A Review on Multi-core Partitioned Scheduling

Shahzaib Waseem
B.Eng. Electronics Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
shahzaib.waseem@stud.hshl.de

***Abstract*—Multi-core systems were introduced as a next step from single core systems for reasons such as better multitasking capabilities. However, it with its own unique set of challenges especially in the domain of task scheduling. Task scheduling in multi-core systems depended on one of three approximation algorithms / task assignment algorithms: first fit, best fit, worst fit. While each one of these have their own advantages and disadvantages, in this paper, best fit algorithm was chosen. Then, after appliying best fit algorithm, each core ran the Earliest Deadline First scheduling algorithm. This paper then showcased a Uppaal implementation of this EDF scheduling algorithm that simulated one core running EDF and which scheduled two periodic tasks with varying periods and burst times.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Computers contained just a single processor up until the late 1900's, around when computers were first introduced [1]. While the invention of computers was a remarkable achievement, these computers were far from perfect. Since they relied on a single CPU, only a single program instruction could be executed at a time [1]. The concept of parallelization was, of course, then unknown [1]. Such single-core computers were naturally more vulnerable to bottlenecks when interfacing with faster computer peripherals that may have needed to wait for the computer to complete the task at hand [1]. Additionally, the weak multitasking nature of these computers especially in the context of hard-real time systems in combination with their constrained memory motivated computer engineers to research and develop multi-core systems [1].

As the name suggests, multi-core systems contain more than one processing unit and are able to handle execution of multiple program instructions in parallel [2]. This addresses the shortcoming of single-core processors first in the area of computer peripherals waiting for the CPU to become free by having another processor to handle their interfacing tasks [2]. Next, multi-core processors relatively increase a computer's ability to do multitasking by dividing execution of tasks among their multiple processors [2]. Interestingly, this leads to another benefit of better power consumption, as the workload on a single core is reduced [2]. However, even multi-core systems have their flaws in certain scenarios.

Having multiple cores or processors in a system implies that these cores must cooperate by sharing resources. However, mutually exclusive ownership/usage of a resource by a process at any given point in time may cause blocking and starvation for other cores and tasks. This delay faced by cores may lead to missing task deadlines, and that too in hard real-time systems in particular can cause catastrophic outcomes. Another factor in missing deadlines in multi-core systems is allowing tasks to switch between cores by disturbing the continuity of their usage of a shared resource, and having them potentially wait for their turn in doing so again [5]. Therefore, scheduling algorithms tailored and specialized for multi-core systems are needed.

Scheduling algorithms for multi-core systems are not and can not be directly mapped from existing scheduling algorithms for single-core systems [3]. Two of the most important reasons for this is that there is no concept of shared resources in single-core systems, and that the compatibility and affinity between a task and its executing core is not considered in single-core systems [4]. Below are some of the factors that a scheduling algorithm running on a multi-core system must consider, in the author's understanding:

- Predictability
- Resource Utilization
- Synchronization

Predictability is the ability to determine beforehand whether a set of tasks can be scheduled in a way such that all their deadlines are met (schedulaibility test) [8]. This can naturally allow for computational power to be directed towards set of tasks for which success is foreseen, and readjustment of the set of tasks or the algorithm in case failure is foreseen [8]. Resource Utilization refers to maximizing throughput and minimizing idle times for every core in the system [7]. Doing so can increase productivity as tasks can be executed faster in larger quantities since the core responsible for them performs all the work it can instead of being occasionally idle [7]. This is important especially for real-time systems where achieving the least amount of delay in completing a set of tasks is crucial, as it is important for the system to be free for any incoming tasks in the future. Synchronization is important to prevent race conditions in real-time systems [9]. This can happen when multiple tasks attempt to concurrently read and/or modify a shared resource, and end up overwriting data that another task was originally using, thereby causing outdated information to be used in computations within the real-time multi-core system [9]. Synchronization is also important in avoiding deadlocks in a system. Deadlocks can turn out to be extremely troublesome and tricky to handle, considering that their very nature contributes to increasing the likelihood of a task missing

its deadline [9].

Considering the factors described above, quite a few scheduling algorithms for multi-core systems have been developed. These task scheduling algorithms are of three categories:

- Global scheduling
- Semi-partitioned scheduling
- Partitioned scheduling

Partitioned scheduling algorithms, which is the focus of this paper, includes the following [10]:

- Partitioned Earliest Deadline First (P-EDF)
- Partitioned Rate Monotonic Scheduling (P-RMS)
- Partitioned Deadline Monotonic Scheduling (P-DMS)

Partitioned EDF, RMS, and DMS, all are extensions of these original single-core algorithms that have been adjusted and modified in a way that they can be applied to multi-core algorithms [11]. Furthermore, due to the fact that these algorithms must run on multi-core systems, the original single-core EDF, RMS, or DMS run on a each core independently, depending on whether partitioned EDF, RMS, DMS is selected to run on the multi-core system [11]. What this means is that each core has its own ready-queue for storing tasks to schedule, and its own scheduling algorithm [11]. In addition, one of the key differences between a partitioned scheduling algorithm and a global scheduling algorithm is that a task that is initially assigned to a core cannot migrate to another core [10]. Therefore, tasks assigned to cores by partitioned EDF, RMS, and DMS will loop back and forth between being in the ready-queue and being executed in the core they were assigned in until they are executed. We will now go into the details of each these partitioned scheduing algorithms.

II. Partitioned Earliest Deadline First (P-EDF)

A. Overview

Partitioned Earliest Deadline First is a scheduling algorithm derived from the single-core, preemptive scheduling algorithm, 'Earliest Deadline First' to be used now on multi-core systems [11]. For EDF, upon arrival of any task, if a given task in the ready-queue has a deadline that is earlier than that of the currently executed task, then the task with the earlier deadline will switch places with the originally executing task and have the CPU begin its execution instead. This can be best illustrated with an example.

Suppose we have a set of periodic tasks T that all arrive at $t = 0$ with deadlines equal to next periodic time values, as

$$T = \{t_1, t_2, t_3\}$$

a set of periods P for each task as

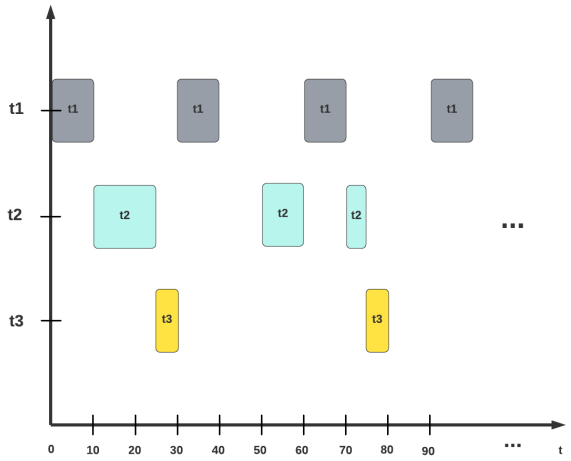
$$P = \{p_1, p_2, p_3\} \quad \Big| \quad p_1 = 30, p_2 = 50, p_3 = 60$$

a set of execution times E for each task as

$$E = \{e_1, e_2, e_3\} \quad \Big| \quad e_1 = 10, e_2 = 15, e_3 = 5$$

Figure 1 shows a gantt chart describing how tasks in T can scheduled with respect to periods in P, and execution times in E:

Fig. 1. Gantt chart of tasks T scheduled according to Earliest Deadline First (EDF), with periods P and execution times E



III. Implementation

A. Tasks and cores initialization

For the implementation described in this paper, the following table describes the periods, burst times, and sizes of the tasks considered:

Task I.D	Period	Burst time	Size in KB
1	5	2	100
2	8	3	120
3	0	0	125
4	0	0	130
5	0	0	132
6	0	0	142
7	0	0	148
8	0	0	150

An example of a 4-core system is considered in this implementation, and the free-memory size of each core is described in the table below:

Core I.D	Total free memory in KB
Core 1	224
Core 2	270
Core 3	280
Core 4	300

B. Tasks and cores allocation

Tasks are allocated to cores based on the first fit algorithm. An advantage of using first fit algorithm is the reduced time complexity of assigning a task to a core. The table below describes the task-to-core assignment scheme:

Core I.D	Tasks allocated
Core 1	Task 1 and 2
Core 2	Task 3 and 4
Core 3	Task 5 and 6
Core 4	Task 7 and 8

C. Uppaal model code

```
int core1_t = 1;
int core2_t = 0;

const int task1_period = 5;
const int task2_period = 8;

const int task1_burstTime = 2;
const int task2_burstTime = 3;

int task1_interruptedTime = 0;
int task2_interruptedTime = 0;

int core1_readyQueue[4] = {0, 0, 0, 0};

int task1_t = 0;
int task2_t = 0;
```

The code above describes the initializations necessary for implementation of EDF in Uppaal. The variable $core1_t$ describes the pseudo-clock of core 1. Then, the periods and burst times of tasks 1 and 2 are initialized as described in the table in subsection A. A ready queue is also defined as an array that tries to simulate an actual ready queue used by schedulers. If the element in index 0 is a 1, then task 1 is in the ready queue; if the element in index 1 is a 1, then task 2 is in the ready queue, and so on. Finally, the variables $task1_t$ and $task2_t$ contain the measured execution time when a task begins being executed, and resets to 0 when finished.

```
void replaceCurrentlyExecutingTaskWith
(int taskNum){

    if(taskNum == 1){

        task2_interruptedTime = task2_t;
        core1_t = core1_t + task2_t;
        core1_readyQueue[1] = 1;
        core1_readyQueue[0] = 0;

    } else if(taskNum == 2){

        task1_interruptedTime = task1_t;
        core1_t = core1_t + task1_t;
        core1_readyQueue[0] = 1;
        core1_readyQueue[1] = 0;

    }

}

void concludeTaskExecution(int taskNum){

    if(taskNum == 1){

        task1_t = 0;
        core1_t = core1_t + (task1_burstTime -
        task1_interruptedTime);
        task1_interruptedTime = 0;
        core1_readyQueue[0] = 0;


    } else if(taskNum == 2){

        task2_t = 0;
        core1_t = core1_t + (task2_burstTime -
        task2_interruptedTime);
        task2_interruptedTime = 0;
        core1_readyQueue[1] = 0;

    }


}
```

The "replaceCurrentlyExecutingTaskWith" function swaps an incoming task with the task currently executing. This is reflected by assigning a 1 or 0 to the relevant index based on the argument provided to the function. The "concludeTaskExecution" function resets measured execution times for a given task based on the argument, and it adds the achieved burst time amount of the task to the core1 clock to reflect the passage of real-time. Finally, any interrupted time marks are also reset, and the task is removed from the ready queue.

```
int findTaskInReadyQueue(){

    int i = 0;
    for(i : int[0, 3]){

        if(core1_readyQueue[i] == 1){
            return i + 1;
        }

    }
```

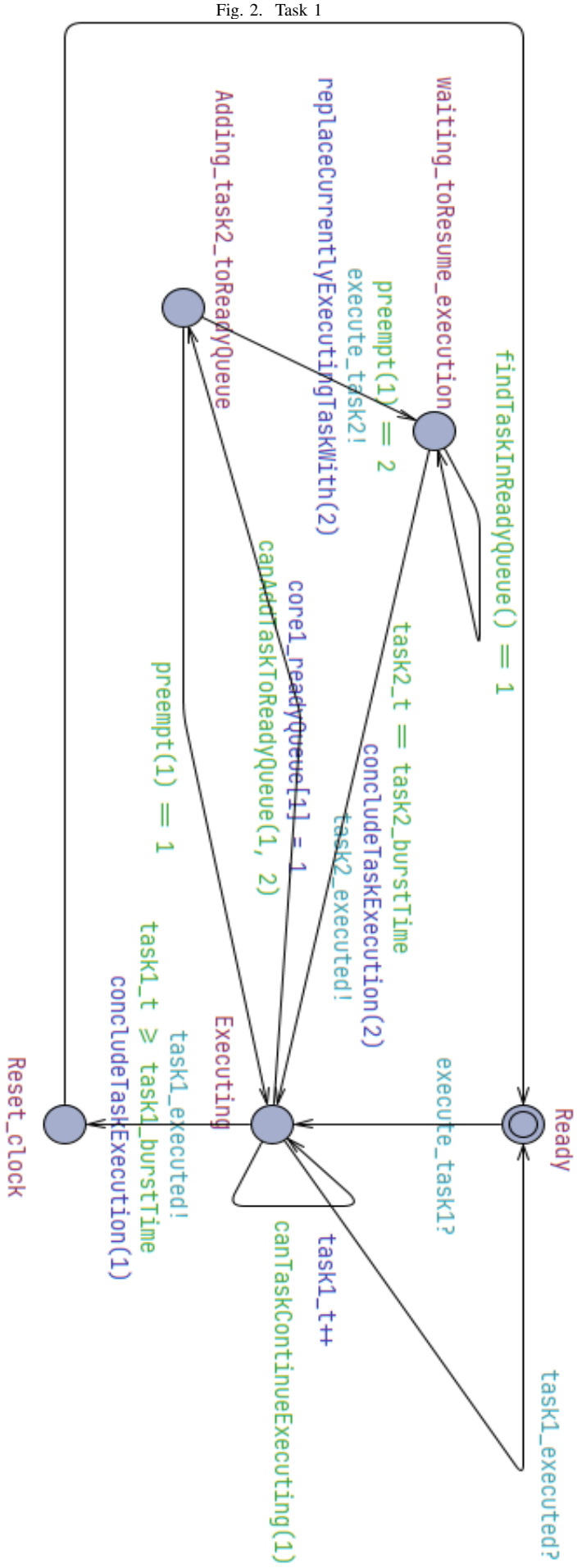
```
    return -1;

}


int preempt(int currentlyExecutingTask){

    int i;

    if(currentlyExecutingTask == 1){
        i = core1_t + task1_t;
    } else if(currentlyExecutingTask
    == 2) {
        i = core1_t + task2_t;
    }

    while(true){

        i = i + 1;

        if(i % task1_period == 0){

            return 1;

        } else if(i % task2_period == 0){

            return 2;

        }

    }

    return -1;

}
```

The ”findTaskInReadyQueue” function simply iterates over the ready queue, and upon finding a value of 1 at an index, which indicates the associated task being in the ready queue, returns the index value + 1. A 1 is added in the return statement because array indices start at 0, and the return value works with the actual task number, e.g: 1 for task 1, and 2 for task 2. The ”preempt” function first determines the real-time based on the task currently being executed, and starts incrementing a copy of this real-time value. After every increment, it checks whether this incremented time is a deadline of a task. This way, the earliest deadline task would be found first and the function would return its task number.

```
bool canTaskContinueExecuting(int taskNum){

    if(taskNum == 1){

        return task1_t < task1_burstTime &&
        !canAddTaskToReadyQueue(1, 2);

    } else if(taskNum == 2){

        return task2_t < task2_burstTime &&
        !canAddTaskToReadyQueue(2, 1);

    }

    return -1;

}

bool canAddTaskToReadyQueue(int executingTask,
int incomingTask){

    if(executingTask == 1 && incomingTask
    == 2){

        return (core1_t + task1_t) %
        task2_period == 0 &&
        findTaskInReadyQueue() != 2 &&
        task1_t < task1_burstTime;

    } else if(executingTask == 2 &&
    incomingTask == 1){

        return (core1_t + task2_t)
        % task1_period == 0 &&
        findTaskInReadyQueue() != 1 &&
        task2_t < task2_burstTime;

    } else if(executingTask == -1 &&
    incomingTask == 1){

        return core1_t % task1_period
        == 0 && findTaskInReadyQueue() != 1;

    } else if(executingTask == -1 && i
    ncomingTask == 2){

        return core1_t % task2_period == 0
        && findTaskInReadyQueue() != 2;

    }

    return -1;

}

bool canCoreClockIncrementIdle(){

    return core1_t % task1_period != 0 &&
    core1_t % task2_period != 0 &&
    findTaskInReadyQueue() == -1;

}
```

The ”canTaskContinueExecuting” function allows a task to increment its measured execution time value. The only way a task is allowed to execute is if it still has not reached its burst time, and if there is no other incoming task to be added to the ready queue, so that preemption can be considered. The ”canAddTaskToReadyQueue” function is used by both the Core and Tasks processes. A first argument of -1 is used in the Core process to imply that no task is currently being executed, and a value of 1 or 2 is used by the Task processes to indicate the currently executing task. The second argument of 1 or 2 indicates the the task number to look out for; checking if the current time is an arrival time of a potentially incoming task. Note that arrival times are simply multiples of the periods of tasks.

Fig. 2. Task 1



REFERENCES

[1] Edwin Robledo, What Was the First Computer?, 8th November 2022
[2] GeeksForGeeks, Advantages and Disadvantages of Multicore Processors, 15th April 2023
[3] Kumar, N., Vidyarthi, D.P., A novel energy-efficient scheduling model for multi-core systems, Cluster Comput 24, 643–666 (2021).
[4] Zhuo Chen, and Diana Marculescu, Fellow, IEEE, Task Scheduling for Heterogeneous Multicore System
[5] Behnaz Pourmohseni, Fedor Smirnov, Stefan Wildermann, Jürgen Teich, Real-Time Task Migration for Dynamic Resource Management in Many-Core Systems, Workshop on Next Generation Real-Time Embedded Systems, 20th January 2020
[6] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment Scheduling Algorithms for Multiprogramming,” J. Assoc. Comput. Mach., vol. 20, no. 1, pp. 46–61, 1973
[7] Xu, J., Shi, H. & Chen, Y. Efficient tasks scheduling in multicore systems integrated with hardware accelerators. J Supercomput 79, 7244–7271 (2023)
[8] M. Lordwin Cecil Prabhaker, R. Saravana Ram Real-Time Task Schedulers for a High-Performance Multi-Core System. Aut. Control Comp. Sci. 54, 291–301 (2020).
[9] Herlihy, M., & Shavit, N. (2012). The Art of Multiprocessor Programming, Revised Reprint. Elsevier
[10] Artem Burmyakov, Borislav Nikolić, An exact comparison of global, partitioned, and semi-partitioned fixed-priority real-time multiprocessor schedulers, 15th October 2021
[11] Baruah, S. Partitioned EDF scheduling: a closer look. Real-Time Syst 49, 715–729 (2013)
[12] Subham Datta, Scheduling: Earliest Deadline First, 18th March 2024

Fig. 3. Task 2

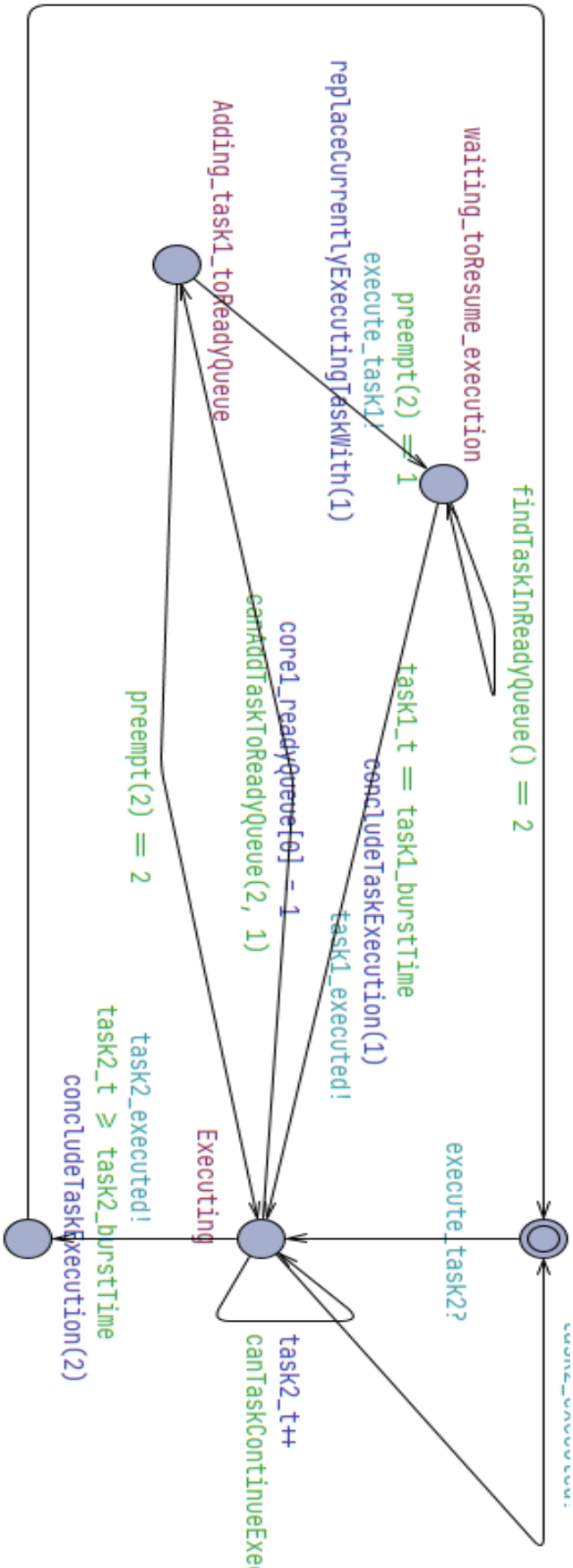


Fig. 4. Core