<u>OPERATING SYSTEM PROJECT</u>

CONTENT :

1. *write a C program to display resource allocation graph with single instance for deadloack detection.*

2. *write a c program for resource allocation graph (wait for graph) to detect deadlock in system*

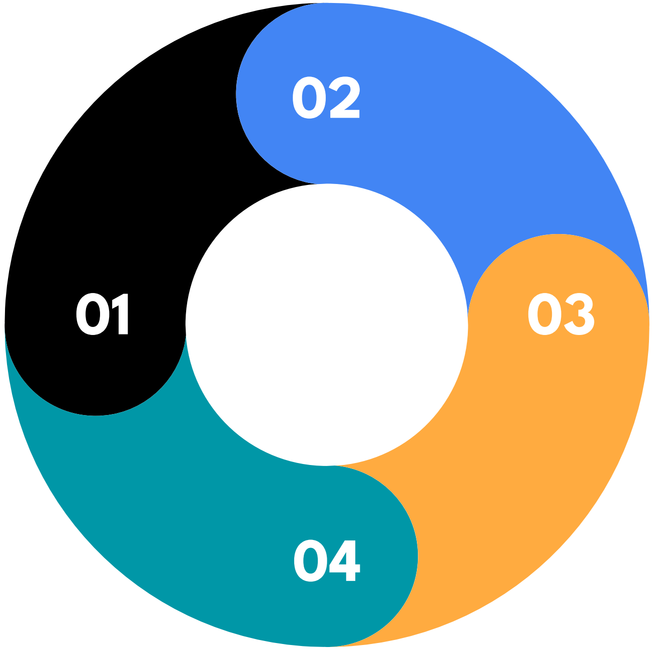
3. *write a c program to avoid deadlock using safety algorithm for single instance. Assume there are 4 processes and 16 resources types. give the safe sequence.*

P.no	Max need	Current need
P0	9	5
P1	4	3
P2	9	1
P3	6	2

Give the safe sequence.

Resource Allocation Graph

This presentation provides an overview of resource allocation graphs and how they can be used to detect deadlocks in a system.



01 *A resource allocation graph is a graphical representation of resource allocation and resource request relationships in a system.*

02 *Vertices represent processes and resources, while edges represent allocation and request relationships.*

03 *Deadlocks occur when there is a cycle in the resource allocation graph.*

04 *It is used to analyze and detect deadlocks in a system by examining the allocation and request edges between processes and resources.*

write a C program to display resource allocation graph with single instance for deadloack detection.

```
#include <stdio.h>

int main() {
  int available = 3;
  int max[]={7,3};
  int allocation[] = {0,0};
  int need[]= {7,3};

  int request[]={2,2}; //process request resource
  for{int i =0; i<2;i++}{  //check if request granted
    if(request[i] <=need[i] && request[i]<=available){
      available -= request[i];
      allocation += request[i];
      need[i] -= request[i];
    }
    else{
      printf("Request is denied.process is not in a
safe state");
      return 1;
    }
 }
    //check for safety

    if(need[0]== 0 && need[1] ==0){
      printf("Request granted.System is in safe
state.");
    }
else{
      printf("Request granted.System is not in
safe state.")
    }
  }
  return 0;
}
```

write a c program for resource allocation graph (wait for graph) to detect deadlock in system

Detecting deadlocks using a resource allocation graph (wait-for graph) is a common method. In this approach, you represent processes and resources as nodes in a graph, and edges indicate the allocation and request relationships between processes and resources. Deadlocks can be detected by finding cycles in this graph. If there is a cycle, it implies a potential deadlock. Here's a C program to detect deadlock using a resource allocation graph:

```
#include<stdio.h>
#inlude<stdbool.h>

#define NUM_PROCESSES 3
#define NUM_RESOURCES 3

int allocation[NUM_PROCESSES][NUM_RESOURCES];
int max_demand[NUM_PROCESSES][NUM_RESOURCES];
int available[NUM_RESOURCES];

bool isDeadlock{
for(int i=0;i<NUM_PROCESSES;i++){
    bool we_allocate = false;
      for(int j=0;j<NUM_RESOURCES;j++){
          if(max_demand[i][j] - allocation[i][j]<= available[j]{
                we_allocatate = true;
                break;
                }
        }
```

```
if(!we_allocate){
        return true;  //deadlock detected
    }
  }
return false; //no deadloack
}

int main(){
     if(isDeadlock){
          printf("deadlock detected");
}
else{
        printf("No deadlock");
}
return 0;
}
```

write a c program to avoid deadlock using safety algorithm for single instance. Assume there are 4 processes and 16 resources types. Resource allocation table is given below.

P.no	Max need	Current need
P0	9	5
P1	4	3
P2	9	1
P3	6	2

Give the safe sequence.

```c
#include <stdio.h>

#define NUM_PROCESSES 4
#define NUM_RESOURCES 16

int max_need[NUM_PROCESSES][NUM_RESOURCES] = {
        {9, 1, 0, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0}
};
int current_need[NUM_PROCESSES][NUM_RESOURCES] = {
        {5, 1, 0, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0}
};
int available_resources[NUM_RESOURCES] = {
        3, 1, 0, 2, 1, 0, 0, 0, 2, 1, 0, 0, 1, 0, 1, 0
      };

      int is_safe(int process, int work[], int finish[]) {
        for (int i = 0; i < NUM_RESOURCES; i++) {
          if (current_need[process][i] > work[i]) {
             return 0;
    }
  }
  return 1;
}
int find_safe_sequence(int safe_sequence[]) {
  int work[NUM_RESOURCES];
  int finish[NUM_PROCESSES];

  for (int i = 0; i < NUM_RESOURCES; i++) {
      work[i] = available_resources[i];
  }
for (int i = 0; i < NUM_PROCESSES; i++) {
```

```c
int count = 0;
  while (count < NUM_PROCESSES) {
    int found = 0;
    for (int i = 0; i < NUM_PROCESSES; i++) {
      if (finish[i] == 0 && is_safe(i, work, finish)) {
        for (int j = 0; j < NUM_RESOURCES; j++) {
          work[j] += current_need[i][j];
        }
        safe_sequence[count++] = i;
        finish[i] = 1;
        found = 1;
      }
    }
    if (!found) {
      return 0;  // No safe sequence found
    }
  }
  return 1;  // Safe sequence found
}
int main() {
  int safe_sequence[NUM_PROCESSES];
  if (find_safe_sequence(safe_sequence)) {
    printf("Safe Sequence: ");
    for (int i = 0; i < NUM_PROCESSES; i++) {
      printf("P%d ", safe_sequence[i]);
    }
    printf("\n");
  } else {
    printf("No safe sequence found.\n");
  }
  return 0;
}
```

Conclusion

Resource allocation graphs and safety algorithms are essential tools in preventing and detecting deadlocks in systems.

By understanding how resource allocation graphs work and implementing safety algorithms, developers can ensure efficient and deadlock-free system resource management.

It is crucial to regularly analyze resource allocation and request relationships to avoid potential deadlocks.

Thank you for your time 😊