OS: Tasks

OS: Page Replacement Task

OS: CPU Scheduling Task

OS: Deadlock Task

3rd Computer Engineering, Helwan University

1- (030) ايه معوض راشد
2- (043) ريهام محمد ابو اليزيد
3- (038) حسن ابراهيم فتوح
4- (097) مصطفى مجدى احمد عثمان
5- (105) هبة اشرف فؤاد طه

```
/**************************************************************************
                            OS: Page Replacement Task
                     3rd Computer Engineering,Helwan University
        1- (030) ايه معوض راشد
        2- (043) ريهام محمد ابو اليزيد
        3- (038) حسن ابراهيم فتوح
        4- (097) مصطفى مجدى احمد عثمان
        5- (105) هبة اشرف فؤاد طه

*************************************************************************/
#include <iostream> /* Input & Output  */
#include <stdlib.h> /* Standard Library */

using namespace std;
/*************************************************************************/

  /***********************************************************************/
 /****************************Functions Declarations**********************/
/***********************************************************************/
void Page_Replacement(void);
int getReplaceposition(int counter[], int n);
void FIFO(int pages[], int nPages, int nFrames);
void LFU(int arr[], int nPages, int nFrames);
int min(int counter[], int nFrames);
void LRU(int arr[], int ArraySize, int NFrames);
void MFU(int arr[], int nPages, int nFrames);
void Optimal(int Pages[], int NPages, int NFrames);
void SecondChance(int Pages[], int NPages, int NFrames);
/*************************************************************************/

  /***********************************************************************/
 /*********************************Main Function**************************/
/***********************************************************************/
int main() {
  Page_Replacement();
  return 0;
}
/*************************************************************************/

  /***********************************************************************/
 /*************************Functions Implementations**********************/
/***********************************************************************/
void Page_Replacement(void){
  // Entering seed number
  int seednumber;
  cout << "Enter seed number: ";
  cin >> seednumber;

  // Function to change time for Random function
  srand(seednumber);

  // Variables
  int NPages;    /* TO Set The Number Of Pages */
  int NFrames;   /* To Set The Number Of Frames */
  int Algorithm; /* To Chooses memory management algorithm*/

  // Taking The Numbers Of Pages and Frames
  cout << "Enter The Number of Pages :  ";
  cin >> NPages;
```

```
  cout << "Enter The Number of Frames :  ";
  cin >> NFrames;
  system("CLS"); /* Clearing the output screen */

  // Pages Array
  int *Pages = new int[NPages];

  // Pushing Random Numbers into the array from (1) to (10)
  cout << "Array : " << endl;
  for (int i = 0; i < NPages; i++) {
    // cin >> Pages[i];
    Pages[i] = rand() % 10 + 1;
    cout << Pages[i] << "   ";
  }

  // For Choosing a Number
  cout << endl << endl;
  cout << "1- First In First Out(FIFO)" << endl;
  cout << "2- Least Recently used(LRU)" << endl;
  cout << "3- Least Frequently used(LFU)" << endl;
  cout << "4- Most Frequently used(MFU)" << endl;
  cout << "5- Optimal " << endl;
  cout << "6- Second Chance" << endl << endl;
  cout << "Choose a Number :  ";

  // Taking an input
  cin >> Algorithm;
  //    system("CLS");      /* Clearing the output screen */

  // Checking the input for executing the certain function
  if (Algorithm == 1) {
    /* Headers/FIFO.h */
    FIFO(Pages, NPages, NFrames);
  } else if (Algorithm == 2) {
    /* Headers/LRU.h */
    LRU(Pages, NPages, NFrames);
  } else if (Algorithm == 3) {
    /* Headers/LFU.h */
    LFU(Pages, NPages, NFrames);
  } else if (Algorithm == 4) {
    /* Headers/MFU.h */
    MFU(Pages, NPages, NFrames);
  } else if (Algorithm == 5) {
    /* Headers/Optimal.h */
    Optimal(Pages, NPages, NFrames);
  } else if (Algorithm == 6) {
    /* Headers/SecondChance.h */
    SecondChance(Pages, NPages, NFrames);
  } else {
    cout << "Please Choose a valid Number" << endl;
  }
}
/**************************************FIFO************************************/
int getReplaceposition(int counter[], int n) {
  int max = counter[0];
  int pos = 0;
  for (int i = 0; i < n; i++) {
    if (counter[i] > max) {
      pos = i;
```

```
      max = counter[i];
    }
  }

  return pos;
}

// FIFO function
void FIFO(int pages[], int nPages, int nFrames) {
  // Complete this function

  int flag, hlt, Totalhlt = 0;
  int pageFault = 0;
  //    int *pages = new int[nPages];
  int *frames = new int[nFrames];
  int *counter = new int[nFrames];

  for (int i = 0; i < nFrames; i++) {
    frames[i] = 0;
    counter[i] = 0;  // here 0 referes an empty space in frame
  }

  for (int i = 0; i < nPages; i++) {
    flag = 0;
    hlt = 0;
    for (int j = 0; j < nFrames; j++) {
      if (frames[j] == pages[i]) {
        flag = 1;  // if page is present in frame (flag=1)
        hlt = 1;
        Totalhlt++;
        break;
      }
    }

    // if page is not present in frame (flag=0)
    if (flag == 0) {
      pageFault++;
      for (int j = 0; j < nFrames; j++) {
        if (frames[j] == 0) {
          frames[j] = pages[i];
          flag = 1;
          hlt = 0;
          counter[j]++;
          break;
        }
      }
    }

    // if there is no empty frame
    if (flag == 0) {
      int pos = getReplaceposition(counter, nFrames);
      frames[pos] = pages[i];
      counter[pos] = 1;
      for (int k = 0; k < nFrames; k++) {
        if (k != pos) counter[k]++;
      }
    }

    cout << endl;
```

```
    for (int j = 0; j < nFrames; j++) {
      if (hlt == 1) {
        cout << "  ";
      } else {
        cout << frames[j] << " ";
      }
    }
  }

  cout << "\nTotal Hlt: " << Totalhlt;
  cout << "\nTotal Miss: " << pageFault;
}
/*************************************LFU*************************************/
void LFU(int arr[], int nPages, int nFrames) {
  int p;
  bool done;
  int totalMiss = 0;
  int *frames = new int[nFrames]; /* array for frames */
  int *frequency =
      new int[nFrames]; /* array to check frequency for each page */
  int *check =
      new int[nPages]; /* array to be checked if page leave memory or not */
  int totalHlt = 0;

  // initialize frames as empty
  for (int i = 0; i < nFrames; i++) {
    frames[i] = -1;
  }
  // initialize all frequency with 0 for expected pages 1-10
  for (int i = 0; i < nFrames; i++) {
    frequency[i] = 0;
  }
  // initialize check bit for each page
  for (int i = 0; i < nPages; i++) {
    check[i] = -1;
  }

  for (int readyPage = 0; readyPage < nPages; readyPage++) {
    done = false;  // to check if page finds a frame
    for (int i = 0; i < nFrames; i++) {
      // check if page is already exist
      if (arr[readyPage] == frames[i]) {
        totalHlt++;
        // increase frequency of the page
        frequency[i]++;
        done = true;
        break;
      }
      // you find empty frame
      else if (frames[i] == -1) {
        totalMiss++;
        frames[i] = arr[readyPage];
        frequency[i]++;
        done = true;
        break;
      }
    }

    // you have to swap with another page
```

```
    if (done == false) {
      int least = frequency[0];    /* least as value */
      int leastFrequentlyUsed = 0; /* least as frame index */

      // find frequency of current pages in the memory
      for (int k = 0; k < nFrames; k++) {
        // you find the least
        if (frequency[k] < least) {
          least = frequency[k];
          leastFrequentlyUsed = k;
          p = k;
        }
        // you find more than one page has the same frequency
        else if (frequency[k] == least) {
          // check if the page leave the memory before
          for (int j = 0; j < readyPage; j++) {
            // find first in
            if (arr[j] == frames[leastFrequentlyUsed] && check[j] != 0) {
              p = j;  // save swapped page
              break;
            } else if (arr[j] == frames[k] && check[j] != 0) {
              least = frequency[k];
              leastFrequentlyUsed = k;
              p = j;  // save swapped page
              break;
            }
          }
        }
      }

      // swap with the least or first in
      frames[leastFrequentlyUsed] = arr[readyPage];
      done = true;
      frequency[leastFrequentlyUsed] = 1;
      check[p] = 0;  // page leaved memory
      totalMiss++;
    }
    for (int qq = 0; qq < nFrames; qq++) cout << frames[qq] << " ";
    cout << "\n";

    // end of if statment
  }  // end of for loop
  cout << "total miss: " << totalMiss << "\n";
  cout << "total HLT: " << totalHlt << "\n";

}  // end of function

// minimum Freq.
int min(int counter[], int nFrames) {
  int minimum = counter[0];
  int pos = 0;
  for (int i = 1; i < nFrames; i++) {
    if (minimum > counter[i]) {
      minimum = counter[i];
      pos = i;
    }
  }
  return pos;
}
```

```
/*********************************LRU**********************************/
void LRU(int arr[], int ArraySize, int NFrames) {
  // Complete this function
  int Frames[NFrames]; /* The Array Of Frames That We have */

  int counter[ArraySize], recent = 0;

  int pageFault = 0;

  int PageHLT = 0;
  for (int i = 0; i < NFrames; i++) {
    Frames[i] = 0;
    counter[i] = 0;  // here 0 referes an empty space in frame
  }
  for (int i = 0; i < ArraySize; i++)

  {
    int flag = 0, HLTflag = 0;
    for (int j = 0; j < NFrames; j++) {
      if (Frames[j] == arr[i]) {
        flag = 1;
        counter[j] = recent++;  // counter holds which frame is recently used,
        // recently used page in frame will have a bigger number
        // and least recently used page in frame will have a lower number
        HLTflag = 1;
        break;
      }
    }

    if (flag == 0) {
      for (int j = 0; j < NFrames; j++) {
        if (Frames[j] == 0) {
          Frames[j] = arr[i];
          counter[j] = recent++;
          flag = 1;
          pageFault++;
          break;
        }
      }
    }

    if (flag == 0) {
      int PositionToreplace = min(counter, NFrames);
      Frames[PositionToreplace] = arr[i];
      counter[PositionToreplace] = recent++;
      pageFault++;
    }

    // print frames
    cout << endl;
    for (int j = 0; j < NFrames; j++) {
      if (HLTflag == 1) {
        PageHLT++;
        break;
      }
      cout << Frames[j] << " ";
    }
  }
  cout << "\nNumber Of Page HLT: " << PageHLT;
```

```
  cout << "\nTotal Miss: " << pageFault;
}
/*************************************MFU**************************************/
void MFU(int arr[], int nPages, int nFrames) {
  int p;
  bool done;
  int totalMiss = 0;
  int *frames = new int[nFrames]; /* array for frames */
  int *frequency =
      new int[nFrames]; /* array to check frequency for each page */
  int *check =
      new int[nPages]; /* array to be checked if page leave memory or not */
  int totalHlt = 0;

  // initialize frames as empty
  for (int i = 0; i < nFrames; i++) {
    frames[i] = -1;
  }
  // initialize all frequency with 0 for expected pages 1-10
  for (int i = 0; i < nFrames; i++) {
    frequency[i] = 0;
  }
  // initialize check bit for each page
  for (int i = 0; i < nPages; i++) {
    check[i] = -1;
  }

  for (int readyPage = 0; readyPage < nPages; readyPage++) {
    done = false;  // to check if page finds a frame
    for (int i = 0; i < nFrames; i++) {
      // check if page is already exist
      if (arr[readyPage] == frames[i]) {
        totalHlt++;
        // increase frequency of the page
        frequency[i]++;
        done = true;
        break;
      }
      // you find empty frame
      else if (frames[i] == -1) {
        totalMiss++;
        frames[i] = arr[readyPage];
        frequency[i]++;
        done = true;
        break;
      }
    }

    // you have to swap with another page
    if (done == false) {
      int Most = frequency[0];    /* Most as value */
      int MostFrequentlyUsed = 0; /* Most as frame index */

      // find frequency of current pages in the memory
      for (int k = 0; k < nFrames; k++) {
        // you find the Most
        if (frequency[k] > Most) {
          Most = frequency[k];
          MostFrequentlyUsed = k;
```

```
          p = k;
        }
        // you find more than one page has the same frequency
        else if (frequency[k] == Most) {
          // check if the page leave the memory before
          for (int j = 0; j < readyPage; j++) {
            // find first in
            if (arr[j] == frames[MostFrequentlyUsed] && check[j] != 0) {
              p = j;  // save swapped page
              break;
            } else if (arr[j] == frames[k] && check[j] != 0) {
              Most = frequency[k];
              MostFrequentlyUsed = k;
              p = j;  // save swapped page
              break;
            }
          }
        }
      }

      // swap with the Most or first in
      frames[MostFrequentlyUsed] = arr[readyPage];
      done = true;
      frequency[MostFrequentlyUsed] = 1;
      check[p] = 0;  // page leaved memory
      totalMiss++;
    }
    for (int qq = 0; qq < nFrames; qq++) cout << frames[qq] << " ";
    cout << "\n";

    // end of if statment
  }  // end of for loop
  cout << "total miss: " << totalMiss << "\n";
  cout << "total HLT: " << totalHlt << "\n";

}  // end of function
/*********************************Optimal**********************************/
void Optimal(int Pages[], int NPages, int NFrames) {
  // Frames Array
  int *Frames;
  Frames = new int[NFrames];
  for (int i = 0; i < NFrames; i++) Frames[i] = -1;  // Empty Frame

  int TotalMiss = 0;  // Total Miss Counter

  // Loop on Pages
  for (int i = 0; i < NPages; i++) {
    bool isThereEmptyFrame = false;
    bool isPageAlreadyPresented = false;

    // Loop on Frames
    for (int j = 0; j < NFrames; j++) {
      // Check if the Page is aleardy presented
      if (Frames[j] == Pages[i]) {
        isPageAlreadyPresented = true;
        break;
      }

      // Check if there is Empty Frame
```

```
      else if (Frames[j] == -1) {
        TotalMiss++;
        Frames[j] = Pages[i];
        isThereEmptyFrame = true;
        break;
      }
    }  // End of Loop on Frames

    // Need to Replace
    if ((!isThereEmptyFrame) && (!isPageAlreadyPresented)) {
      TotalMiss++;
      int MaxDistance = 0;
      int Index = -1;

      // Loop on Frames
      for (int j = 0; j < NFrames; j++) {
        bool isPageUsedInFuture = false;

        // Loop on Future use Pages
        for (int k = i + 1; k < NPages; k++) {
          // is Page Used In Future
          if (Frames[j] == Pages[k]) {
            isPageUsedInFuture = true;

            if ((k - i) > MaxDistance) {
              MaxDistance = k - i;
              Index = j;
            }
            break;
          }
        }  // End Loop on Future use Pages

        if (!isPageUsedInFuture) {
          MaxDistance = NPages;  // The Biggest Value forever
          Index = j;
          break;
        }
      }  // End of Loop on Frames

      // Replace The Frame's Page
      Frames[Index] = Pages[i];
    }

    // Show Frames
    for (int j = 0; j < NFrames; j++) {
      cout << Frames[j] << " ";
    }
    cout << endl;

  }  // End of Loop on Pages

  // Show Tota Miss
  cout << "Total Miss = " << TotalMiss << endl;
}
/********************************Second Chance*********************************/
void SecondChance(int Pages[], int NPages, int NFrames) {
  int *frames = new int[NFrames];            /* array for frames */
  bool *secondChanceBit = new bool[NFrames]; /*SECOND CHANCE Bit */
  bool valid[10];
```

```
  int frame = 0; /* index of the next frame to add pages in */
  bool done;     /* check if page find frame */
  int totalMiss = 0;

  // initialize frames as empty
  for (int i = 0; i < NFrames; i++) {
    frames[i] = -1;
    secondChanceBit[i] = false;
  }
  // initialize all valid with 0 for expected pages 1-10
  for (int i = 0; i < 10; i++) valid[i] = false;
  for (int readyPage = 0; readyPage < NPages; readyPage++) {
    do {
      if (frames[frame] == -1 && valid[Pages[readyPage] - 1] == false) {
        cout << "first condition";

        frames[frame] = Pages[readyPage];
        valid[Pages[readyPage] - 1] = true;
        secondChanceBit[frame] = false;
        cout << "you are at frame " << frame;
        frame = (frame + 1) % NFrames;
      } else if (valid[Pages[readyPage] - 1] == true) {
        cout << "second condition";

        cout << "you are at frame " << frame;
        for (int i = 0; i < NFrames; i++) {
          if (Pages[readyPage] == frames[i]) secondChanceBit[i] = true;
        }
      } else if (secondChanceBit[frame] == true) {
        cout << "third condition"
             << "\n";
        cout << "you are at frame " << frame;
        secondChanceBit[frame] = false;
        frame = (frame + 1) % NFrames;

      } else if (secondChanceBit[frame] == false) {
        cout << "fourth condition";
        cout << "you are at frame " << frame;
        valid[frames[frame] - 1] = false;
        frames[frame] = Pages[readyPage];
        secondChanceBit[frame] = false;
        frame = (frame + 1) % NFrames;
        valid[Pages[readyPage] - 1] = true;
        totalMiss++;
      }

    } while (valid[Pages[readyPage] - 1] == false);

    cout << "total miss: " << totalMiss << "\n";
    for (int qq = 0; qq < NFrames; qq++) cout << frames[qq] << " ";
    cout << "\n";
  }
}
  /***************************************************************************/
 /********************************<The End>********************************/
/***************************************************************************/
```

```
/*******************************************************************************
                           OS: CPU Scheduling Task
                    3rd Computer Engineering,Helwan University
        1- (030) ايه معوض راشد
        2- (043) ريهام محمد ابو اليزيد
        3- (038) حسن ابراهيم فتوح
        4- (097) مصطفى مجدى احمد عثمان
        5- (105) هبة اشرف فؤاد طه

*******************************************************************************/

/******************************************************************************/
#include <iostream>

using namespace std;

#define Empty -100
/******************************************************************************/

  /****************************************************************************/
 /*******************************Linked List Queue****************************/
/****************************************************************************/
class LinkedListQueue {
  // Linked List Queue Node
  struct QueueNode {
    int Data;         // Hold the value
    QueueNode* Next;  // Point to the Next Node
  };

 private:
  QueueNode *Front, *Rear;

 public:
  // Consrurctor: Initialization of Queue with create Empty Node
  LinkedListQueue(void) {
    Front = new QueueNode;
    Front->Next = NULL;
    Rear = Front;
  }

  bool isEmpty(void) { return Front == Rear; }

  void enqueue(int data) {
    Rear->Data = data;
    QueueNode* temp = new QueueNode;
    temp->Next = NULL;
    Rear->Next = temp;
    Rear = temp;
  }

  int dequeue(void) {
    if (!isEmpty()) {
      int data = Front->Data;
      QueueNode* temp = Front;
      Front = Front->Next;
      delete temp;
      return data;
    }
    return Empty;
```

```
  }

  int getActualLength(void) {
    int ActualLength = 0;
    for (QueueNode* temp = Front; temp != Rear; temp = temp->Next) {
      ActualLength++;
    }
    return ActualLength;
  }

  void printQueue(void) {
    for (QueueNode* temp = Front; temp != Rear; temp = temp->Next) {
      cout << temp->Data << "\t";
    }
    cout << "\n";
  }
};
/******************************************************************************/

  /****************************************************************************/
 /****************************Functions Declarations**************************/
/****************************************************************************/
void CPU_Scheduling(void);
void FCFS(int** Processes, int NProcesses);
void SJF_P(int** Processes, int NProcesses);
void SJF_NP(int** Processes, int NProcesses);
void Priority_P(int** Processes, int NProcesses);
void Priority_NP(int** Processes, int NProcesses);
void RR(int** Processes, int NProcesses, int TimeQuantum);
void SortingProcessesAccordingToArrivalTime(int** Processes, int NProcesses);
void FCFS_SJF_NP_Priority_NP_CalclationsOfTimeLine(int** Processes,
                                                   int NProcesses);
void ReArrangingProcessesAccordingToBurstTime(int** Processes, int NProcesses);
void ReArrangingProcessesAccordingToPriority(int** Processes, int NProcesses);
/******************************************************************************/

  /****************************************************************************/
 /*********************************Main Function******************************/
/****************************************************************************/
int main(void) {
  CPU_Scheduling();
  return 0;
}
/******************************************************************************/

  /****************************************************************************/
 /**************************Functions Implementations*************************/
/****************************************************************************/
void CPU_Scheduling(void) {
  // Enter Number of Processes and Time Quantum
  int NProcesses, TimeQuantum;
  cout << "***********************************" << endl;
  cout << "* Enter Number of Processes:\t";
  cin >> NProcesses;
  cout << "***********************************" << endl;
  cout << "* Enter The Time Quantum:\t";
  cin >> TimeQuantum;
  cout << "***********************************" << endl;
  cout << endl;
```

```
  // create Processes Array: 2D
  // Each Process Has (Number & Arrival Time & Burst Time & Priority
  //                          & Waiting Time & Start Time & End Time)
  int** Processes = new int*[NProcesses];
  for (int i = 0; i < NProcesses; i++) {
    Processes[i] = new int[7];
  }

  // Enter Processes
  cout << "***********************************" << endl;
  cout << "**********Enter Processes**********" << endl;
  cout << "***********************************" << endl;
  int TotalBurstTime = 0;
  for (int i = 0; i < NProcesses; i++) {
    cout << "* Process No.(" << i + 1 << "):" << endl;
    Processes[i][0] = i + 1;
    cout << "*\t Arrival Time = ";
    cin >> Processes[i][1];
    cout << "*\t Burst Time = ";
    cin >> Processes[i][2];
    TotalBurstTime += Processes[i][2];
    cout << "*\t Priority = ";
    cin >> Processes[i][3];
    cout << "***********************************" << endl;
    // Waiting Time -> Processes[i][4]
    // Start Time -> Processes[i][5]
    // End Time -> Processes[i][6]
  }
  cout << "*\tTotal Burst Time = " << TotalBurstTime << endl;
  cout << "***********************************" << endl;
  cout << endl;

  FCFS(Processes, NProcesses);
  SJF_P(Processes, NProcesses);
  SJF_NP(Processes, NProcesses);
  Priority_P(Processes, NProcesses);
  Priority_NP(Processes, NProcesses);
  RR(Processes, NProcesses, TimeQuantum);
}
/****************************First Come First Served***************************/
void FCFS(int** Processes, int NProcesses) {
  // Sorting Processes According To Arrival Time
  SortingProcessesAccordingToArrivalTime(Processes, NProcesses);

  cout << "***********************************" << endl;
  cout << "*****First Come First Served*******" << endl;
  cout << "***********************************" << endl;

  // Calclations Of TimeLine
  FCFS_SJF_NP_Priority_NP_CalclationsOfTimeLine(Processes, NProcesses);
}
/******************Sorting Processes According To Arrival Time*****************/
void SortingProcessesAccordingToArrivalTime(int** Processes, int NProcesses) {
  // Sorting Processes According To Arrival Time
  for (int i = 0; i < NProcesses; i++) {
    int MinProcessLoc = i;
    // Get Minimum Process Location
    for (int j = i + 1; j < NProcesses; j++) {
```

```cpp
      if (Processes[j][1] < Processes[MinProcessLoc][1]) {
        MinProcessLoc = j;
      }
    }
    // Swaping
    for (int k = 0; k < 7; k++) {
      int temp = Processes[i][k];
      Processes[i][k] = Processes[MinProcessLoc][k];
      Processes[MinProcessLoc][k] = temp;
    }
  }
}
/************(FCFS + SJF_NP + Priority_NP) Calclations Of Time Line************/
void FCFS_SJF_NP_Priority_NP_CalclationsOfTimeLine(int** Processes,
                                                   int NProcesses) {
  int TotalWaitingTime = 0;
  int TimeLine = Processes[0][1];

  for (int i = 0; i < NProcesses; i++) {
    Processes[i][5] = TimeLine;                             // Start Time
    Processes[i][6] = Processes[i][5] + Processes[i][2];  // End Time
    // Calclate the Waiting Time = Start Time - Arrival Time
    Processes[i][4] = Processes[i][5] - Processes[i][1];
    TotalWaitingTime += Processes[i][4];
    cout << "* Time(" << Processes[i][5] << "->" << Processes[i][6];
    cout << "): Process No.(" << Processes[i][0] << ")" << endl;
    TimeLine += Processes[i][2];
  }
  // Calclate Average Waiting Time
  float AverageWaitingTime = (float)TotalWaitingTime / NProcesses;
  cout << "*************************************" << endl;
  cout << "*\tAverage Waiting Time = " << AverageWaitingTime << endl;
  cout << "*************************************" << endl;
  cout << endl;
}
/***********************Shortest Job First Non-Preemptive*********************/
void SJF_NP(int** Processes, int NProcesses) {
  // Sorting Processes According To Arrival Time
  SortingProcessesAccordingToArrivalTime(Processes, NProcesses);

  // ReArranging Processes According To Burst Time
  ReArrangingProcessesAccordingToBurstTime(Processes, NProcesses);

  cout << "*************************************" << endl;
  cout << "*Shortest Job First Non-Preemptive*" << endl;
  cout << "*************************************" << endl;

  // Calclations Of TimeLine
  FCFS_SJF_NP_Priority_NP_CalclationsOfTimeLine(Processes, NProcesses);
}
/*****************ReArranging Processes According To Burst Time***************/
void ReArrangingProcessesAccordingToBurstTime(int** Processes, int NProcesses) {
  // ReArranging Processes According To Burst Time
  int TimeLineFlage = 0;
  for (int i = 0; i < NProcesses - 1; i++) {
    TimeLineFlage = TimeLineFlage + Processes[i][2];
    int MinProcessLoc = i + 1;
    for (int j = i + 1; j < NProcesses; j++) {
      if (TimeLineFlage >= Processes[j][1] &&
```

```
          Processes[j][2] < Processes[MinProcessLoc][2]) {
        MinProcessLoc = j;
      }
    }
    // Swaping
    for (int k = 0; k < 7; k++) {
      int temp = Processes[i + 1][k];
      Processes[i + 1][k] = Processes[MinProcessLoc][k];
      Processes[MinProcessLoc][k] = temp;
    }
  }
}
/****************************Priority Non-Preemptive***************************/
void Priority_NP(int** Processes, int NProcesses) {
  // Sorting Processes According To Arrival Time
  SortingProcessesAccordingToArrivalTime(Processes, NProcesses);

  // ReArranging Processes According To Priority
  ReArrangingProcessesAccordingToPriority(Processes, NProcesses);

  cout << "***********************************" << endl;
  cout << "*****Priority Non-Preemptive*******" << endl;
  cout << "***********************************" << endl;

  // Calclations Of TimeLine
  FCFS_SJF_NP_Priority_NP_CalclationsOfTimeLine(Processes, NProcesses);
}
/******************ReArranging Processes According To Priority*****************/
void ReArrangingProcessesAccordingToPriority(int** Processes, int NProcesses) {
  // ReArranging Processes According To Priority
  int TimeLineFlage = 0;
  for (int i = 0; i < NProcesses - 1; i++) {
    TimeLineFlage = TimeLineFlage + Processes[i][2];
    int MinProcessLoc = i + 1;
    for (int j = i + 1; j < NProcesses; j++) {
      if (TimeLineFlage >= Processes[j][1] &&
          Processes[j][3] < Processes[MinProcessLoc][3]) {
        MinProcessLoc = j;
      }
    }
    // Swaping
    for (int k = 0; k < 7; k++) {
      int temp = Processes[i + 1][k];
      Processes[i + 1][k] = Processes[MinProcessLoc][k];
      Processes[MinProcessLoc][k] = temp;
    }
  }
}
/*************************Shortest Job First Preemptive*************************/
void SJF_P(int** Processes, int NProcesses) {
  // Sorting Processes According To Arrival Time
  SortingProcessesAccordingToArrivalTime(Processes, NProcesses);

  cout << "***********************************" << endl;
  cout << "***Shortest Job First Preemptive***" << endl;
  cout << "***********************************" << endl;

  int TotalWaitingTime = 0;
  int TimeLine = Processes[0][1];
```

```
/********************************************************/
// Array to save the Remaining Time for each process;initial 0
int* RemainingTime = new int[NProcesses]();
for (int i = 0; i < NProcesses; i++) {
  RemainingTime[i] = Processes[i][2];
}
int CounterOfCompletedProcesses = 0;
int LastProcessNumber = -1;
int LastTimeLine = TimeLine;

while (CounterOfCompletedProcesses < NProcesses) {
  int j;
  for (j = 0; j < NProcesses; j++) {
    if (Processes[j][1] > TimeLine) {
      break;
    }
  }
  // Sorting Processes According To Remaining Time
  for (int z = 0; z < j; z++) {
    int MinProcessLoc = z;
    // Get Minimum Process Location
    for (int y = z + 1; y < j; y++) {
      if (RemainingTime[y] < RemainingTime[MinProcessLoc]) {
        MinProcessLoc = y;
      }
    }
    // Swaping
    int temp = RemainingTime[z];
    RemainingTime[z] = RemainingTime[MinProcessLoc];
    RemainingTime[MinProcessLoc] = temp;
    for (int k = 0; k < 7; k++) {
      temp = Processes[z][k];
      Processes[z][k] = Processes[MinProcessLoc][k];
      Processes[MinProcessLoc][k] = temp;
    }
  }
  if (j > 0) {
    for (j = 0; j < NProcesses; j++) {
      if (RemainingTime[j] != 0) {
        break;
      }
    }
    if (Processes[j][1] > TimeLine) {
      TimeLine = Processes[j][1];
    }
    Processes[j][6] = TimeLine + 1;
    RemainingTime[j]--;
    if ((Processes[j][0] != LastProcessNumber) && (LastProcessNumber != -1)) {
      cout << "* Time(" << LastTimeLine;
      cout << "->" << TimeLine;
      cout << "): Process No.(" << LastProcessNumber << ")" << endl;
      LastTimeLine = TimeLine;
    }
    LastProcessNumber = Processes[j][0];
  }
  TimeLine++;
  CounterOfCompletedProcesses = 0;
  for (j = 0; j < NProcesses; j++) {
    if (RemainingTime[j] == 0) {
```

```
        CounterOfCompletedProcesses++;
      }
    }
  }
  cout << "* Time(" << LastTimeLine;
  cout << "->" << TimeLine;
  cout << "): Process No.(" << LastProcessNumber << ")" << endl;

  for (int i = 0; i < NProcesses; i++) {
    Processes[i][4] = Processes[i][6] - (Processes[i][1] + Processes[i][2]);
    TotalWaitingTime += Processes[i][4];
  }
  /************************************************************/
  // Calclate Average Waiting Time
  float AverageWaitingTime = (float)TotalWaitingTime / NProcesses;
  cout << "***********************************" << endl;
  cout << "*\tAverage Waiting Time = " << AverageWaitingTime << endl;
  cout << "***********************************" << endl;
  cout << endl;
}
/******************************Priority Preemptive*****************************/
void Priority_P(int** Processes, int NProcesses) {
  // Sorting Processes According To Arrival Time
  SortingProcessesAccordingToArrivalTime(Processes, NProcesses);

  cout << "***********************************" << endl;
  cout << "*******Priority Preemptive*********" << endl;
  cout << "***********************************" << endl;

  int TotalWaitingTime = 0;
  int TimeLine = Processes[0][1];
  /************************************************************/
  // Array to save the Remaining Time for each process;initial 0
  int* RemainingTime = new int[NProcesses]();
  for (int i = 0; i < NProcesses; i++) {
    RemainingTime[i] = Processes[i][2];
  }
  int CounterOfCompletedProcesses = 0;
  int LastProcessNumber = -1;
  int LastTimeLine = TimeLine;

  while (CounterOfCompletedProcesses < NProcesses) {
    int j;
    for (j = 0; j < NProcesses; j++) {
      if (Processes[j][1] > TimeLine) {
        break;
      }
    }
    // Sorting Processes According To Priority
    for (int z = 0; z < j; z++) {
      int MinProcessLoc = z;
      // Get Minimum Process Location
      for (int y = z + 1; y < j; y++) {
        if (Processes[y][3] < Processes[MinProcessLoc][3]) {
          MinProcessLoc = y;
        }
      }
      // Swaping
      int temp = RemainingTime[z];
```

```
      RemainingTime[z] = RemainingTime[MinProcessLoc];
      RemainingTime[MinProcessLoc] = temp;
      for (int k = 0; k < 7; k++) {
        temp = Processes[z][k];
        Processes[z][k] = Processes[MinProcessLoc][k];
        Processes[MinProcessLoc][k] = temp;
      }
    }
    if (j > 0) {
      for (j = 0; j < NProcesses; j++) {
        if (RemainingTime[j] != 0) {
          break;
        }
      }
      if (Processes[j][1] > TimeLine) {
        TimeLine = Processes[j][1];
      }
      Processes[j][6] = TimeLine + 1;
      RemainingTime[j]--;
      if ((Processes[j][0] != LastProcessNumber) && (LastProcessNumber != -1)) {
        cout << "* Time(" << LastTimeLine;
        cout << "->" << TimeLine;
        cout << "): Process No.(" << LastProcessNumber << ")" << endl;
        LastTimeLine = TimeLine;
      }
      LastProcessNumber = Processes[j][0];
    }
    TimeLine++;
    CounterOfCompletedProcesses = 0;
    for (j = 0; j < NProcesses; j++) {
      if (RemainingTime[j] == 0) {
        CounterOfCompletedProcesses++;
      }
    }
  }
  cout << "* Time(" << LastTimeLine;
  cout << "->" << TimeLine;
  cout << "): Process No.(" << LastProcessNumber << ")" << endl;

  for (int i = 0; i < NProcesses; i++) {
    Processes[i][4] = Processes[i][6] - (Processes[i][1] + Processes[i][2]);
    TotalWaitingTime += Processes[i][4];
  }
  /**************************************************************/
  // Calclate Average Waiting Time
  float AverageWaitingTime = (float)TotalWaitingTime / NProcesses;
  cout << "************************************" << endl;
  cout << "*\tAverage Waiting Time = " << AverageWaitingTime << endl;
  cout << "************************************" << endl;
  cout << endl;
}
/**********************************Round Robin*********************************/
void RR(int** Processes, int NProcesses, int TimeQuantum) {
  // Sorting Processes According To Arrival Time
  SortingProcessesAccordingToArrivalTime(Processes, NProcesses);

  cout << "************************************" << endl;
  cout << "************Round Robin************" << endl;
  cout << "************************************" << endl;
```

```
int TotalWaitingTime = 0;
int TimeLine = Processes[0][1];

LinkedListQueue ReadyQueue;
// Array to save the Remaining Time for each process;initial 0
int* RemainingTime = new int[NProcesses]();
// Array to indecate if the Process entered the queue before that;initial
// false
bool* EnteredQueueBefore = new bool[NProcesses]();
// Array to indecate if the Process started excution before that;initial false
bool* StartedExcutionBefore = new bool[NProcesses]();

ReadyQueue.enqueue(Processes[0][0]);  // First Process Enter Queue
RemainingTime[0] = Processes[0][2];   // RemainingTime = BurstTime
EnteredQueueBefore[0] = true;

while (ReadyQueue.isEmpty() == false) {
  int ProcessNumber = ReadyQueue.dequeue();
  int ProcessIndex = ProcessNumber - 1;
  int ProcessRemainingTime = RemainingTime[ProcessIndex];

  if (TimeQuantum >= ProcessRemainingTime && ProcessRemainingTime > 0) {
    cout << "* Time(" << TimeLine;
    TimeLine += ProcessRemainingTime;
    cout << "->" << TimeLine;
    cout << "): Process No.(" << ProcessNumber << ")" << endl;
    RemainingTime[ProcessIndex] = 0;
    Processes[ProcessIndex][6] = TimeLine;  // End Time
    // Calclate the Waiting Time = End Time - (Arrival Time + Burst Time])
    Processes[ProcessIndex][4] =
        Processes[ProcessIndex][6] -
        (Processes[ProcessIndex][1] + Processes[ProcessIndex][2]);
    TotalWaitingTime += Processes[ProcessIndex][4];

    for (int i = 0; i < NProcesses; i++) {
      if (Processes[i][1] <= TimeLine && EnteredQueueBefore[i] == false) {
        ReadyQueue.enqueue(Processes[i][0]);
        RemainingTime[i] = Processes[i][2];
        EnteredQueueBefore[i] = true;
      }
    }
  } else if (TimeQuantum < ProcessRemainingTime) {
    cout << "* Time(" << TimeLine;
    TimeLine += TimeQuantum;
    cout << "->" << TimeLine;
    cout << "): Process No.(" << ProcessNumber << ")" << endl;
    RemainingTime[ProcessIndex] = ProcessRemainingTime - TimeQuantum;

    for (int i = 0; i < NProcesses; i++) {
      if (Processes[i][1] <= TimeLine && EnteredQueueBefore[i] == false) {
        ReadyQueue.enqueue(Processes[i][0]);
        RemainingTime[i] = Processes[i][2];
        EnteredQueueBefore[i] = true;
      }
    }

    ReadyQueue.enqueue(ProcessNumber);
  }
```

```
    }
    // Calclate Average Waiting Time
    float AverageWaitingTime = (float)TotalWaitingTime / NProcesses;
    cout << "***********************************" << endl;
    cout << "*\tAverage Waiting Time = " << AverageWaitingTime << endl;
    cout << "***********************************" << endl;
    cout << endl;
}
  /****************************************************************************/
 /**********************************<The End>*********************************/
/****************************************************************************/
```

```
/******************************************************************************

                      OS: Deadlock Task/Detection Algorithm
                    3rd Computer Engineering,Helwan University
        1- (030) ايه معوض راشد
        2- (043) ريهام محمد ابو اليزيد
        3- (038) حسن ابراهيم فتوح
        4- (097) مصطفى مجدى احمد عثمان
        5- (105) هبة اشرف فؤاد طه

*******************************************************************************/

/*******************************************************************************/
#include <iostream>

using namespace std;
/*******************************************************************************/

  /*****************************************************************************/
 /*********************************Main Function******************************/
/*****************************************************************************/
int main() {
  // Enter NO. of processes
  int NProcesses;
  cout << "***********************************" << endl;
  cout << "* Enter Number of Processes:\t";
  cin >> NProcesses;
  cout << "***********************************" << endl;
  cout << endl;

  // create : 2D Array
  // Each Process Has (Number & Allocation(ABC) & MAX(ABC) & Available (ABC)
  //                          & Need(ABC) )//13 coulmn

  // Safe_Sequance Array
  int *Safe_Sequance = new int[NProcesses];

  // Array to hold the remaining Processes
  int *hold = new int[NProcesses];

  int **Processes = new int *[NProcesses];
  for (int i = 0; i < NProcesses; i++) {
    Processes[i] = new int[13];
  }
  // Enter Processes
  cout << "***********************************" << endl;
  cout << "**********Enter Processes**********" << endl;
  cout << "***********************************" << endl;

  // Enter the Allocation & MAX
  for (int i = 0; i < NProcesses; i++) {
    cout << "* Process No.(" << i + 1 << "):" << endl;
    Processes[i][0] = i + 1;
    cout << "*\t Allocation(ABC) = ";
    cin >> Processes[i][1];
    cin >> Processes[i][2];
    cin >> Processes[i][3];
    cout << "*\t Need(ABC) = ";
    cin >> Processes[i][4];
```

```
    cin >> Processes[i][5];
    cin >> Processes[i][6];
    cout << "************************************" << endl;
  }

  // Available resource
  cout << "*\t Enter Available(ABC) = ";
  cin >> Processes[0][7];
  cin >> Processes[0][8];
  cin >> Processes[0][9];
  cout << "************************************" << endl;

  // Print Processes Array
  cout << "************************************" << endl;
  cout << "************Processes Array**********" << endl;
  cout << "************************************" << endl;
  cout << "Process | Allocation | MAX | Available | Need |" << endl;
  cout << "------  |----ABC---  | ABC | ---ABC--- | -ABC |" << endl;
  cout << "----------------------------------------------" << endl;
  for (int i = 0; i < NProcesses; i++) {
    for (int j = 0; j < 13; j++) {
      cout << Processes[i][j] << " "
           << " ";
    }
    cout << endl;
  }

  // pointer to fill the Safe_Sequance Array and hold Array
  int spointer = 0;
  int hpointer = 0;

  for (int i = 0; i < NProcesses; i++) {
    // if Process needs < Available put it on Safe_Sequance Array and change the
    // Available resources
    if (Processes[i][10] < Processes[0][7] &&
        Processes[i][11] < Processes[0][8] &&
        Processes[i][12] < Processes[0][9]) {
      Safe_Sequance[spointer] = Processes[i][0];
      spointer++;
      Processes[0][7] = Processes[0][7] + Processes[i][1];
      Processes[0][8] = Processes[0][8] + Processes[i][2];
      Processes[0][9] = Processes[0][9] + Processes[i][3];
    } else {
      hold[hpointer] = Processes[i][0];
      hpointer++;
    }
  }

  // the Processes in hold Array
  while (hpointer != 0) {
    for (int i = 0; i < hpointer; i++) {
      for (int j = 0; j < NProcesses; j++) {
        if (hold[i] == Processes[j][0] && Processes[j][10] < Processes[0][7] &&
            Processes[j][11] < Processes[0][8] &&
            Processes[j][12] < Processes[0][9]) {
          Safe_Sequance[spointer] = hold[i];
          spointer++;
          Processes[0][7] += Processes[j][1];
          Processes[0][8] += Processes[j][2];
```

```
          Processes[0][9] += Processes[j][3];
          hpointer--;
        } else {
          cout << " There is a Deadlock " << endl;
          return 0;
        }
      }
    }
  }

  // print Safe_Sequance Array
  cout << "***********************************" << endl;
  cout << "************Safe_Sequance**********" << endl;
  cout << "***********************************" << endl;
  for (int i = 0; i < spointer; i++) {
    cout << Safe_Sequance[i] << "\t";
  }

  return 0;
}
  /****************************************************************************/
 /**********************************<The End>*********************************/
/****************************************************************************/
```

```
/******************************************************************************

                        OS:Deadlock Task/Banker Algorithm
                    3rd Computer Engineering,Helwan University
        1- (030) ايه معوض راشد
        2- (043) ريهام محمد ابو اليزيد
        3- (038) حسن ابراهيم فتوح
        4- (097) مصطفى مجدى احمد عثمان
        5- (105) هبة اشرف فؤاد طه

*******************************************************************************/

/******************************************************************************/
#include <iostream>

using namespace std;
/******************************************************************************/

  /****************************************************************************/
 /*********************************Main Function******************************/
/*****************************************************************************/
int main() {
  // Enter NO. of processes
  int NProcesses;
  cout << "***********************************" << endl;
  cout << "* Enter Number of Processes:\t";
  cin >> NProcesses;
  cout << "***********************************" << endl;
  cout << endl;

  // create : 2D Array
  // Each Process Has (Number & Allocation(ABC) & MAX(ABC) & Available (ABC)
  //                          & Need(ABC) )//13 coulmn

  int **Processes = new int *[NProcesses];
  for (int i = 0; i < NProcesses; i++) {
    Processes[i] = new int[13];
  }
  // Enter Processes
  cout << "***********************************" << endl;
  cout << "**********Enter Processes**********" << endl;
  cout << "***********************************" << endl;

  // Enter the Allocation & MAX
  for (int i = 0; i < NProcesses; i++) {
    cout << "* Process No.(" << i + 1 << "):" << endl;
    Processes[i][0] = i + 1;
    cout << "*\t Allocation(ABC) = ";
    cin >> Processes[i][1];
    cin >> Processes[i][2];
    cin >> Processes[i][3];
    cout << "*\t MAX(ABC) = ";
    cin >> Processes[i][4];
    cin >> Processes[i][5];
    cin >> Processes[i][6];
    cout << "***********************************" << endl;
  }

  // Available resource
```

```
  cout << "*\t Enter Available(ABC) = ";
  cin >> Processes[0][7];
  cin >> Processes[0][8];
  cin >> Processes[0][9];
  cout << "***********************************" << endl;

  // Calculate Need of Each Process (MAX - Available)
  for (int i = 0; i < NProcesses; i++) {
    Processes[i][10] = Processes[i][4] - Processes[i][1];
    Processes[i][11] = Processes[i][5] - Processes[i][2];
    Processes[i][12] = Processes[i][6] - Processes[i][3];
  }

  // Print Processes Array
  cout << "***********************************" << endl;
  cout << "************Processes Array**********" << endl;
  cout << "***********************************" << endl;
  cout << "Process | Allocation | MAX | Available | Need |" << endl;
  cout << "------  |----ABC---  | ABC | ---ABC--- | -ABC |" << endl;
  cout << "-----------------------------------------------" << endl;
  for (int i = 0; i < NProcesses; i++) {
    for (int j = 0; j < 13; j++) {
      cout << Processes[i][j] << " "
           << " ";
    }
    cout << endl;
  }

  // Safe_Sequance Array
  int *Safe_Sequance = new int[NProcesses];

  // Array to hold the remaining Processes
  int *hold = new int[NProcesses];

  // pointer to fill the Safe_Sequance Array and hold Array
  int spointer = 0;
  int hpointer = 0;

  for (int i = 0; i < NProcesses; i++) {
    // if Process needs < Available put it on Safe_Sequance Array and change the
    // Available resources
    if (Processes[i][10] < Processes[0][7] &&
        Processes[i][11] < Processes[0][8] &&
        Processes[i][12] < Processes[0][9]) {
      Safe_Sequance[spointer] = Processes[i][0];
      spointer++;
      Processes[0][7] = Processes[0][7] + Processes[i][1];
      Processes[0][8] = Processes[0][8] + Processes[i][2];
      Processes[0][9] = Processes[0][9] + Processes[i][3];
    } else {
      hold[hpointer] = Processes[i][0];
      hpointer++;
    }
  }

  // the Processes in hold Array
  while (hpointer != 0) {
    for (int i = 0; i < hpointer; i++) {
      for (int j = 0; j < NProcesses; j++) {
```

```
        if (hold[i] == Processes[j][0] && Processes[j][10] < Processes[0][7] &&
            Processes[j][11] < Processes[0][8] &&
            Processes[j][12] < Processes[0][9]) {
          Safe_Sequance[spointer] = hold[i];
          spointer++;
          Processes[0][7] += Processes[j][1];
          Processes[0][8] += Processes[j][2];
          Processes[0][9] += Processes[j][3];
          hpointer--;
        }
      }
    }
  }

  // print Safe_Sequance Array
  cout << "***********************************" << endl;
  cout << "************Safe_Sequance**********" << endl;
  cout << "***********************************" << endl;
  for (int i = 0; i < spointer; i++) {
    cout << Safe_Sequance[i] << "\t";
  }
  return 0;
}
  /****************************************************************************/
 /**********************************<The End>*********************************/
/****************************************************************************/
```