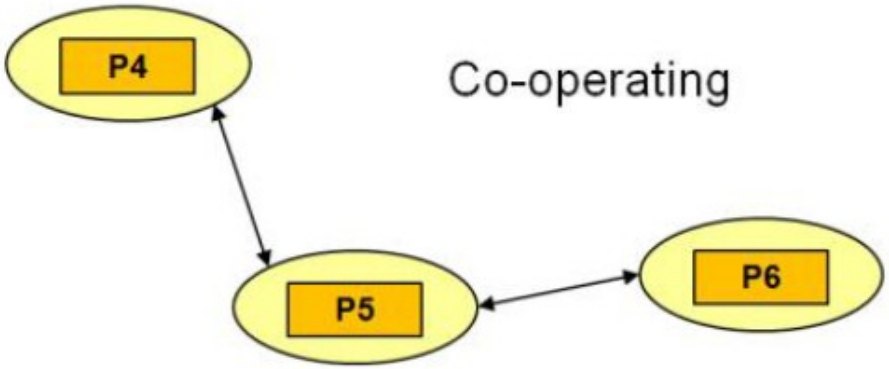
Process synchronization problem arises in the case of Cooperative process.

Independent and Cooperative Processes





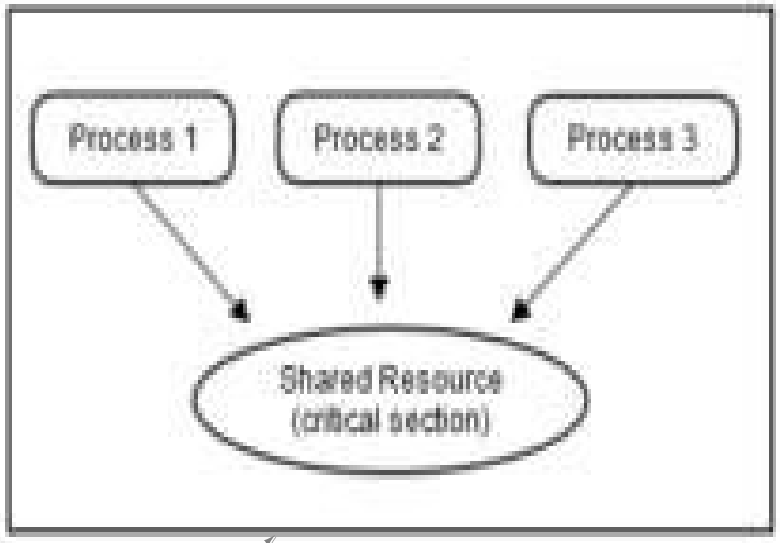
Independent



Co-operating

What is process synchronization?

- Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.
- Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.



Race Condition

- When more than **one processes** are executing the same code or accessing the same memory or any shared variable/resource/code in that condition there is a possibility that **the output or the value of the shared variable is wrong** so for that all the processes doing race to say that my output is correct this condition known as **race condition**.
- Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Producer and consumer Problem

Producer

```
while (true) {
    /* produce an item in next
produced */

while (counter ==BUFFER_SIZE)
;
       /* do nothing */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

- **Consumer**

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;

        counter--;
    /* consume the item in
next consumed */
}
```

Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```


Consider this execution interleaving with "count = 5" initially:

T0: producer execute $register_1 = counter$ *{*$register_1 = 5$*}*

T1: producer execute $register_1 = register_1 + 1$ *{*$register_1 = 6$*}*

T2: consumer execute $register_2 = counter$ *{*$register_2 = 5$*}*

T3: consumer execute $register_2 = register_2 - 1$ *{*$register_2 = 4$*}*

T4: producer execute $counter = register_1$ *{counter = 6}*

T5: consumer execute $counter = register_2$ *{counter = 4}*

Race Condition cont..

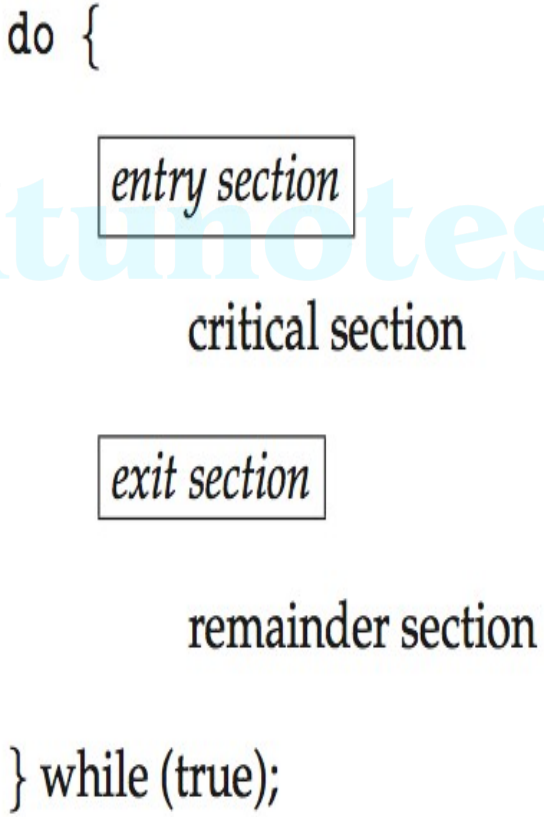
we have arrived at the incorrect state “counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at *T4 and T5, we would arrive at the incorrect state* “counter == 6”.

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**
- **Entry Section: Sections contains code which is used by the process to get permission to enter into critical section.**
- **Remainder section : contains the remaining codes.**
- **Exit section : Contains code that is used by the process to come out of the CS**

Critical Section cont…

- General structure of process P_i



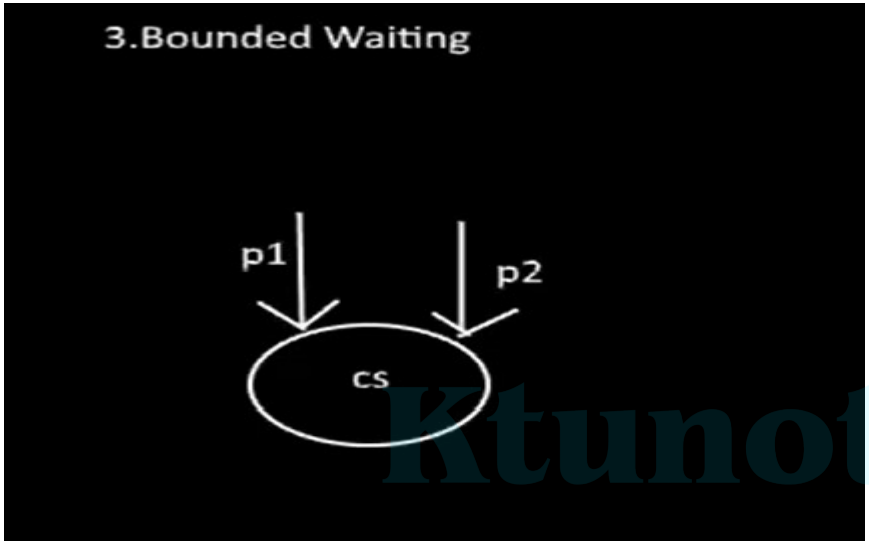
Sections of a Program

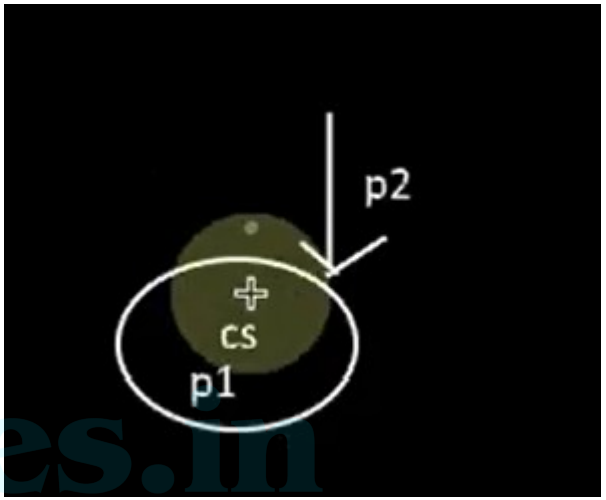
- Here, are four essential elements of the critical section:
  - **Entry Section:** It is part of the process which decides the entry of a particular process.
  - **Critical Section:** This part allows one process to enter and modify the shared variable.
  - **Exit Section:** Exit section allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It also checks that a process that finished its execution should be removed through this Section.
  - **Remainder Section:** All other parts of the Code, which is not in Critical, Entry, and Exit Section, are known as the Remainder Section.

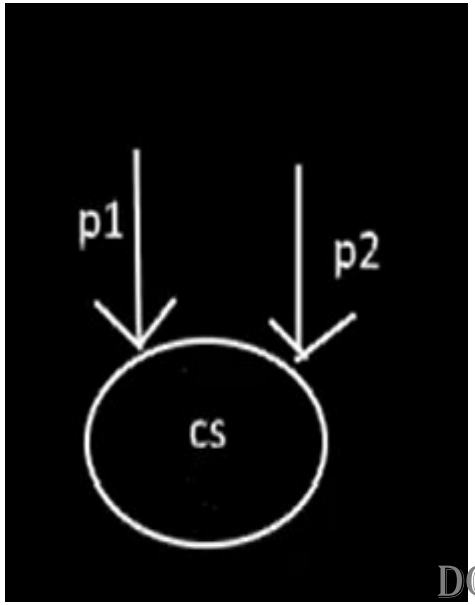
A solution to the critical-section problem must satisfy the following three requirements:

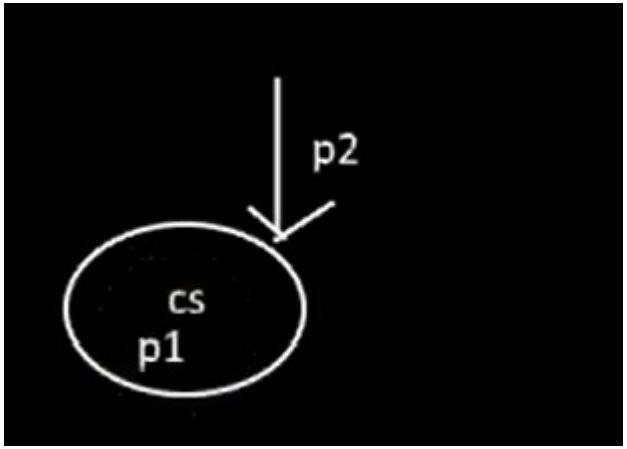
1. **Mutual exclusion.** If process *Pi is executing in its critical section, then no* other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after process has made a request to enter its critical section and before that request is granted.

1.Mutual Exclusion









Critical-Section Handling in OS

Two approaches depending on if kernel is: preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's solution

- A classic software-based solution to the critical-section problem known as **Peterson's solution.**
- **Because of the way modern computer** architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures.
- However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. For convenience, lets call the processes Pi and Pj.

Peterson's solution requires the two processes to share two data items:

int turn;

boolean flag[2];

The variable **turn** indicates **whose turn it is to enter its critical section**.

That is, if turn == i, then process *Pi is allowed to execute in its critical section.*

The **flag array** is used to indicate if a process is ready to enter its critical section.

For example, if **flag[i] is true**, this value indicates that ***Pi is ready to enter* its critical section.**

The structure of process Pi in Peterson's solution.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Process Pi

```
do {

flag[i] = true;
turn = j;
while (flag[j] && turn == j);

critical section

flag[i] = false;

remainder section

}while (true);
```

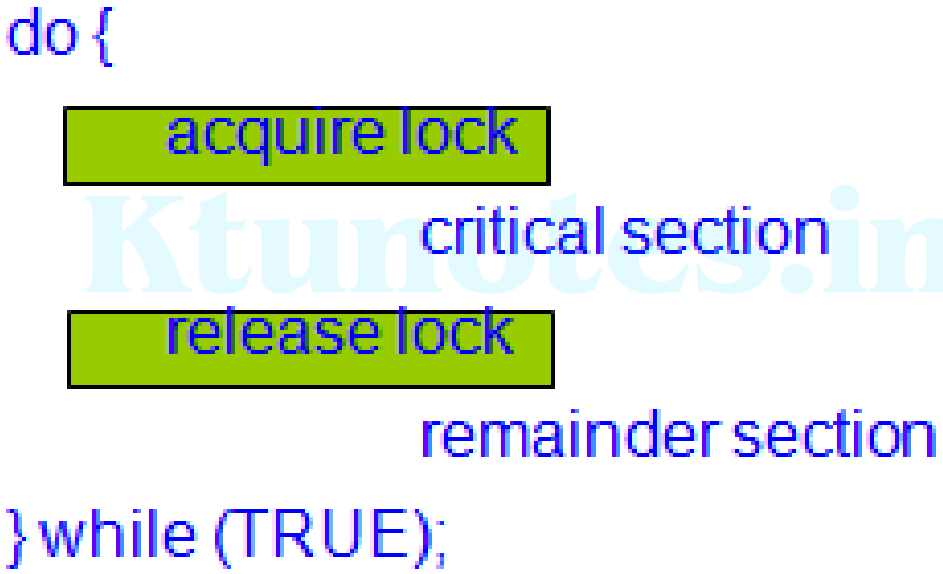
Process Pj

```
do {

flag[j] = true;
turn = i;
while (flag[i] && turn == i);

critical section

flag[j] = false;

remainder section

}while (true);
```

Synchronization Hardware

- Problems of Critical Section are also solvable by hardware .
- Uniprocessor systems disables interrupts while a Process Pi is using the CS but it is a great disadvantage in multiprocessor systems
- Some systems provide a lock functionality where a Process acquires a lock while entering the CS and releases the lock after leaving it. Thus another process trying to enter CS cannot enter as the entry is locked. It can only do so if it is free by acquiring the lock itself.
- Another advanced approach is the Atomic Instructions (Non-Interruptible instructions)

Solution to Critical-section Problem Using Locks



Solution using TestAndSet

- Shared Boolean variable lock, initialized to false.
- Solution:

```
do {
        while ( TestAndSet (&lock )) ;
                // do nothing
                //    critical section
        lock = FALSE;
                //      remainder section
} while (TRUE);
```

TestAndSet Instruction

- Definition:

```
boolean  TestAndSet (boolean *target)
{
    boolean  rv = *target;
    *target = TRUE;
    return  rv;
}
```

Must be executed atomically

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```


1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap

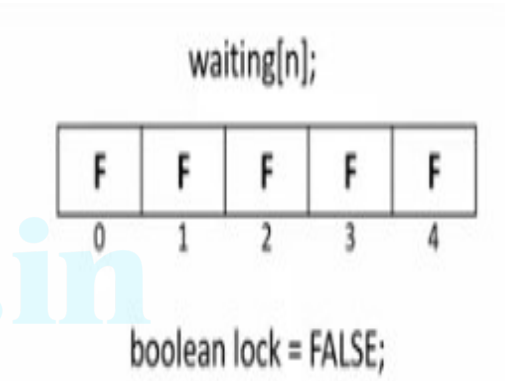
Shared integer "lock" initialized to 0;

Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
     ; /* do nothing */
   /* critical section */
 lock = 0;
   /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```



Mutex Locks

- Used to protect critical regions and thus prevent race conditions.
- A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire()function acquires the lock, and the release() function releases the lock.


```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (true);
```

Solution to the critical-section problem using mutex locks.

Mutex Locks

- The definition of acquire() is as follows:

```
acquire()
  {
     while (!available); /* busy wait */
      available = false;
   }
```

- The definition of release() is as follows:

```
release()
   {
      available = true;
   }
```

Mutex Locks

- Calls to either acquire() or release() must be performed atomically.

Disadvantage

- Busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- This type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.

Semaphores

- Semaphore is a Synchronization tool.
- Semaphore are used to solve busy waiting issue.
- A Semaphore is indicated by an integer variable S ;Process are accessed in Mutually exclusive manner.
- A semaphore S is an integer variable that, is accessed only through two standard **atomic operations: wait() and signal().**
- The **wait()** operation was originally termed P (from the Dutch *proberen, "to* test").
- **signal()** was originally called V (from *verhogen, "to increment").*

Semaphores cont..

1. Wait: It is used to test the semaphore variable. It decrement the semaphore value. If the value become negative, then the execution of wait() operation is blocked.

2. Signal: It increment the semaphore value.

- **Entry to the critical section is controlled by wait operation and exit from a critical section is taken care by signal operation.**

Semaphores cont..

The definition of wait() is as follows:

```
wait(S)
  {
    while (S <= 0); // busy wait or Do Nothing
operation
      S--; (or)  S=S-1;
  }
```

The definition of signal() is as follows:

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

```
do{
        Wait(S)                         Entry section
        Critical Section
        Signal(S)               Exit Section
        Remainder Section
        } while(True);
```

Eg: Initially S =0 (indefinite wait)

Later S=1

Semaphores

- when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

- In wait(S), the testing of the integer value of S ($S \leq 0$), and (S--), must be executed without interruption.

Semaphore Usage

Two types of Semaphores

Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.

Binary Semaphore

- The value of a binary semaphore can range only between 0 and 1.
- Thus, binary semaphores behave similarly to mutex locks.

Semaphore Usage

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a **wait()** operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a **signal()** operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphore Usage

Semaphores can be used to solve various synchronization problems

- Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2.

- It requires that S2 be executed only after S1 has completed.

- Let us implement this scheme by letting P1 and P2 share a common semaphore synch, initialized to 0.

Semaphore Implementation

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows:
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
- Instead of 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 the CPU scheduler, which selects another process to execute.

Semaphore Implementation

- 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.
- The process is then placed in the ready queue

Semaphore Implementation

```
typedef struct
  {
     int value;
     struct process *list;
  } semaphore;
```


- Each semaphore has an integer value and a list of processes list.
- When a process must wait on a semaphore, it is added to the list of processes.
- A signal() operation removes one process from the list of waiting processes and awakens that process.

Semaphore Implementation

The wait() semaphore operation can be defined as

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

Semaphore Implementation

- The signal() semaphore operation can be defined as

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

Semaphore Implementation

- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P.
- These two operations are provided by the operating system as basic system calls.

Semaphore Implementation

- Semaphore values may be negative , whereas semaphore values are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB).
- Each semaphore contains an integer value and a pointer to a list of PCBs.

Semaphore Implementation

- We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time.

- We can solve it by simply disabling interrupts during the time the wait() and signal() operations are executing.

- This scheme works in a single-processor environment because, once interrupts are disabled, instructions from different processes cannot be interleaved.

- Only the currently running process executes until interrupts are reenabled and the scheduler can regain control

Semaphore Implementation

- Disabling interrupts on every processor can be a difficult task on a multiprocessor system.

- This scheme limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short (if properly coded, they should be no more than about ten instructions).

- Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time.

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.

- When such a state is reached, these processes are said to be **deadlocked.**

Deadlocks and Starvation

P_0	P_1
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 P0 executes wait(Q), it must wait until P1 executes signal(Q).
- Similarly,whenP1 executes wait(S), it must wait until P0 executes signal(S).
- Since these signal() operations cannot be executed, P0 and P1 are deadlocked

Classical problems of synchronization

The Bounded-Buffer Problem

- Consider two processes named as producer and consumer. A producer process produces information that is consumed by a consumer process.
- A shared buffer that can be filled by the producer and emptied by the consumer.
- A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized.
- Two types of buffers used are
  - 1. Unbounded buffer – Variable size buffer
  - 2. Bounded buffer – Fixed size buffer

The Bounded-Buffer Problem

- The producer and consumer processes share the following data structures:

```
int  n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

- We assume that the pool consists of **n buffers**, each capable of holding one item.
- The **mutex semaphore** provides mutual exclusion for accesses to the buffer pool and is initialized to the value **1**.
- The empty and full semaphores count the number of empty and full buffers.
- The semaphore **empty** is initialized to the value **n**; the semaphore **full** is initialized to the value **0**.

The Bounded-Buffer Problem

```
do {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
} while (true);
```

The structure of the producer process.

The Bounded-Buffer Problem

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

The structure of the consumer process.

```
#define BUFFER_SIZE 10
int in = 0, out = 0,  count = 0;
int buffer[BUFFER_SIZE];
```

Producer process

```
Void Producer()
{
int item;
while (true){
item = produceitem();
if(count == BUFFER_SIZE); /* do nothing */
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
count++;  }
```

Consumer process

```
Void consumer()
{
Int item;
while (true) {
if(count == 0); /* do nothing */
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
count--;
}
```

The Dining-Philosophers Problem

The Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks .

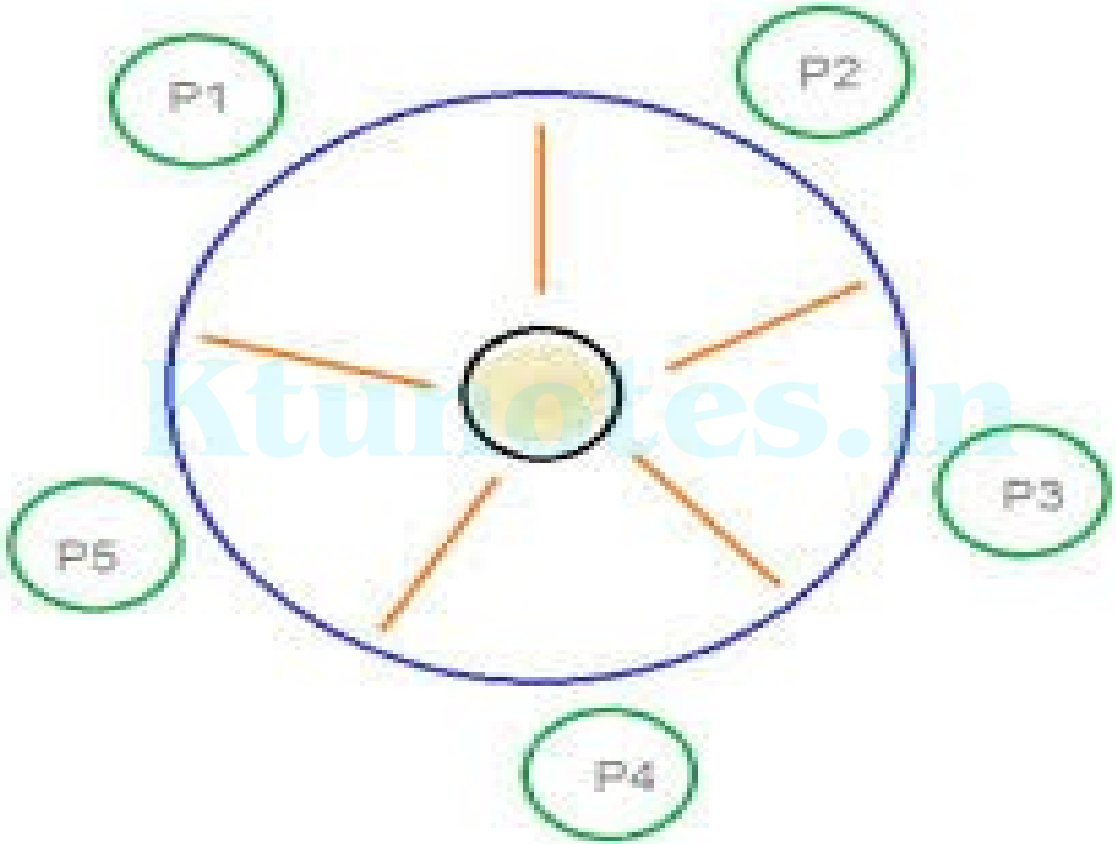
The situation of the dining philosophers.

The Dining-Philosophers Problem

- When a philosopher thinks, he/she does not interact with his/her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to his/her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time.
- Obviously, he/she cannot pick up a chopstick that is already in the hand of a neighbour.
- When a hungry philosopher has both his/her chopsticks at the same time, he/she eats without releasing the chopsticks.
- When he/she is finished eating, he/she puts down both chopsticks and starts thinking again.

The Dining-Philosophers Problem

- The dining-philosophers problem is considered a classic synchronization problem.
- It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- One simple solution is to represent each chopstick with a semaphore.
- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores.



Dining Philosophers Problem

The Dining-Philosophers Problem

- Thus, the shared data are

  semaphore chopstick[5];

where all the elements of chopstick are initialized to 1

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

The structure of philosopher *i*.

The Dining-Philosophers Problem

Deadlock

- Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
- All the elements of chopstick will now be equal to 0.
- When each philosopher tries to grab her right chopstick, she will be delayed forever.

Remedies to the deadlock problem :

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his/her chopsticks only if both chopsticks are available.
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first his/her left chopstick and then his/her right chopstick, whereas an even numbered philosopher picks up his/her right chopstick and then his/her left chopstick.

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do ***not*** perform any updates
  - **Writers** – can both read and write
- If two readers access the shared data simultaneously, no adverse effects will result.
- If a writer and some other process (either a reader or a writer) access the database simultaneously, adverse effects will result.
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

- The semaphores mutex and rw_mutex are initialized to 1; read count is initialized to 0.
- The semaphore rw_mutex is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable read_count is updated.
- The read count variable keeps track of how many processes are currently reading the object.

Readers-Writers Problem (Cont.)

The structure of a writer process

```
do {
    wait(rw_mutex);
      ...
    /* writing is performed */
      ...
   signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
    wait(mutex);
read_count++;

    if (read_count == 1)
   wait(rw_mutex);
signal(mutex);
    ...
    /* reading is performed */
    ...
wait(mutex);
    read count--;
    if (read_count == 0)
signal(rw_mutex);
signal(mutex);
} while (true);
```

Monitor

Monitors

- One of the fundamental high-level synchronization.

Monitor Usage

- A *monitor type is an ADT that includes a set of programmer defined* operations that are provided with mutual exclusion within the monitor.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Monitors

```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . . .
    }

    function P2 ( . . . ) {
        . . .
    }

            .
            .
            .
    function Pn ( . . . ) {
        . . .
    }

    initialization_code ( . . . ) {
        . . .
    }
}
```

Syntax of a monitor.

- Monitors can be used for achieving mutual exclusion: only one process can be active in a monitor at any instant.

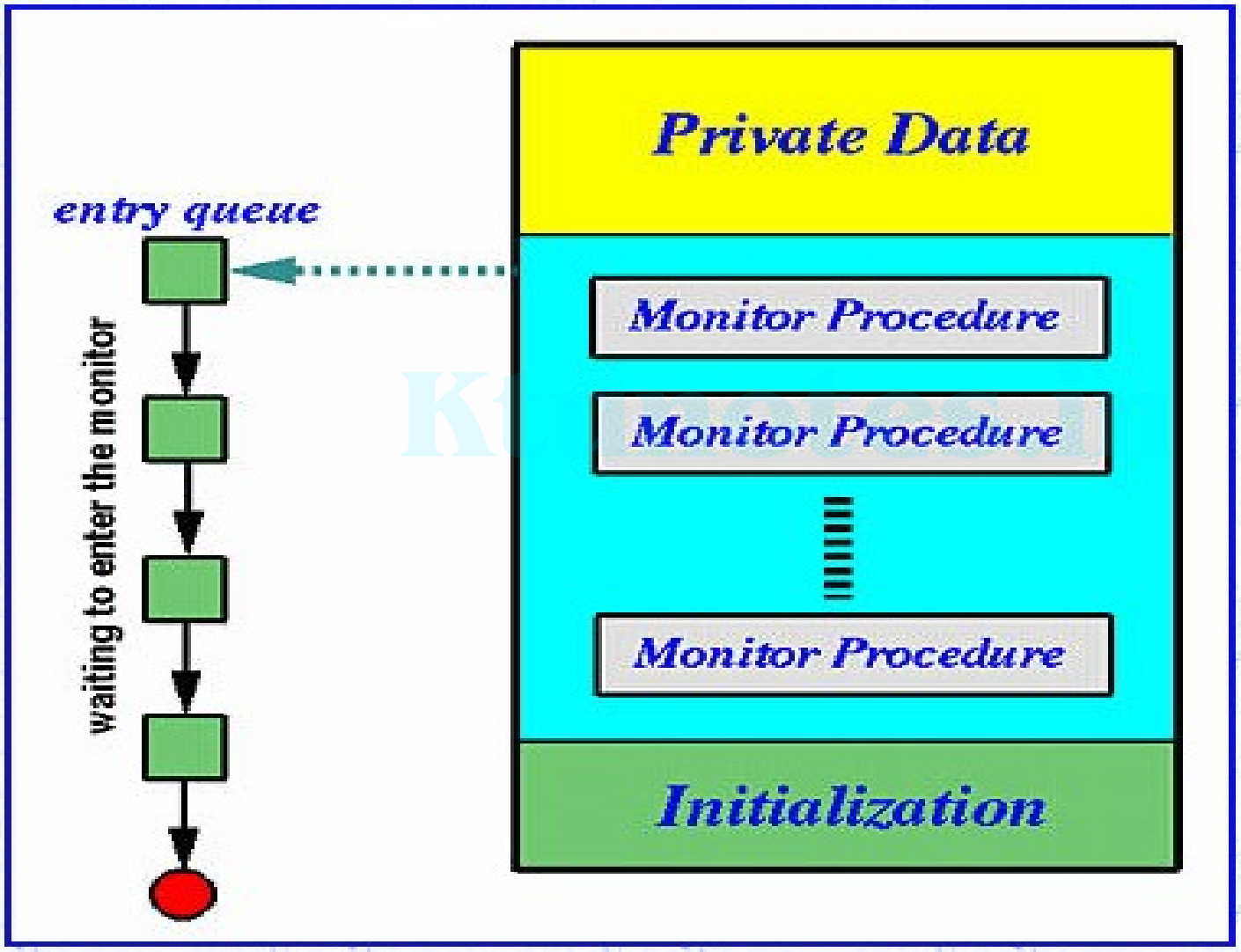
- Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls.

- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

Monitors

- A monitor has **four** components : initialization, private data, monitor procedures, and monitor entry queue.
- The **initialization** component contains the code that is used exactly once when the monitor is created.
- The **private data** section contains all private data, including private procedures, that can only be used within the monitor. These private items are not visible from outside of the monitor.
- The **monitor procedures** are procedures that can be called from outside of the monitor.
- The **monitor entry queue** contains all processes that called monitor procedures but have not been granted permissions.

Monitors



Mutual Exclusion of a Monitor

- Monitors are used in a multithreaded or multiprocess environment in which multiple threads/processes may call the monitor procedures at the same time asking for service.
- Thus, a monitor guarantees that ***at any moment at most one process can be executing in a monitor*** the calling process.
- If two processes are in the monitor (*i.e.*, they are executing two, possibly the same, monitor procedures), some private data may be modified by both processes at the same time causing race conditions to occur.

- Therefore, to guarantee the integrity of the private data, a monitor enforces mutual exclusion ***implicitly***.
- If a process calls a monitor procedure, this process will be blocked if there is another process executing in the monitor.
- Those process that were not granted the entering permission will be queued to a monitor ***entry queue*** outside of the monitor.
- When the monitor becomes empty (*i.e.*, no process is executing in it), one of the process in the entry queue will be released and granted the permission to execute the called monitor procedure.
- In summary, monitors ensure mutual exclusion automatically so that there is no more than one process can be executing in a monitor at any time. This is a very usable and handy capability.

- The monitor construct ensures that only one process at a time is active within the monitor.
- Monitor construct, is not sufficiently powerful for modeling some synchronization schemes. To avoid this problem condition construct is used
- condition x, y;
- 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.

- Now suppose that, when the x.signal() operation is invoked by a process *P, there exists a suspended process Q associated with condition x.*
- *If the* suspended process *Q is allowed to resume its execution, the signaling process P must wait.*
- Two possibilities exist:

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

Dining-Philosophers Solution Using Monitors

enum {THINKING, HUNGRY, EATING} state[5];

- 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).
- 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

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
           state[i] = EATING;
           self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

A monitor solution to the dining-philosopher problem.

- 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);
```

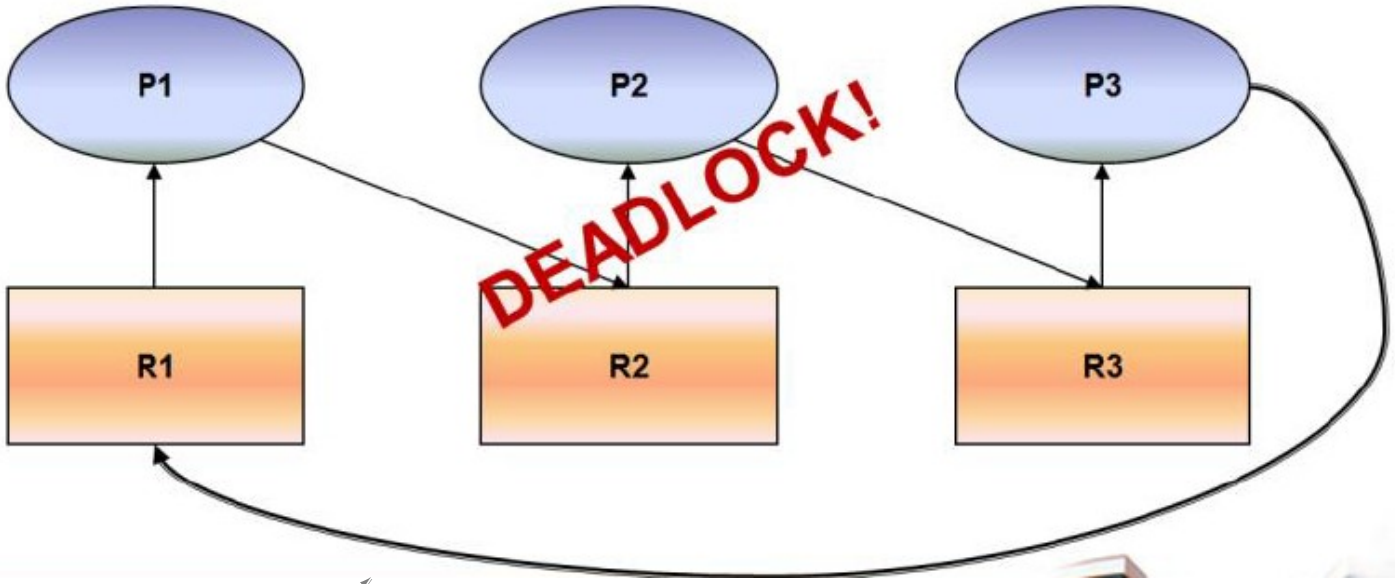
Deadlocks

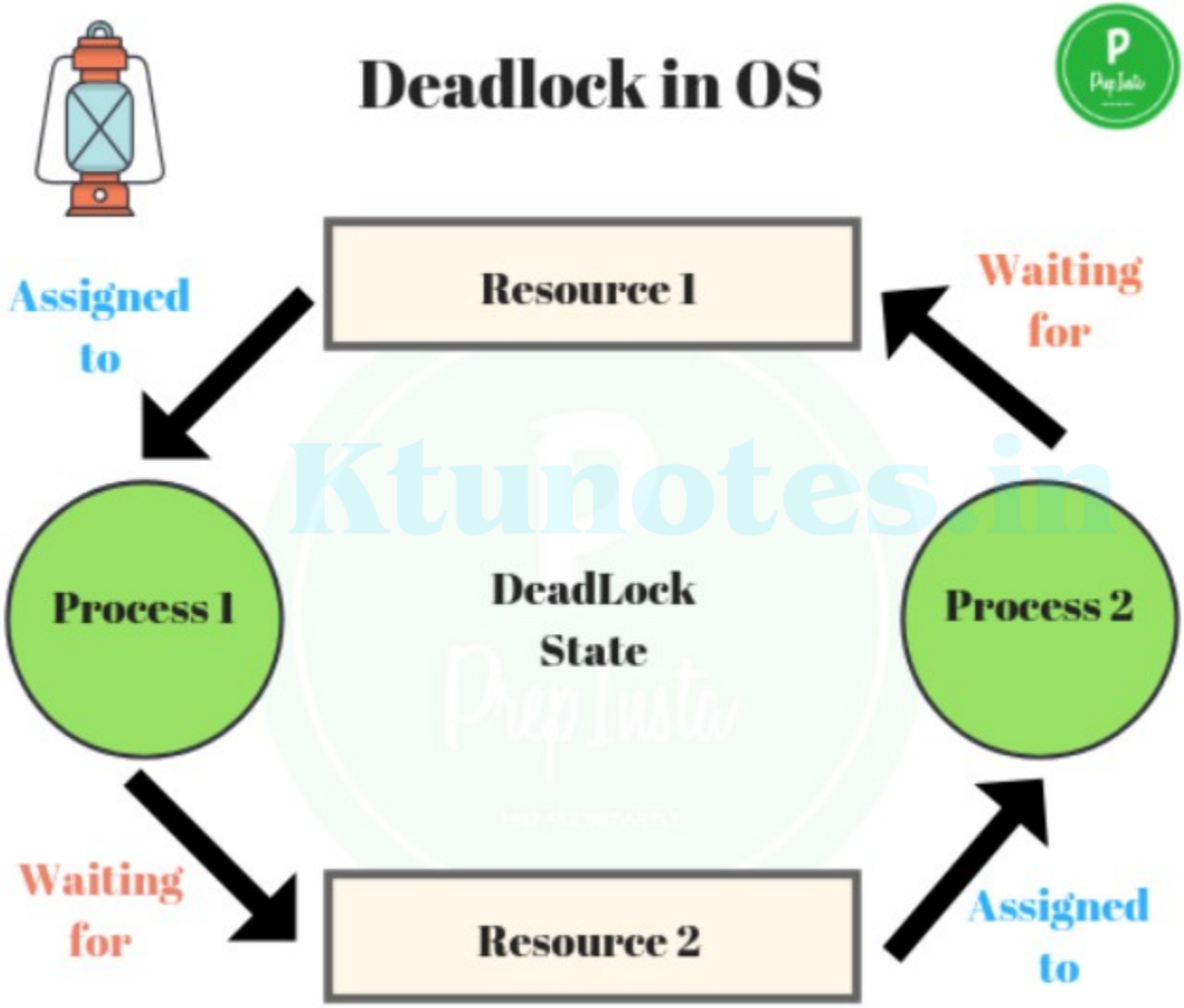
- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process **requests resources;** if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.
- Operating System work based on Deadlock(DL).
- Deadlock are rarely occur so no developer write code for DL.
  - Deadlock Ignorance: Windows and Linux.
  - Deadlock Prevention and Avoidance

Deadlock Scenario

- Process P1 holds resource R1 and requests resource R2
- Process P2 holds resource R2 and requests resource R3
- Process P3 holds resource R3

If P3 will request for resource R1 then What will happen?





But in deadlock situation, both the processes wait for the other process . Let's look at one example to understand it – Say, Process A has resource R1 , Process B has resource R2. If Process A request resource R2 and Process B requests resource R1, at the same time , then deadlock occurs.

Deadlock refers to the condition when 2 or more processes are waiting for each other to release a resource indefinitely. A process in nature requests a resource first and uses it and finally releases it.

DEADLOCKS

- Two processes each want to record a scanned document on a CD.
- Process *A requests permission to use the scanner and is granted it.*
- *Process B* is programmed in such a way that it requests the CD recorder first and is also granted it.
- Now *A asks for the CD recorder, but the request is denied until B releases it.*
- Instead of releasing the CD recorder *B asks for the scanner.*
- *At this* point both processes are blocked and will remain so forever. This situation is called a **deadlock.**

DEADLOCKS

- Deadlocks can occur on hardware resources or on software resources.
- Eg: In a database system, a program may have to lock several records it is using, to avoid race conditions. If process *A locks record R1 and process B locks record R2, and then each process tries to lock the other one's record, then there is a* a deadlock.

DEADLOCKS

RESOURCES

- A resource is anything that can be used by only a single process at any instant of time.
- A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database).
- Some resources, several identical instances may be available, such as three tape drives.
- When several copies of a resource are available, any one of them can be used to satisfy any request for the resource.

DEADLOCKS

Preemptable and Nonpreemptable Resources

- A preemptable resource is one that can be taken away from the process owning it with no ill effects.
- Eg: Memory
- A non preemptable resource, is one that cannot be taken away from its current owner without causing the computation to fail.
- Eg: CD Recorder
- If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD, CD recorders are not preemptable at an arbitrary moment.

DEADLOCKS

- Deadlocks involve nonpreemptable resources.
- Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another.
- The sequence of events required to use a resource
    1. Request the resource.
    2. Use the resource.
    3. Release the resource.

DEADLOCKS

DEADLOCKS

- *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*
- A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock.**

DEADLOCKS

Conditions for Deadlock

- A deadlock situation can arise if the following four conditions hold simultaneously in a system

1. Mutual exclusion condition.

2. Hold and wait condition.

3.No preemption condition..

4.Circular wait condition.

DEADLOCKS

1. **Mutual exclusion:** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

DEADLOCKS

3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait.:** A set *{P0, P1, ..., Pn} of waiting processes must exist such* that *P0 is waiting for a resource held by P1, P1 is waiting for a resource* held by *P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting* for a resource held by *P0.*

- **All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.**

Resource-Allocation Graph

- Deadlocks can be described precisely in terms of a directed graph called a **system resource-allocation graph.**

- This graph consists of a set of vertices *V* and a set of edges *E*.
- *The set of vertices V is partitioned into two different types* of nodes:
  - *P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the* system.
  - *R = {R1, R2, ..., Rm}, the set consisting of all resource types in the* system.

Resource-Allocation Graph

Request edge

- *A directed edge Pi → Rj is called a* **request edge.**
- It signifies that process *Pi has requested an instance of resource type Rj and* is currently waiting for that resource.

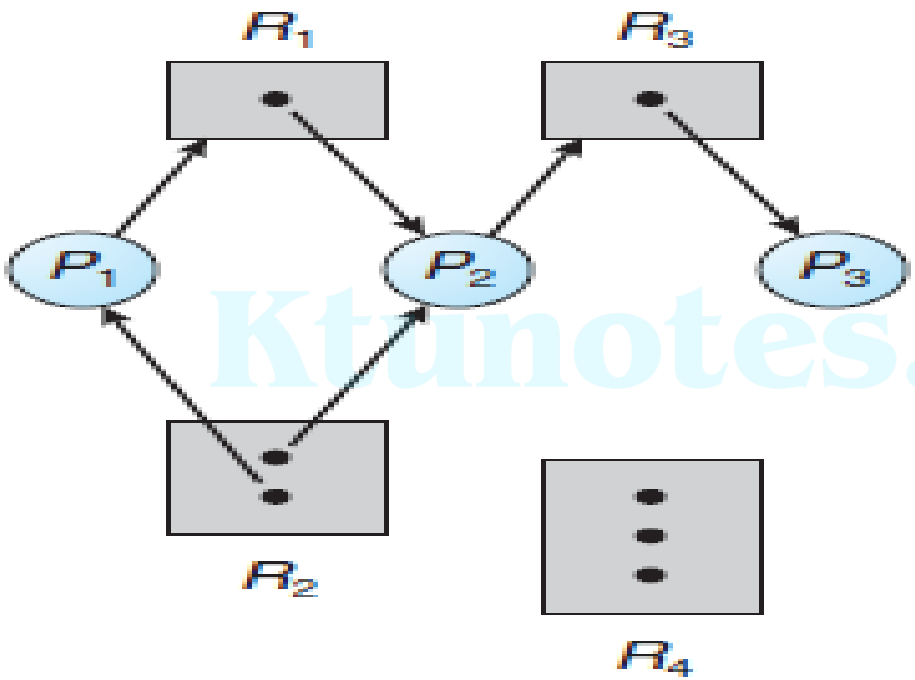
Assignment edge

- A directed edge *Rj → Pi is called an* ***assignment edge*** .
- *It signifies that an instance of resource* type *Rj has been allocated to process Pi.*

Resource-Allocation Graph

- Pictorially, we can represent each process *Pi as a* ***circle*** *and each resource type Rj as a* ***rectangle***.
- *Since resource type Rj may have more than one instance, we* represent each instance as a dot within the rectangle.
- When process *Pi requests an instance of resource type Rj, a request edge* is inserted in the resource-allocation graph.
- When this request can be fulfilled, the request edge is *transformed to an assignment edge.*
- *When* the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

Resource-Allocation Graph-Example



Resource-allocation graph.

Resource-Allocation Graph-Example

- **The sets *P, R, and E:***

  $P = \{P1, P2, P3\}$

  $R = \{R1, R2, R3, R4\}$

  $E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$

- **Resource instances:**
  - One instance of resource type *R1*
  - Two instances of resource type *R2*
  - One instance of resource type *R3*
  - Three instances of resource type *R4*

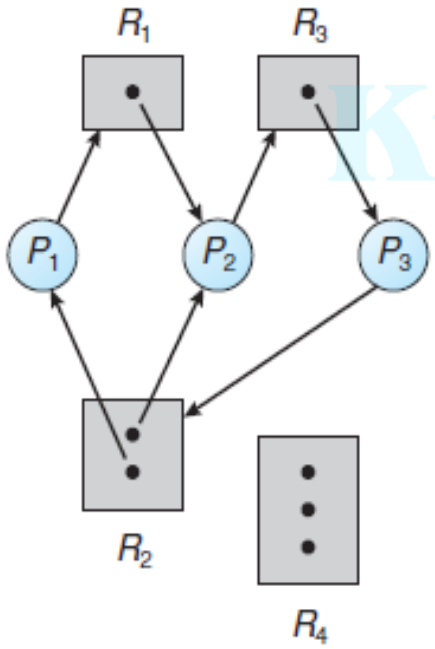
- **Process states:**
  - Process *P1 is holding an instance of resource type R2 and is waiting for* an instance of resource type *R1.*
  - Process *P2 is holding an instance of R1 and an instance of R2 and is* waiting for an instance of *R3.*
  - Process *P3 is holding an instance of R3.*

Resource-Allocation Graph

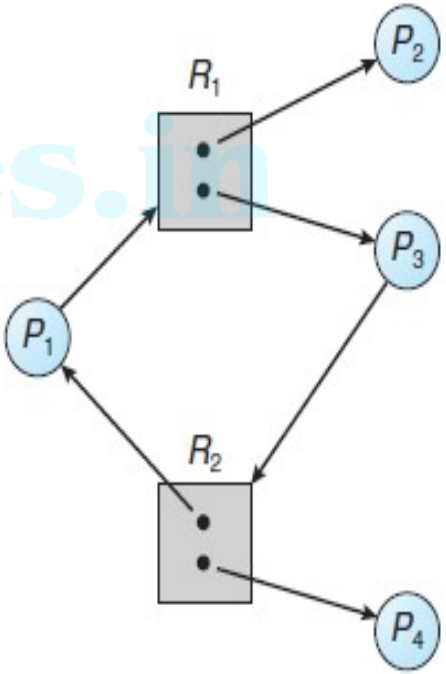
- If the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Resource-Allocation Graph

- If a resource-allocation graph does not have a cycle, then the system is ***not in a deadlocked state. If there is a cycle, then the system may or*** may not be in a deadlocked state.



Resource-allocation graph with a deadlock.



Resource-allocation graph with a cycle but no deadlock.

DEADLOCK HANDLING

Methods for Handling Deadlocks

- Deadlock handling can be done in one of three ways:

1. By using a protocol to prevent or avoid deadlocks, ensuring that the system will ***never enter a deadlocked state.***
2. Allow the system to enter a deadlocked state, detect it, and recover.
3. Ignore the problem altogether and pretend that deadlocks never occur in the system.

Methods for Handling Deadlocks

1)Prevent or avoid Deadlock

- To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme.
- **Deadlock prevention provides a** set of methods to ensure that at least one of the necessary conditions for deadlock cannot hold.
- **Deadlock avoidance requires that the operating system be given additional** information in advance concerning which resources a process will request and use during its lifetime.
- With this additional knowledge, the operating system can decide for each request whether or not the process should wait.

Methods for Handling Deadlocks

2) Detect and recover from deadlock.

- The system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

3)Ignore the problem altogether and pretend that deadlocks never occur in the system.

- In the absence of algorithms to detect and recover from deadlocks, system is in a deadlocked state yet has no way of recognizing what has happened.
- Eventually, the system will stop functioning and will need to be restarted manually.

Deadlock Avoidance

Deadlock Avoidance -Safe state

- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.

- A state is *safe if the system can allocate resources to each process (up to its* maximum) in some order and still avoid a deadlock.
- A system is in a safe state only if there exists a **safe sequence.**

- A sequence of processes *<P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently* available resources plus the resources held by all *Pj, with j <i.*

Safe state

- If the resources that *Pi needs are not immediately available, then Pi can wait* until all *Pj have finished.*

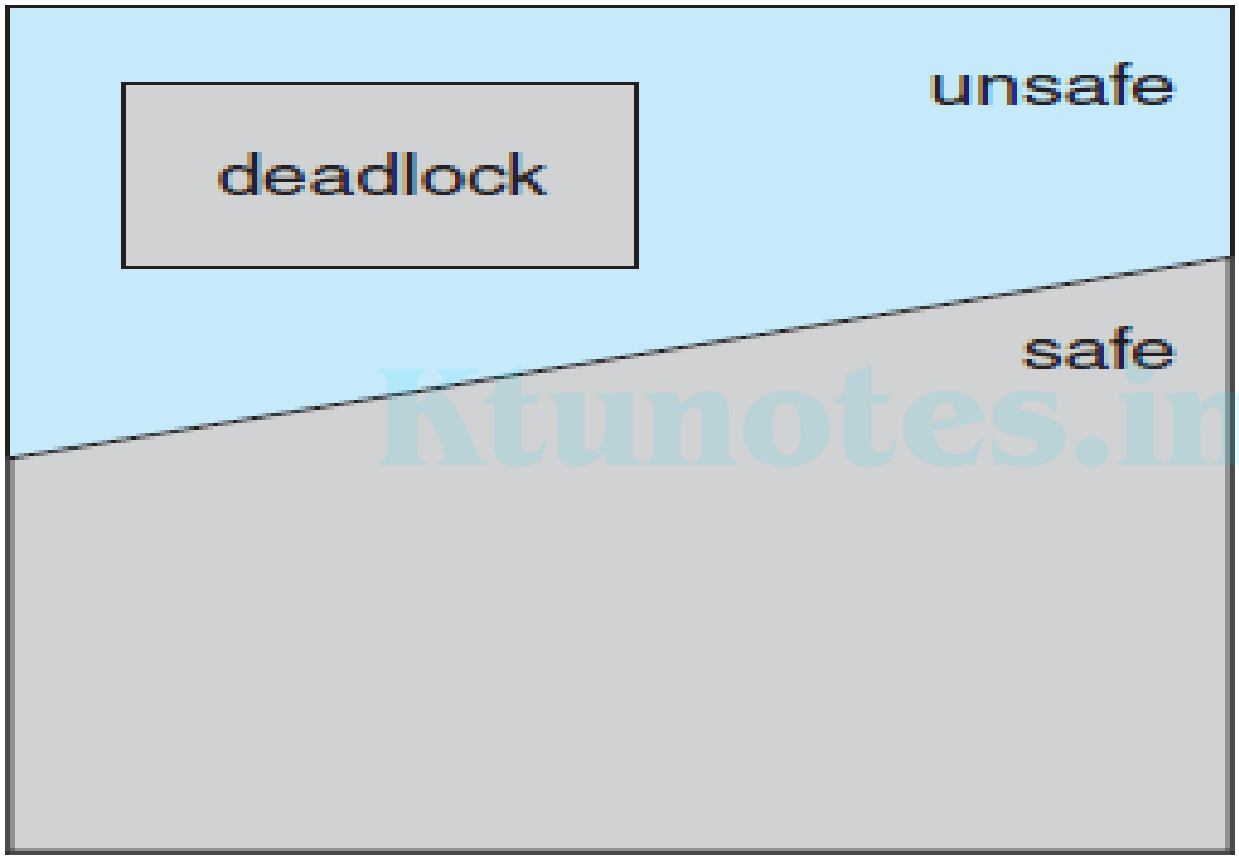
- *When they have finished, Pi can obtain all of its* needed resources, complete its designated task, return its allocated resources, and terminate.

- When *Pi terminates, Pi+1 can obtain its needed resources, and* so on. If no such sequence exists, then the system state is said to be *unsafe.*

Safe state

- A safe state is not a deadlocked state.
- A deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks.
- An unsafe state *may lead to a deadlock.*
- *As long as the state is safe, the* operating system can avoid unsafe (and deadlocked) states.
- In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs.

Safe state



Safe, unsafe, and deadlocked state spaces.

Safe state

- Consider a system with twelve magnetic tape drives and three processes: *P0, P1, and P2.*
- *Process P0 requires ten tape drives, process P1* may need four tape drives, and process *P2 may need up to nine tape* drives.
- Suppose that, at time *t0, process P0 is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus,* there are three free tape)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

Safe state

- At time *t0, the system is in a safe state. The sequence <P1, P0, P2> satisfies* the safety condition.

- Process *P1 can immediately be allocated all its tape drives* and then return them (the system will then have five available tape drives).

- Then process *P0 can get all its tape drives and return them (the system will then* have ten available tape drives).

- Finally process *P2 can get all its tape drives* and return them (the system will then have all twelve tape drives available).

Deadlock Avoidance Algorithms

1)Resource-Allocation-Graph Algorithm

- Used for single instance of each resource types

2)Banker's Algorithm

- Used for system with multiple instances of each resource type.

Resource-Allocation-Graph Algorithm

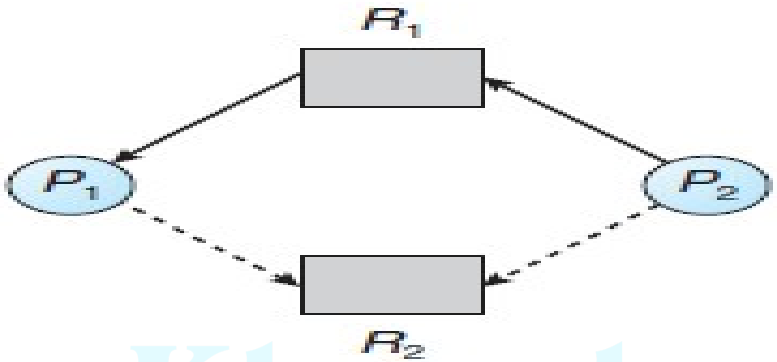
Claim edge

- A claim edge *Pi → Rj indicates that process Pi may request resource Rj at* some time in the future.
- This edge is represented in the graph by a dashed line.
- When process *Pi requests resource Rj, the claim edge Pi → Rj is converted to a request edge.*
- *Similarly, when a* resource *Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a* claim edge *Pi → Rj* .

Resource-Allocation-Graph Algorithm

- Suppose that process *Pi requests resource Rj*.
- *The request can be* granted only if converting the request edge *Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation* graph.
- We check for safety by using a cycle-detection algorithm.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state

Resource-Allocation-Graph Algorithm_Example



Resource-allocation graph for deadlock avoidance.

- Suppose that *P2 requests R2.*
- *Although R2 is currently free, we* cannot allocate it to *P2, since this action will create a cycle in the graph.*
- A cycle, indicates that the system is in an unsafe state.
- If *P1* requests *R2, and P2 requests R1, then a deadlock will occur.*

Banker's Algorithm

- **The name was chosen because** the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Banker's Algorithm

- Let *n is the number of* processes in the system and *m is the number of resource types*.

Data structures used d to implement the banker's algorithm

- **Available**. A vector of length m indicates the number of available resources of each type. If $Available[j]$ equals k, then k instances of resource type R_j are available.
- **Max**. An $n \times m$ matrix defines the maximum demand of each process. If $Max[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j.
- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that $Need[i][j]$ equals $Max[i][j] - Allocation[i][j]$.

Banker's Algorithm

Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == true$ for all i, then the system is in a safe state.

Banker's Algorithm

Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.

Let $Request_i$ be the request vector for process P_i. If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi is allocated its resources*.
- *If the new state* is unsafe, then *Pi must wait for* $Request_i$, *and the old resource-allocation* state is restored.

Banker's Algorithm-Example

To illustrate the use of the banker's algorithm, consider a system with five processes P_0 through P_4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- Suppose now that process *P1 requests one additional instance of resource type A and two instances of* resource type *C, so Request1 = (1,0,2). Check whether this request can be granted* immediately ?

Input:

Total Resources	R1	R2	R3
	10	5	7

Process	Allocation			Max		
	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3
P2	2	0	0	3	2	2
P3	3	0	2	9	0	2
P4	2	1	1	2	2	2

Explanation:

Total resources are R1 = 10, R2 = 5, R3 = 7 and allocated resources are R1 = (0+2+3+2 =) 7, R2 = (1+0+0+1 =) 2, R3 = (0+0+2+1 =) 3. Therefore, remaining resources are R1 = (10 – 7 =) 3, R2 = (5 – 2 =) 3, R3 = (7 – 3 =) 4.

Remaining available = Total resources – allocated resources

and

Remaining need = max – allocated

Process	Max			Allocation			Available			Needed		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	7	5	3	0	1	0	3	3	4	7	4	3
P2	3	2	2	2	0	0				1	2	2
P3	9	0	2	3	0	2				6	0	0
P4	2	2	2	2	1	1				0	1	1
				7	2	3						

So, we can start from either P2 or P4. We can not satisfy remaining need from available resources of either P1 or P3 in first or second attempt step of Banker's algorithm. There are only four possible safe sequences. These are :

P2–> P4–> P1–> P3

P2–> P4–> P3–> P1

Safe state

- A system can go from a safe state to an unsafe state.
- Suppose that, at time *t1, process P2 requests and is allocated one more tape drive. The system is no* longer in a safe state.
- At this point, only process *P1 can be allocated all its tape* drives.
- When it returns them, the system will have only four available tape drives. Since process *P0 is allocated five tape drives but has a maximum of ten,* it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process *P2 may request six additional tape* drives and have to wait, resulting in a deadlock.
- *If we had made P2 wait* until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Safe state

- By using the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state.
- Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.

DEADLOCK DETECTION

Deadlock Detection

Two Algorithms:

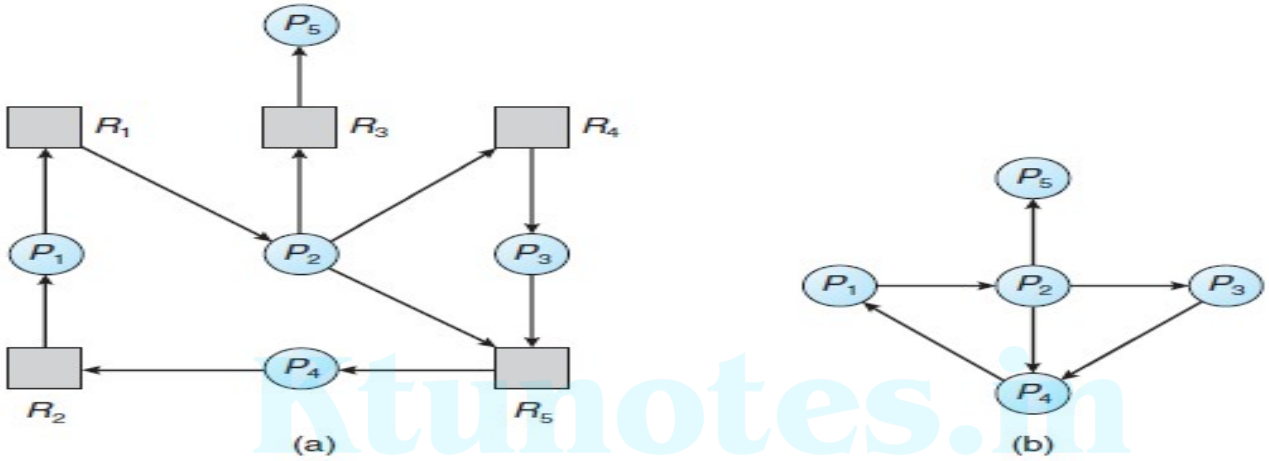
1)Algorithm for Single Instance of Each Resource Type

2) Algorithm for Several Instances of a Resource Type

Deadlock Detection-Single Instance of each Resource Type

- Deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- Wait for graph can be obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from *Pi to Pj in a wait-for graph implies that* process *Pi is waiting for process Pj to release a resource that Pi needs.*
- *An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation* graph contains two edges *Pi → Rq and Rq → Pj for some resource Rq* .

Deadlock Detection-Single Instances of each Resource type



(a) Resource-allocation graph. (b) Corresponding wait-for graph.

- A deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to ***maintain the wait for*** graph and periodically ***invoke an algorithm that searches for a cycle in*** the graph.

Deadlock Detection-Several Instances of a Resource Type

Data structures used

- **Available.** A vector of length m indicates the number of available resources of each type.
- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request.** An $n \times m$ matrix indicates the current request of each process. If $Request[i][j]$ equals k, then process P_i is requesting k more instances of resource type R_j.

Deadlock Detection-Several Instances of a Resource Type

Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n-1$, 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. $Request_i \leq Work$

   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, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

Deadlock Detection-Example

- Consider a system with five processes *P0* through *P4 and three resource types A, B, and C. Resource type A has seven* instances, resource type *B has two instances, and resource type C has six* instances. Suppose that, at time *T0, we have the following resource-allocation* state: **Detect deadlock.**

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Deadlock Detection-Example

Suppose now that process *P2 makes one additional request for an instance* of type *C, will the system be in deadlock state?*

Request

A B C

P_0 0 0 0

P_1 2 0 2

P_2 0 0 1

P_3 1 0 0

P_4 0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

DEADLOCK RECOVERY

Recovery from Deadlock

- There are two options for breaking a deadlock.

1. Abort one or more processes to break the circular wait.
2. Preempt some resources from one or more of the deadlocked processes.

Recovery from Deadlock

Process Termination

Abort all deadlocked processes.

- The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and will have to be recomputed later.

Abort one process at a time until the deadlock cycle is eliminated

- After each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked

Recovery from Deadlock

- Processes to be aborted are selected based on following.

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Recovery from Deadlock

Resource Preemption

- To eliminate deadlocks using resource preemption, successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Three issues need to be addressed:

1. **Selecting a victim**

- Which resources and which processes are to be preempted? we must determine the order of preemption to minimize cost.

2. **Rollback.**

- After preemption of resources the process must rollback ie. abort the process and then restart it.

Recovery from Deadlock

3. **Starvation.**

- We must ensure that starvation will not occur. That means resources will not always be preempted from the same process.
- We must ensure that a process can be picked as a victim only a (small) finite number of times.