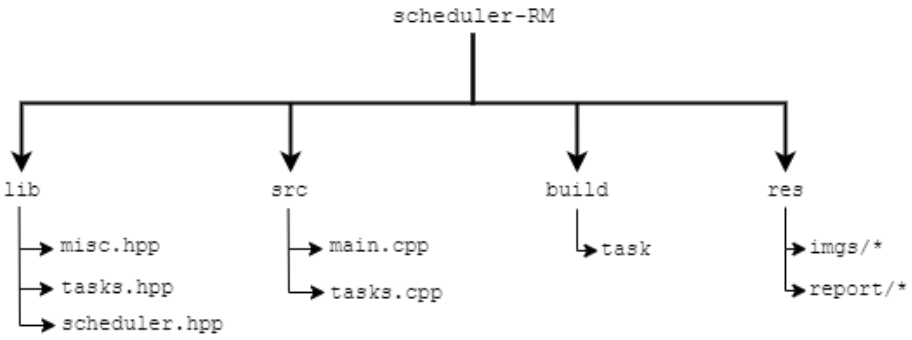
Code listing for the software

The entire source code is split up into multiple files, which are orgainised in the following manner:



main.cpp: This is the main application which the user will interact with.

```
#include "tasks.hpp"
#include "misc.hpp"
#include "scheduler.hpp"

int main(void){

  // Create three tasks
  createTask("T1", 3, 20);
  createTask("T2", 2, 5);
  createTask("T3", 2, 10);

  // Show the set of tasks
  showTaskSet();

  // Run the scheduler
  //std::vector <std::string> s = scheduleRM();
  printSchedule(scheduleRM());
  return 0;
}
```

tasks.hpp : This file contains the definition for the taskControlBlock structure and various functions that pertains to the manipulation of a task.

```
#ifndef TASKS
#define TASKS
#include <iostream>
#include <vector>

typedef struct {
  std::string taskID;          // Stores the task identifier
  int compTime;                // Stores the computation time for the task
  int period;                  // Stores the period(=deadline) of the task
  int remTime;                 // Stores the remaining execution time of the task
  float utility;               // Stores the utility of the task
  int isReady;                 // Stores the current state of the task
} taskControlBlock;

extern std::vector<taskControlBlock> taskSet;

void createTask(std::string tName, int c, int p);
void showTaskSet();
std::string printTaskState(taskControlBlock t);

#endif
```

tasks.cpp : This file contains the source code for the functions defined in `lib/tasks.hpp`

```
#include "tasks.hpp"
#include <ctime>

std::vector<taskControlBlock> taskSet;

/* function createTask : creates a task by initialising
 * a taskControlBlock element with the given values and
 * pushing it to the vector of taskControlBlock elements.
 */
void createTask(std::string tName, int c, int p) {
  taskControlBlock temp;

  temp.taskID   = tName;
  temp.compTime = c;
  temp.period   = p;
  temp.remTime  = c;
  temp.utility  = (float)c/(float)p;
  temp.isReady  = 1;

  taskSet.push_back(temp);
}

/* function printTaskState : Small helper function to print the state of the task
 */
std::string printTaskState(taskControlBlock t) {
  if(t.isReady)
    return "READY";
  else
    return "WAITING";
}

/* function showTaskSet : Iterates through the set of
 * tasks and print them.
 */
void showTaskSet() {

  std::cout << "\t======== TASK STATS: ========\n" << std::endl;
  std::cout << "Task ID" << "     Computation Time" << "\tPeriod" << "\tUtility" << "\
    tState" << std::endl;
  for(auto elem: taskSet) {
    std::cout << "    " << elem.taskID << "\t\t" << elem.compTime << "\t\t" << elem.
    period << "\t" << elem.utility << "\t" << printTaskState(elem) << std::endl;
  }
  std::cout << "\t\t====== * ======" << std::endl;
}
```

misc.hpp: This is a small file with helper functions pertaining to a more general use case.

```
#ifndef MISC
#define MISC

#include "tasks.hpp"

/* function comparePeriod : takes in two taskControlBlock
 * elements, and returns true if the first element has a
 * smaller task period than the second element.
 */
bool comparePeriod(taskControlBlock x, taskControlBlock y) {
  if(x.period < y.period)
    return true;
  else
    return false;
}

/* function gcd : Computes the Greatest Common Denominator
 * between two given numbers using recursion
 */
int gcd(int a, int b) {
  if(b == 0) {
    return a;
  }
  else {
    return gcd(b, a%b);
  }
}

/* function computeHyperPeriod : computes the hyperperiod
 * of a given vector of taskControlBlock elements.
 */
int computeHyperPeriod(std::vector <taskControlBlock> vec) {
  int n = vec.size();               // Calculate the size of the vector

  int ans = vec.begin()->period;   // Choose the first element as the default answer

  for (auto elem : vec) {
    ans = (int)(elem.period * ans) / (gcd(elem.period, ans));
  }
  return ans;
}

#endif
```

scheduler.hpp: This file contains all the important functions and parameters related to the scheduler.

```
#ifndef SCHEDULER
#define SCHEDULER

#include "misc.hpp"
#include <algorithm>
#include <cmath>

bool necTest = false, sufTest = false, isSched = false;
float totalUtil = 0.00, feasible = 0.0;
int idleTime = 0;

/* function testSchedulability : Runs important schedulability
 * tests on the given set of task
 */
bool testSchedulability() {

  std::cout << "\n\n" << "-------- SCHEDULABILITY ANALYSIS ------- " << std::endl;

  for(auto elem: taskSet)
    totalUtil += elem.utility;            // Iterate through the task set and compute
   the total utilisation

  feasible = taskSet.size() * (std::pow(2, 1/(double)taskSet.size()) - 1);
  std::cout << "[INFO] Number of tasks            : " << taskSet.size() << std::endl;
  std::cout << "[INFO] Hyperperiod is             : " << computeHyperPeriod(taskSet) <<
     " time units" << std::endl;
  std::cout << "[INFO] Tota Processor Utilisation : " << totalUtil << std::endl;
  std::cout << "[INFO] Feasibility Metric is      : " << feasible << std::endl;
  std::cout << "[INFO] Running Sufficient Test    : ";
  if(totalUtil < feasible) {              // Sufficient schedulability test
    std::cout << "PASSED" << std::endl;
    sufTest = true;
  }
  else {
    std::cout << "FAILED" << std::endl;
    sufTest = false;
  }

  std::cout << "[INFO] Running Necessary Test     : ";
  if(totalUtil < 1.0) {                   // Necessary schedulability test
    std::cout << "PASSED" << std::endl;
    necTest = true;
  }
  else {
    std::cout << "FAILED" << std::endl;
    necTest = false;
  }

  std::cout << "-------------------------------------" << std::endl;

  if(necTest & sufTest)
    isSched = true;                     // Returns true if both tests pass

  return isSched;
}
```

scheduler.hpp *(contd.)*

```
/* function scheduleRM : Implements the Fixed Priority Rate Monotonic
 * scheduling algorithm on the task set
 */
std::vector<std::string> scheduleRM(){

  std::vector<std::string> sched;
  int hp = computeHyperPeriod(taskSet);
  int n = taskSet.size();
  sort(taskSet.begin(), taskSet.end(), comparePeriod);      // Sort the task set in
    ascending order of period

  if(testSchedulability()){                                  // If the task list is
    schedulable, then go on
    for(int i=0 ; i< hp ; i++){
      if(i>=taskSet[0].period)
        for(int j=0 ; j<n ; j++){
          if(!(i%taskSet[j].period))                        // If the task is restarted
              taskSet[j].isReady=1;                         // Set the state to ready
        }
        for(int j=0 ; j<n ; j++){
          if(taskSet[j].isReady && (taskSet[j].remTime)){  //If task is ready and has
    remaining execution time
              if(!(--(taskSet[j].remTime))){               //If task has executed fully
                  taskSet[j].isReady=0;                    //Set task status to waiting
                  taskSet[j].remTime=taskSet[j].compTime;  //Reset the remaining
    execution time of the task
              }
              sched.push_back(" " + taskSet[j].taskID + " ");
              break;
          }
          if(j == n-1) {
            sched.push_back("IDLE");
            idleTime++;
          }
        }
    }
  }
  return sched;
}

void printSchedule(std::vector<std::string> s) {

  std::cout << "\n\tSchedule:\t\t" << std::endl;
  std::cout << "Task\t|   ";

  for(auto elem: s)
    std::cout << elem << "   ";

  std::cout << "\n\nTime\t|   ";
  for(int t = 0; t < computeHyperPeriod(taskSet); t++) {
    std::cout << t << "    ";
  }
  std::cout << "\n-------------------------------" <<std::endl;
  std::cout << "[INFO] Maximum Processor Idle     : " << idleTime << " time units" <<
    std::endl;
  std::cout << std::endl;
}

#endif
```