feasibility_tests.c

```c
/********************************************************************************
 * Copyright (C) 2023 by Sam Siewert, Parth Thakkar
 *
 * Redistribution, modification or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Parth Thakkar and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/

/**
 * @file    feasibility_tests.c
 * @brief  This example code provides feasibiltiy decision tests for single core fixed
 * priority rate monontic systems only (not dyanmic priority such as deadline
 * driven EDF and LLF). These are standard algorithms which either estimate
 * feasibility (as the RM LUB does) or automate exact analysis (scheduling point,
 * completion test) for a set services sharing one CPU core. This can be emulated on Linux SMP
 * multi-core systemes by use of POSIX thread affinity, to "pin" a thread to a
 * specific core.  Coded based upon standard definition of:
 * 1) RM LUB based upon model by Liu and Layland
 * 2) Scheduling Point - an exact feasibility algorithm based upon Lehoczky, Sha, and Ding exact analysis
 * 3) Completion Test - an exact feasibility algorithm
 *
 *
 * @author  Parth Thakkar, Sam Siewert
 * @date    20th Sept 2023
 *
 */

#include <math.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

#define U32_T unsigned int
#define EXAMPLES 10

#define MAX_TASKS 4

typedef struct
{
    U32_T wcet[MAX_TASKS];
    U32_T period[MAX_TASKS];
    U32_T deadline[MAX_TASKS];
    U32_T num_tasks;
} task_set_t;

// Example tasks
```

```

task_set_t tasks[EXAMPLES] = {
    {
        // 0
        .period = {2, 10, 15},   // Example Periods
        .wcet = {1, 1, 2},       // Example Worst Case Execution Times
        .deadline = {2, 10, 15}, // Example Deadlines
        .num_tasks = 3           // Number of tasks in this set
    },

    {
        // 1
        .period = {2, 5, 7},   // Example Periods
        .wcet = {1, 1, 2},     // Example Worst Case Execution Times
        .deadline = {2, 5, 7}, // Example Deadlines
        .num_tasks = 3         // Number of tasks in this set
    },

    {
        // 2
        .period = {2, 5, 7, 13},   // Example Periods
        .wcet = {1, 1, 1, 2},      // Example Worst Case Execution Times
        .deadline = {2, 5, 7, 13}, // Example Deadlines
        .num_tasks = 4             // Number of tasks in this set
    },

    {
        // 3
        .period = {3, 5, 15},   // Example Periods
        .wcet = {1, 2, 3},      // Example Worst Case Execution Times
        .deadline = {3, 5, 15}, // Example Deadlines
        .num_tasks = 3          // Number of tasks in this set
    },

    {
        // 4
        .period = {2, 4, 16},   // Example Periods
        .wcet = {1, 1, 4},      // Example Worst Case Execution Times
        .deadline = {2, 4, 16}, // Example Deadlines
        .num_tasks = 3          // Number of tasks in this set
    },
    {
        // 5
        .period = {2, 4, 16},   // Example Periods
        .wcet = {1, 1, 4},      // Example Worst Case Execution Times
        .deadline = {2, 4, 16}, // Example Deadlines
        .num_tasks = 3          // Number of tasks in this set
    },

    {
        // 6
        .period = {2, 5, 7, 13},   // Example Periods
        .wcet = {1, 1, 1, 2},      // Example Worst Case Execution Times
        .deadline = {2, 3, 7, 15}, // Example Deadlines
        .num_tasks = 4             // Number of tasks in this set
    },

    {
        // 7
```

```c
        .period = {3, 5, 15},   // Example Periods
        .wcet = {1, 2, 4},      // Example Worst Case Execution Times
        .deadline = {3, 5, 15}, // Example Deadlines
        .num_tasks = 3          // Number of tasks in this set
    },

    {
        // 8
        .period = {2, 5, 7, 13},   // Example Periods
        .wcet = {1, 1, 1, 2},      // Example Worst Case Execution Times
        .deadline = {2, 5, 7, 13}, // Example Deadlines
        .num_tasks = 4             // Number of tasks in this set
    },
    {
        // 9
        .period = {6, 8, 12, 24},   // Example Periods
        .wcet = {1, 2, 4, 6},       // Example Worst Case Execution Times
        .deadline = {6, 8, 12, 24}, // Example Deadlines
        .num_tasks = 4              // Number of tasks in this set
    }};

// Feasibility test functions
bool completion_time_feasibility(task_set_t *task_set);
bool scheduling_point_feasibility(task_set_t *task_set);
bool rate_monotonic_least_upper_bound(task_set_t *task_set);
int edf_feasibility(task_set_t *task_set, bool deadline);
int llf_feasibility(task_set_t *task_set, bool deadline);
bool deadline_monotonic_feasibility(task_set_t *task_set);

int main(void)
{

    for (int i = 0; i < EXAMPLES; i++)
    {
        printf("\n****************\n");
        printf("Example %d\n", i);

        // Calculate and print total utilization
        double U = 0.0;
        for (int j = 0; j < tasks[i].num_tasks; j++)
        {
            U += (double)tasks[i].wcet[j] / tasks[i].period[j];
        }
        printf("C: ");
        for (int j = 0; j < tasks[i].num_tasks; j++)
        {
            printf("%d ", tasks[i].wcet[j]);
        }
        printf("\nT: ");
        for (int j = 0; j < tasks[i].num_tasks; j++)
        {
            printf("%d ", tasks[i].period[j]);
            if(tasks[i].period[j] ==0){
                printf("Pseriod is zero\n");
                exit(0);
            }
        }
        printf("\nD: ");
        int dm = 0;
```

```c
        for (int j = 0; j < tasks[i].num_tasks; j++)
        {
            printf("%d ", tasks[i].deadline[j]);
            if(tasks[i].deadline[j] ==0){
                printf("Deadline is zero\n");
                exit(0);
            }
            if (tasks[i].deadline[j] != tasks[i].period[j])
            {
                dm++;
            }
        }
        // printf("\nUtility : %4.2f%%\n", U * 100);

        // Perform and print feasibility tests
        printf("RM LUB: %s\n", rate_monotonic_least_upper_bound(&tasks[i]) ? "Feasible" : "Infeasible");
        printf("Completion time feasibility: %s\n", completion_time_feasibility(&tasks[i]) ? "Feasible" : "Infeasible");
        printf("Scheduling point feasibility: %s\n", scheduling_point_feasibility(&tasks[i]) ? "Feasible" : "Infeasible");

        if (dm != 0)
        {
            printf("Deadline monotonic: %s\n", deadline_monotonic_feasibility(&tasks[i]) ? "Feasible" : "Infeasible");

            printf("\n(Period)");
            printf("EDF on Period: %s\n", edf_feasibility(&tasks[i], false) ? "Feasible" : "Infeasible");
            printf("LLF on Period: %s\n", llf_feasibility(&tasks[i], false) ? "Feasible" : "Infeasible");

            printf("\n(Deadline)");
            printf("EDF on Deadline: %s\n", edf_feasibility(&tasks[i], true) ? "Feasible" : "Infeasible");
            printf("LLF on Deadline: %s\n", llf_feasibility(&tasks[i], true) ? "Feasible" : "Infeasible");
        }
        else if (i > 4)
        {
            printf("\n(Period)");
            printf("EDF: %s\n", edf_feasibility(&tasks[i], false) ? "Feasible" : "Infeasible");
            printf("LLF: %s\n", llf_feasibility(&tasks[i], false) ? "Feasible" : "Infeasible");
        }

        // Add other feasibility tests here

        printf("\n");
    }
}

bool rate_monotonic_least_upper_bound(task_set_t *task_set)
{
    double utility_sum = 0.0, lub = 0.0;
    int idx;

    // Sum the C(i) over the T(i) for utility calculation
    printf("\n\n");
```

```c
    for (idx = 0; idx < task_set->num_tasks; idx++)
    {
        utility_sum += ((double)task_set->wcet[idx] / (double)task_set->period[idx]);
        printf("Task %d, WCET=%u, Period=%u, Utility Sum = %lf\n", idx, task_set->wcet[idx], task_set->period[idx], utility_sum);
    }
    printf("\nTotal Utility Sum = %lf\n", utility_sum);

    // Compute LUB for the number of services
    lub = (double)task_set->num_tasks * ((pow(2.0, (1.0 / (double)task_set->num_tasks))) - 1.0);
    printf("LUB = %lf\n", lub);

    // Compare the utility sum to the bound and return feasibility
    if (utility_sum <= lub)
        return TRUE;
    else
        return FALSE;
}

bool completion_time_feasibility(task_set_t *task_set)
{
    int i, j;
    U32_T an, anext;
    int set_feasible = TRUE;

    // For all tasks in the analysis
    for (i = 0; i < task_set->num_tasks; i++)
    {
        an = 0;
        anext = 0;

        for (j = 0; j <= i; j++)
        {
            an += task_set->wcet[j];
        }

        while (1)
        {
            anext = task_set->wcet[i];

            for (j = 0; j < i; j++)
                anext += ceil((double)an / (double)task_set->period[j]) * task_set->wcet[j];

            if (anext == an)
                break;
            else
                an = anext;
        }

        if (an > task_set->period[i])
        {
            set_feasible = FALSE;
        }
    }

    return set_feasible;
}
```

```c

bool scheduling_point_feasibility(task_set_t *task_set)
{
    int rc = TRUE, i, j, k, l, status, temp;

    // For all tasks in the analysis
    for (i = 0; i < task_set->num_tasks; i++)
    { // iterate from highest to lowest priority
        status = 0;

        // Look for all available CPU minus what has been used by higher priority tasks
        for (k = 0; k <= i; k++)
        {
            // find available CPU windows and take them
            for (l = 1; l <= (floor((double)task_set->period[i] / (double)task_set->period[k])); l++)
            {
                temp = 0;

                for (j = 0; j <= i; j++)
                    temp += task_set->wcet[j] * ceil((double)l * (double)task_set->period[k] / (double)task_set->period[j]);

                // Can we get the CPU we need or not?
                if (temp <= (l * task_set->period[k]))
                {
                    // insufficient CPU during our period, therefore infeasible
                    status = 1;
                    break;
                }
            }
            if (status)
                break;
        }

        if (!status)
            rc = FALSE;
    }
    return rc;
}

int llf_feasibility(task_set_t *task_set, bool deadline)
{
    double totalU = 0.0;
    if (!deadline)
    {
        for (int i = 0; i < task_set->num_tasks; i++)
        {
            totalU += (double)task_set->wcet[i] / task_set->period[i];
        }
    }
    else
    {
        for (int i = 0; i < task_set->num_tasks; i++)
        {
            totalU += (double)task_set->wcet[i] / task_set->deadline[i];
        }
    }
```

```c
    printf("Total utility in LLF: %f ", totalU);
    if (totalU <= 1.0)
    {
        printf("Which is less than 1.0 \n");
        return TRUE;
    }
    else
    {
        printf("Which is less than 1.0 \n");
        return FALSE;
    }
}

bool deadline_monotonic_feasibility(task_set_t *task_set)
{
    //Ensure tasks are sorted by their deadlines before running this feasibility test.
    int status = 0;
    for (int i = 0; i < task_set->num_tasks; i++)
    {
        float interference = 0;
        float utilization = 0;
        for (int j = 0; j < i; j++)
        {
            interference += (ceil((float)task_set->deadline[i] / (float)task_set->period[j])) * (float)task_set->wcet[j];
        }
        utilization = ((float)(task_set->wcet[i]) / (float)task_set->deadline[i]) + (interference / (float)task_set->deadline[i]);
        if (utilization > 1)
        {
            status = 1;
            break;
        }
    }
    if (status == 1)
        return FALSE;
    else
        return TRUE;
}

int edf_feasibility(task_set_t *task_set, bool deadline)
{
    double totalU = 0.0;
    if (!deadline)
    {
        for (int i = 0; i < task_set->num_tasks; i++)
        {
            totalU += (double)task_set->wcet[i] / task_set->period[i];
        }
    }
    else
    {
        for (int i = 0; i < task_set->num_tasks; i++)
        {
            totalU += (double)task_set->wcet[i] / task_set->deadline[i];
        }
    }
    printf("\nTotal utility in EDF: %f ", totalU);
```

```
    if (totalU <= 1.0)
    {
        printf("Which is less than 1.0 \n");
        return TRUE;
    }
    else
    {
        printf("Which is less than 1.0 \n");
        return FALSE;
    }
}

```