DYNAMIC SCHEDULING FOR MULTI-CORE ARCHITECTURES

Pavana Krishna Bhat
Saranya Shanmugam
Vidhya S Chandrashekar
Table of Contents

Introduction .................................................................................................................3
Problem Definition ........................................................................................................3
Relevance of this project to Operating system ...........................................................3
Why other approach is not good .................................................................................3
Why our approach is better .........................................................................................3
Statement of the problem ............................................................................................3
Scope of the project .....................................................................................................3
Theoretical basis and literature review .......................................................................4
Theoretical background of the problem .......................................................................4
Related research to solve the problem .........................................................................4
Advantages/disadvantages of those research ...............................................................4
Proposed solution/why this is better from other solutions ...........................................4
Hypothesis ....................................................................................................................4
Methodology ...............................................................................................................5
Input generation/Collection ........................................................................................5
How to solve the problem ............................................................................................5
How to generate the output ..........................................................................................6
How to test against hypothesis ....................................................................................6
Implementation ............................................................................................................6
Data analysis and discussion .......................................................................................15
Test against Hypothesis ..............................................................................................16
Abnormal case explanation .........................................................................................17
Statistic regression .....................................................................................................17
Conclusions and recommendations ............................................................................17
Bibliography ...............................................................................................................18
Appendices ................................................................................................................18
INTRODUCTION

PROBLEM DEFINITION

In Multi-core systems, all the cores necessarily use shared memory architecture. There is a definite need for scheduling algorithms suitable for shared memory architecture to increase the efficiency of multi-core processors in presence of multiple tasks within an application. Most of the proposed scheduling algorithms for multi-core processors concentrate on scheduling tasks that are independent of each other. This means that execution of one task does not affect or is not dependent on the result of other tasks and they may execute concurrently. To utilize multi-core processors more efficiently for embedded applications where only one single application executes at any time, the application should be divided into subtasks. This demands a scheduling algorithm that can be efficient enough to exploit the multi core architecture to achieve an optimal schedule in terms of time of execution and processor utilization.

RELEVANCE OF THIS PROJECT TO OPERATING SYSTEM

Scheduling is one of the major tasks performed by the operating systems. The project involves implementation of a scheduling algorithm for multi core systems. Implementation requires usage of threads, semaphores and mutexes.

WHY OTHER APPROACH IS NOT GOOD

Round robin scheduling is inefficient in scheduling multiple dependent tasks in terms of higher turnaround times and context switches.

WHY OUR APPROACH IS BETTER

The design gives higher priority to those tasks which resolve more dependencies and thus giving low turnaround times and context switches.

STATEMENT OF THE PROBLEM

Solving problems of higher turnaround time and context switches for dependent tasks scheduled by traditional scheduling algorithm - Round Robin scheduling.

SCOPE OF THE PROJECT

Implementation of scheduler for dependent tasks in multi core system.
THEORETICAL BASIS AND LITERATURE REVIEW

THEORETICAL BACKGROUND OF THE PROBLEM

The availability of multicore has forced software programmers to change the way they think and write their applications. Unfortunately, the applications written so far are sequential in nature. We can extract the inherent parallelism in such applications to exploit the available multi core architecture.

RELATED RESEARCH TO SOLVE THE PROBLEM

Research was done for scheduling designs for homogeneous/heterogeneous multicore architectures. Different kind of scheduling designs are studied. Also, studied the importance of parallel programming.

ADVANTAGES/DISADVANTAGES OF THOSE RESEARCH

Advantage:

The papers discussed efficient scheduling algorithms for NUMA architectures.

Disadvantage:

The papers did not consider the dependency factor existing between multiple tasks to decide on scheduling order.

PROPOSED SOLUTION/WHY THIS IS BETTER FROM OTHER SOLUTIONS

Design and implementation of a scheduler for multi core systems considering the task dependency and execution time. Most of the conventional schedulers for multi-core processors do not concentrate on task dependencies for scheduling decisions. The proposed scheduler gives priority to the tasks that resolve more dependencies and hence make sure that the updates to the ready-to-execute tasks list are done accordingly.

HYPOTHESIS

1. If the applications have more dependent tasks, then the proposed scheduler gives better performance in terms of turnaround time for dependent tasks.
2. The proposed scheduler assumes that the system has symmetric multiprocessor architecture.
**METHODOLOGY**

**INPUT GENERATION/COLLECTION**

Data analysis techniques are to be used to find the dependent tasks in an application. In this project, we have come up with numerous test cases involving multiple dependent tasks.

The scheduler requires following information for each task to make scheduling decisions.

1. **Task Identifier, Tid**: Unique identifier to each task.
2. **Execution Time, Tx**: Time taken to execute by each task Tid.
3. **Number of tasks Tid is dependent upon, Nparent**: Tid moves to ready-to-execute only after the execution of ‘Nparent’ tasks.
4. **Number of tasks dependent on the task Tid, Nchild**: Number of dependencies to be resolved by Tid, before moving them to ready-to-execute status.
5. **Dependency Task Ids, Tchild**: List of the dependency task Ids.
6. **Dependency Task execution times, TchildStartTime**: List of dependency tasks Tchild, start times.

**HOW TO SOLVE THE PROBLEM**

**Algorithm**

1. Read the input mentioned above.
2. Create a task table in the shared memory.
3. Task table consists of task identifier, task status, execution time, dependency information, priority etc.
4. Each core will have a scheduler instance.
5. Scheduler thread parses the task table to calculate priority and status for each task.
6. Scheduler thread gets a ‘ready-to-execute’ task with the highest priority to execute. If multiple tasks have the same priority, task with shortest execution time is selected for execution. If the task has children, set the time-slice=minimum (TchildStartTime) and set time-slice = constant (fixed internally) when task has no children. Change the task status to ‘running’. Schedule the corresponding task to execute for the time-slice selected. The task starts execution.
7. Task will execute for the selected time-slice. The task is preempted after executing for selected ‘time-slice’.
8. Scheduler thread updates the task table
   a. Update the execution time.
   b. Update Nchild, Tchild, TchildStartTime, Nparent fields if applicable.
   c. If the task Tid is completed, change the task status to completion state.
9. Repeat steps from step 5 till all tasks reach completion.

**Note:**

1. For simulating dual core, 2 scheduler threads run the above algorithm.
2. Task table is shared between the scheduler threads and hence needs protected access.

**Language**

C

**Tools**

Gdb

---

**HOW TO GENERATE THE OUTPUT**

Compile the program using “make” command. Execute the program using “./scheduler”.

Read the input from stdin, and output will be generated in stdout.

**HOW TO TEST AGAINST HYPOTHESIS**

The generated output has turnaround time, waiting time and number of context switch for each task. This can be compared with manually calculated results obtained using round-robin scheduling algorithm.

**IMPLEMENTATION**

**Code**

Main:

```c
/* Main function*/
int main()
{
    pthread_t scheduler_1,scheduler_2;
    int retValue=0;
    int timeSlice=0;
    void *exit_status;
    int threadId1=0,threadId2=0;
    /*initialise the mutex*/
    pthread_mutex_init(&lock,NULL);
    memset(&tds,0, sizeof(tds));
    /*initialise the outputDataStructure*/
    memset(&ods,0, sizeof(ods));
    /* Read the task inputs from the stdin*/
    retValue = ReadTaskInputs();
    if(retValue != SUCCESS)
    {
        printf("nError in reading the Inputs");
    }
    retValue = ScheduleTasks(scheduler_1,scheduler_2);
    if(retValue != SUCCESS)
    {
        printf("nError in scheduling the tasks");
    }
    retValue = GenerateOutput(&ods);
    if(retValue != SUCCESS)
    {
        printf("nError in generating the output");
    }
}
```

return 1;
/*create scheduler1 and 2*/
threadId1 = 1;
retValue = pthread_create(&scheduler_1,NULL,TaskScheduler,&threadId1);
if(retValue != SUCCESS)
{
    printf("\nError in creating thread 1");
    return 1;
}

threadId2 = 2;
retValue = pthread_create(&scheduler_2,NULL,TaskScheduler,&threadId2);
if(retValue != SUCCESS)
{
    printf("\nError in creating thread 2");
    return 1;
}
/* wait till threads(scheduler1/2) are exited*/
retValue = pthread_join(scheduler_1,&exit_status);
if(retValue != SUCCESS)
{
    printf("\n Error in joining thread");
    return 1;
}
retValue = pthread_join(scheduler_2,&exit_status);
if(retValue != SUCCESS)
{
    printf("\n Error in joining thread");
    return 1;
}
calculateWaitTime();
calculateContextSwitches();
printOutputTDS();
/*destroy the mutex*/
pthread_mutex_destroy(&lock);
return 0;
}

Scheduler code:
void  *TaskScheduler(void* arg)
{
    int i = 0,idleTime = 0;
    float timeSlice = 0.0;
    int* threadId;
    threadId = (int*)arg;

    /* Schedulers to run till there are runnable tasks*/
    while(nActiveTasks > 0)
    {
    /* Obtain the lock to access the table*/
    pthread_mutex_lock(&lock);
    /*Timeslice should private. Each thread should modify or access its own timeSlice */
    if(*threadId == 1)
    {
    updateTDS(*threadId,&timeSlice_schd1);
    if(strlen(curTaskId_schd1) == 0)
/*When thread2 is busy and there is no active task to run by thread1, thread1 has to sleep until thread2 releases a new task=> thread1 is idle till that time*/
idleTime = (waitTime_schd2 + timeSlice_schd2) - waitTime_schd1;
usleep(idleTime * 1000);
waitTime_schd1 = idleTime + waitTime_schd1;

/*Abnormal test case: When there are tasks which can never reach ready state due to mutual dependency threads will be active doing nothing => idle*/
if(idleTime == 0)
{
    thread1Idle = 1;
}
else if(*threadId == 2)
{
    updateTDS(*threadId,&timeSlice_schd2);
    if(strlen(curTaskId_schd2) == 0)
    {
        /*When thread1 is busy and there is no active task to run by thread2, thread2 has to sleep until thread1 releases a new task=> thread2 is idle till that time*/
        idleTime = (waitTime_schd1 + timeSlice_schd1) - waitTime_schd2;
        usleep(idleTime * 1000);
        waitTime_schd2 = idleTime + waitTime_schd2;
        /*Abnormal test case: When there are tasks which can never reach ready state due to mutual dependency threads will be active doing nothing => idle*/
        if(idleTime == 0)
        {
            thread2Idle = 1;
        }
    }
}
    timeSlice = timeSlice_schd2;
} /* Release the lock */
pthread_mutex_unlock(&lock);
/* Run the selected task */
if((thread1Idle == 1) && (thread2Idle == 1))
{
    break;
}
runTask(timeSlice);

/* Pthreads to exit once all the tasks reach completion state*/
if((nActiveTasks <= 0) || ((thread1Idle == 1) && (thread2Idle == 1)))
{
    printf("\nNo tasks to run!!!!\n");
    pthread_exit(NULL);
}
return NULL;
}

Task Execution:
/* run the scheduled task for fixed time slice: simulated using sleep()*/
void runTask(float timeSlice)
{
    int time = timeSlice*1000;
    usleep(time);
}
/* update the task data structure*/
void updateTDS(int threadId, float *timeSlice)
{  
  int i = 0, j = 0;
  float waitTime = 0.0;
  char childTid[10] = {}, taskId[10] = {};
  char currTaskId[10] = {};

  /* Each thread should access its own last executed task and wait time*/
  if(threadId == 1)
  {
    strcpy(currTaskId, curTaskId_schd1);
  }
  else if(threadId == 2)
  {
    strcpy(currTaskId, curTaskId_schd2);
  }

  /* update the TDS table for its own last executed task*/
  for(i = 0; i < nTasks; i++)
  {
    if(strcmp(tds[i].Tid, currTaskId) == 0)
    {
      /* update the Tx (execution time) of the last executed task*/
      /* update the task status = Ready if Tx > 0*/
      if(tds[i].Tx > 0)
      {
        tds[i].Tx = tds[i].Tx - *timeSlice;
        tds[i].taskStatus = Ready;
      }
      /* Update the task status = Completed if Tx = 0*/
      if(tds[i].Tx == 0)
      {
        setTaskStatus(currTaskId, Completed);
      }
      /* Update the dependent child info of the last executed task*/
      if(tds[i].Nchild > 0)
      {
        tds[i].Nchild--;
        strcpy(childTid, tds[i].Tchild[0]);
        for(j = 0; j < 256; j++)
        {
          strncpy(tds[i].Tchild[j], tds[i].Tchild[j + 1], 10);
          tds[i].TchildStartTime[j] = tds[i].TchildStartTime[j + 1];
        }
      }
    }
  }

  /* updating turn around time for a particular task executed by particular thread */
  calculateTurnAroundTime(threadId);
  
  /* updating child(of last executed task)'s Nparent */
  for(i = 0; i < nTasks; i++)
  {
    if(strcmp(childTid, tds[i].Tid) == 0 && (tds[i].Nparent > 0))
    {
      tds[i].Nparent--;
    }
  }
  calculatePriority();
  
  /* change the status to READY*/
  setTaskStatusForTable();
  
  /* taskId is the output parameter*/
  selectTaskId(taskId);
  
  /* set the task status = RUNNING for the selected task*/
  setTaskStatus(taskId, Running);
  
  /* Get the time slice of the selected taskId*/
  *timeSlice = getTimeSlice(taskId);
  
  /* Each thread should access its own currently selected task*/
  if(threadId == 1)
Design document and flowchart

1. Read the task inputs for scheduling.

   The following data structure is used to collect task input:

   ```
   typedef struct inputStruct
   {
       char Tid[10];
       float Tx;
       uint32_t Nparent;
       uint32_t Nchild;
       char* Tchild;
       char* TchildStartTime;
   } inputStruct_t;
   ```

2. Create the task table in memory, which is to be shared by two scheduler threads.

3. Task table consists of task identifier, task status, execution time, dependency information, priority etc.

   ```
   typedef struct TaskDataStructure
   {
       char Tid[10];
       float Tx;
       uint32_t Nparent;
       uint32_t Nchild;
       char* Tchild[256];
       float* TchildStartTime;
       float taskTimeSlice[1024];
       float priority;
       eTaskStatus taskStatus;
   } TaskDataStruture_t;
   ```
4. Dual core is simulated using two threads and each core will have a scheduler instance.

5. Scheduler thread parses the task table to calculate priority and status for each task. If the task has children, then its priority is calculated based on Nchild, Tx and the maximum execution time among all the tasks in the task table.
   \[ \text{Priority} = Nchild + \frac{Tx}{\text{Max}(Tx)} \]
   If the task has no children, then its priority is calculated based on its execution time and the maximum execution time among all the tasks in the task table.
   \[ \text{Priority} = \frac{Tx}{\text{Max}(Tx)} \]

6. Along with the priority, the task status is set. If the task has parents, set the status as 'Not Ready'. If the task has no parents, set the status as 'ready-to-execute'.

7. Scheduler thread checks for 'ready-to-execute' tasks from the task table. If the table has 'ready-to-execute' tasks, then it picks the task with the highest priority to execute. If multiple tasks have the same priority, any one of the tasks will be selected for execution. If the task has children, set the time-slice=minimum (TchildStartTime) and set time-slice = constant (fixed internally) when task has no children. Change the task status to 'running'. Schedule the corresponding task to execute for the time-slice selected. If the table has no 'ready-to-execute' task and the other scheduler thread is busy, this scheduler thread will go to sleep until there is any 'ready-to-execute' task to run.

8. Task will execute for the selected time-slice. Task execution is simulated by making the thread sleep for the selected time-slice.

   ```c
   /* run the scheduled task for fixed time slice:simulated using sleep()*/
   void runTask(float timeSlice)
   {
       int time = timeSlice*1000;
       usleep(time);
   }
   ```

9. After the task is executed for the selected time-slice, task is preempted.

10. Scheduler thread updates the task table
    a. Update the execution time.
    b. Update Nchild, Tchild, TchildStartTime, Nparent fields if applicable.
    c. If the task Tid is completed, change the task status to completed and set the priority to -1

11. Repeat steps from step 5 till all tasks reach completion.

12. After all tasks are completed, Turn-around time, waiting time and number of context switches for each task is calculated as below:

    Turn-around time = wait-time + execution time
For dependant tasks,
Context Switch = (Nchild - 1) + (Tx - Max(TchildStartTime)/Fixed_time_slice)

For independent tasks,
Context Switch = (Tx/Fixed_time_slice) - 1

Note:

1. For simulating dual core, 2 scheduler threads run the above algorithm.
2. Task table is shared between the scheduler threads and hence needs protected access.
Start

Read Input

Create task table

Create two threads

Instantiate scheduler instance for each thread

Each thread will access the shared task table by locking the mutex

The thread which acquired the mutex lock, picks the 'ready-to-execute' task with highest priority

If Nchild > 0

Yes

A

Yes

B
Time slice for the task is equal to fixed time slice calculated by data analysis

Scheduler changes the task status to ‘Running’ and runs the selected task for the specified time slice

If total runtime of task = Execution time

Change the task status to ‘Completed’ and update the task table

Preempt the task and update the task details in the task table

Repeat the steps until all the tasks are executed

End
### DATA ANALYSIS AND DISCUSSION

**Output Generation:**

**Sample Input :**

<table>
<thead>
<tr>
<th>TaskId</th>
<th>Tx</th>
<th>Nparent</th>
<th>Nchild</th>
<th>Tchild</th>
<th>TchildStartTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>250</td>
<td>0</td>
<td>2</td>
<td>T1,T2</td>
<td>100,150</td>
</tr>
<tr>
<td>T1</td>
<td>400</td>
<td>1</td>
<td>1</td>
<td>T2</td>
<td>200</td>
</tr>
<tr>
<td>T2</td>
<td>600</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>600</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tasks T0, T5 and T6 are independent tasks and tasks T1,T2 are dependant tasks.

**Output :**

**Task table at the beginning of the scheduling :**

<table>
<thead>
<tr>
<th>TaskId</th>
<th>Tx</th>
<th>Nparent</th>
<th>Nchild</th>
<th>Tchild</th>
<th>TchildStartTime</th>
<th>Priority</th>
<th>Task Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>250</td>
<td>0</td>
<td>2</td>
<td>T1,T2</td>
<td>100,150</td>
<td>2.58</td>
<td>1</td>
</tr>
<tr>
<td>T1</td>
<td>400</td>
<td>1</td>
<td>1</td>
<td>T2</td>
<td>200</td>
<td>1.33</td>
<td>-1</td>
</tr>
<tr>
<td>T2</td>
<td>600</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td>0.00</td>
<td>-1</td>
</tr>
<tr>
<td>T4</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0.17</td>
<td>0</td>
</tr>
<tr>
<td>T5</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0.83</td>
<td>0</td>
</tr>
<tr>
<td>T6</td>
<td>600</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0.00</td>
<td>0</td>
</tr>
</tbody>
</table>

Task status and priority are updated. Currently as shown above, the independent task with highest priority (which resolves maximum dependancies) T0 is picked for execution and its status is changed ‘Running’.

**Task table at the End of the scheduling :**

<table>
<thead>
<tr>
<th>TaskId</th>
<th>Tx</th>
<th>Nparent</th>
<th>Nchild</th>
<th>Tchild</th>
<th>TchildStartTime</th>
<th>Priority</th>
<th>Task Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>T4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>T5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>T6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>-1</td>
<td>2</td>
</tr>
</tbody>
</table>
All the tasks T0, T1, T2, T4, T5, T6 are executed to completion as observed in the Task Status column of the above table.

Output Table:

<table>
<thead>
<tr>
<th>TaskId</th>
<th>Execution Time</th>
<th>Turn-around time</th>
<th>Waiting time</th>
<th>Context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>250</td>
<td>250</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>T1</td>
<td>400</td>
<td>500</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>T2</td>
<td>600</td>
<td>1100</td>
<td>500</td>
<td>5</td>
</tr>
<tr>
<td>T4</td>
<td>500</td>
<td>750</td>
<td>250</td>
<td>4</td>
</tr>
<tr>
<td>T5</td>
<td>100</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T6</td>
<td>600</td>
<td>1350</td>
<td>750</td>
<td>5</td>
</tr>
</tbody>
</table>

Final output is shown above capturing the total turnaround time, waiting time and number of context switches for all the tasks.

TEST AGAINST HYPOTHESIS

The output obtained using the algorithm is compared with theoretically calculated round robin results. The algorithm should give smaller wait time/smaller turnaround time for the dependent tasks.

Dynamic Scheduler for dependent tasks in dual core

<table>
<thead>
<tr>
<th>TaskId</th>
<th>Execution Time</th>
<th>Turn-around time</th>
<th>Waiting time</th>
<th>Context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>250</td>
<td>250</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>T1</td>
<td>400</td>
<td>500</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>T2</td>
<td>600</td>
<td>1100</td>
<td>500</td>
<td>5</td>
</tr>
<tr>
<td>T4</td>
<td>500</td>
<td>750</td>
<td>250</td>
<td>4</td>
</tr>
<tr>
<td>T5</td>
<td>100</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T6</td>
<td>600</td>
<td>1350</td>
<td>750</td>
<td>5</td>
</tr>
</tbody>
</table>

Round Robin with fixed Time slice (100 ms)

<table>
<thead>
<tr>
<th>TaskId</th>
<th>Execution Time</th>
<th>Turn-around time</th>
<th>Waiting time</th>
<th>Context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>250</td>
<td>650</td>
<td>400</td>
<td>2</td>
</tr>
<tr>
<td>T1</td>
<td>400</td>
<td>1200</td>
<td>800</td>
<td>4</td>
</tr>
<tr>
<td>T2</td>
<td>600</td>
<td>1700</td>
<td>1100</td>
<td>9</td>
</tr>
<tr>
<td>T4</td>
<td>500</td>
<td>1300</td>
<td>800</td>
<td>4</td>
</tr>
<tr>
<td>T5</td>
<td>100</td>
<td>400</td>
<td>300</td>
<td>0</td>
</tr>
<tr>
<td>T6</td>
<td>600</td>
<td>1350</td>
<td>750</td>
<td>5</td>
</tr>
</tbody>
</table>
As observed from the above tables, turnaround time, waiting time and number of context switches obtained using our project are smaller compared to traditional round robin design.

**ABNORMAL CASE EXPLANATION**

<table>
<thead>
<tr>
<th>TaskId</th>
<th>Tx</th>
<th>Nparent</th>
<th>Nchild</th>
<th>Tchild</th>
<th>TchildStartTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>250</td>
<td>1</td>
<td>1</td>
<td>T1</td>
<td>50</td>
</tr>
<tr>
<td>T1</td>
<td>400</td>
<td>1</td>
<td>1</td>
<td>T0</td>
<td>200</td>
</tr>
</tbody>
</table>

When the tasks are mutually dependent on each other, the dependency cannot be resolved. The tasks can never reach ‘Ready’ state and therefore cannot reach completion state.

**STATISTIC REGRESSION**

The time slice for the independent tasks are set based on the execution times of the tasks present in an application. Data analysis techniques can be used. The context switch count of the task is dependant on the time slice chosen.

If time slice chosen is less, the context switch count increases and vice versa.

**CONCLUSIONS AND RECOMMENDATIONS**

**Summary and conclusions**

The scheduler designed and implemented in this project increased the utilization of dual core processor. This method gives priority to the tasks which resolve more dependencies and hence makes sure that the updates to the ‘ready-to-execute’ tasks in the task table are done accordingly.

**Recommendations for future studies**

1. This design is easily extendable to heterogeneous multi core.
2. The design may affect in terms of additional wait times for cores since it involves accessing shared task data table through mutexes. As the number of cores increase, the scalability of the design is limited but lock free mechanisms like transactional memory.
BIBLIOGRAPHY

Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures
http://dl.acm.org/citation.cfm?id=1362694

Thread Scheduling for Multi-Core Platforms
http://static.usenix.org/event/hotos07/tech/full_papers/rajagopalan/rajagopalan.pdf

HASS: A Scheduler for Heterogeneous Multicore Systems
http://dl.acm.org/citation.cfm?id=1531804

Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures
http://dl.acm.org/citation.cfm?id=1128029

Improving the Static Real-Time Scheduling on Multicore
http://dl.acm.org/citation.cfm?id=1900008.1900148&coll=DL&dl=ACM&CFID=338201848&CFTOKEN=27341977

parallel task scheduling on multicore platform
http://dl.acm.org/citation.cfm?id=1279711.1279713&coll=DL&dl=ACM&CFID=338201848&CFTOKEN=27341977

APPENDICES

1. Turnaround time is the total time taken between the submission of a program/process/thread/task (Linux) for execution and the return of the complete output to the customer/user

2. Full code

```
#include <stdio.h>
#include <pthread.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#define FIXED_TIME_SLICE 100
#define SUCCESS 0

int createTaskDataStructure(int index);
int copyChildInfo(int index);
float maxExectionTime();
void calculatePriority();
void setTaskStatus(char* taskId,int status);
void *TaskScheduler(void* arg);
```
void selectTaskId(char* taskId);
float getTimeSlice(char* taskId);
void updateTDS(int threadId, float *timeSlice);
void calculateTurnAroundTime(int threadId);
void calculateWaitTime();
void calculateContextSwitches();
void runTask(float timeSlice);
void setTaskStatusForTable();
void printTDS();
void printOutputTDS();
int ReadTaskInputs();
void freeResources();

typedef enum ThreadId
{
    thread1 = 1,
    thread2
} eThreadId;

typedef enum TaskStatus
{
    NotReady = -1,
    Ready,
    Running,
    Completed
} eTaskStatus;

typedef struct inputStruct
{
    char Tid[10];
    float Tx;
    uint32_t Nparent;
    uint32_t Nchild;
    char* Tchild;
    char* TchildStartTime;
} inputStruct_t;

typedef struct TaskDataStructure
{
    char Tid[10];
    float Tx;
    uint32_t Nparent;
    uint32_t Nchild;
    char* Tchild[256];
    float* TchildStartTime;
    float taskTimeSlice[1024];
    float priority;
    eTaskStatus taskStatus;
} TaskDataStruture_t;

typedef struct outputDataStructure
{
    char Tid[10];
    float Tx, turnAroundTime, waitTime;
    uint32_t Nchild;
    float maxTchildTime;
    int contextSwitches;
} outputDataStructure_t;

inputStruct_t input;
TaskDataStruture_t tds[512];
outputDataStructure_t ods[512];

int nTasks=0,nActiveTasks=0;
static char curTaskId_schd1[10]="{}";
static char curTaskId_schd2[10]="{}";
static float timeSlice_schd1 = 0.0,timeSlice_schd2 = 0.0;
static float waitTime_schd1 = 0.0,waitTime_schd2 = 0.0;
pthread_mutex_t lock;
int thread1Idle = 0,thread2Idle = 0;

void *TaskScheduler(void* arg)
{
    int i = 0,idleTime = 0;
    float timeSlice = 0.0;
    int* threadId;
    threadId = (int *)arg;

    /* Schedulers to run till there are runnable tasks*/
    while(nActiveTasks > 0)
    {
        //printf("\n taskScheduler: ntasks: %d nActiveTasks=%d",nTas
        //printf("\n taskScheduler: ntasks: %d nActiveTasks=%d",nTas
        if(*threadId == 1)
        {
            updateTDS(*threadId,&timeSlice_schd1);
            if(strlen(curTaskId_schd1) == 0)
            {
                /*When thread2 is busy and there is no active task to run by thread1,thread1 has to sleep until thread2 releases a
                new task=>thread 1 is idle till that time*/
                idleTime = (waitTime_schd2 + timeSlice_schd2) - waitTime_schd1;
                usleep(idleTime * 1000);
            }
            timeSlice = timeSlice_schd1;
        }
        else if(*threadId == 2)
        {
            updateTDS(*threadId,&timeSlice_schd2);
            if(strlen(curTaskId_schd2) == 0)
            {
                /*When thread1 is busy and there is no active
                task to run by thread2,thread2 has to sleep until thread1 releases a
                new task=>thread 2 is idle till that time*/
                idleTime = (waitTime_schd1 + timeSlice_schd1) - waitTime_schd2;
                usleep(idleTime * 1000);
                waitTime_schd2 = idleTime + waitTime_schd2;
            }
            timeSlice = timeSlice_schd2;
        }
    }
}

void *TaskScheduler(void* arg)
{
    int i = 0,idleTime = 0;
    float timeSlice = 0.0;
    int* threadId;
    threadId = (int *)arg;

    /* Schedulers to run till there are runnable tasks*/
    while(nActiveTasks > 0)
    {
        //printf("\n taskScheduler: ntasks: %d nActiveTasks=%d",nTas
        //printf("\n taskScheduler: ntasks: %d nActiveTasks=%d",nTas
        if(*threadId == 1)
        {
            updateTDS(*threadId,&timeSlice_schd1);
            if(strlen(curTaskId_schd1) == 0)
            {
                /*When thread2 is busy and there is no active task to run by thread1,thread1 has to sleep until thread2 releases a
                new task=>thread 1 is idle till that time*/
                idleTime = (waitTime_schd2 + timeSlice_schd2) - waitTime_schd1;
                usleep(idleTime * 1000);
                waitTime_schd1 = idleTime + waitTime_schd1;
            }
            if(idleTime == 0)
            {
                thread1Idle = 1;
            }
            else if(*threadId == 2)
            {
                updateTDS(*threadId,&timeSlice_schd2);
                if(strlen(curTaskId_schd2) == 0)
                {
                    /*When thread1 is busy and there is no active task to run by thread2,thread2 has to sleep until thread1 releases a
                    new task=>thread 2 is idle till that time*/
                    idleTime = (waitTime_schd1 + timeSlice_schd1) - waitTime_schd2;
                    usleep(idleTime * 1000);
                    waitTime_schd2 = idleTime + waitTime_schd2;
                }
                if(idleTime == 0)
                {
                    thread2Idle = 1;
                }
            }
            else
            {
                timeSlice = timeSlice_schd1;
            }
        }
        else
        {
            timeSlice = timeSlice_schd2;
        }
    }
}
/* Release the lock */
    pthread_mutex_unlock(&lock);
/*Run the selected task */
    if((thread1Idle == 1)&&(thread2Idle == 1))
    {
        break;
    }
    runTask(timeSlice);

/*Pthreads to exit once all the tasks reach completion state*/
    if((nActiveTasks <= 0) || ((thread1Idle == 1) && (thread2Idle == 1)))
    {
        printf("\nNo tasks to run!!!!");
        pthread_exit(NULL);
    }
    return NULL;

/* timeslice = start time of the first child in the Tchild list(Tchild list will have ascending tchild start times now)*/
/* timeslice = FIXED_TIME_SLICE if the task has no child*/
float getTimeSlice(char* taskId)
{
    int i = 0;
    float timeSlice = 0.0;
    for(i = 0; i < nTasks; i++)
    {
        if(strcmp(taskId,tds[i].Tid) == 0)
        {
            /*Time slice calculation for dependent Tasks*/
            if(tds[i].Nchild > 0)
            {
                timeSlice = tds[i].taskTimeSlice[0];
            }
            else
            {
                /*Independent Tasks:If Tx < fixed time slice; timeslice is set to Tx*/
                if(tds[i].Tx < FIXED_TIME_SLICE)
                {
                    timeSlice = tds[i].Tx;
                }
                else
                {
                    timeSlice = FIXED_TIME_SLICE;
                }
            }
        }
        break;
    }
    return timeSlice;
}

/*taskId is output parameter; returns the selected taskId to be executed*/
void selectTaskId(char* taskId)
{
    int i,j;
    float selectedPriority = 0.0;
    for(i = 0; i < nTasks; i++)
    {
        /* select the first ready task*/
if(tds[i].taskStatus == Ready)
{
    selectedPriority = tds[i].priority;
    strcpy(taskId,tds[i].Tid);
    break;
}
#endif

/* select the highest priority task*/
for(j = i; j < nTasks; j++)
{
    if((tds[j].taskStatus == Ready) && (selectedPriority < tds[j].priority))
    {
        selectedPriority = tds[j].priority;
        strcpy(taskId,tds[j].Tid);
    }
}

/* run the scheduled task for fixed time slice:simulated using sleep()*/
void runTask(float timeSlice)
{
    int time = timeSlice * 1000;
    usleep(time);
}

void updateTDS(int threadId, float *timeSlice)
{
    int i = 0, j = 0;
    float waitTime = 0.0;
    char childTid[10] = {}, taskId[10] = {0};
    char currTaskId[10] = {0};

    /* Each thread should access its own last executed task and wait time*/
    if(threadId == 1)
    {
        strcpy(currTaskId, currTaskId_schd1);
    }
    else if(threadId == 2)
    {
        strcpy(currTaskId, currTaskId_schd2);
    }

    /* update the TDS table for its own last executed task*/
    for(i = 0; i < nTasks; i++)
    {
        if(strcmp(tds[i].Tid, currTaskId) == 0)
        {
            /* update the Tx (execution time) of the last executed task*/
            /* update the task status = Ready if Tx > 0*/
            if(tds[i].Tx > 0)
            {
                tds[i].Tx = tds[i].Tx - *timeSlice;
                tds[i].taskStatus = Ready;
            }
            /* Update teh task status = Completed if Tx = 0*/
            if(tds[i].Tx == 0)
            {
                setTaskStatus(currTaskId, Completed);
            }
            /* Update the dependent child info of the last executed task*/
            if(tds[i].Nchild > 0)
            {
                tds[i].Nchild--;
                strcpy(childTid, tds[i].Tchild[0]);
                for(j = 0; j < 256; j++)
                {
                    if(childTid[j] == '')
                    {
                        break;
                    }
                }
            }
        }
    }
}
/* updating turn around time for a particular task executed by particular thread */
calculateTurnAroundTime(threadId);
}

/* updating child(of last executed task)'s Nparent */
for(i = 0;i < nTasks;i++)
{
    if(strcmp(childTid,tds[i].Tid) == 0 && (tds[i].Nparent > 0))
    {
        tds[i].Nparent--;
    }
calculatePriority();
/*change the status to READY*/
setTaskStatusForTable();
/*taskId is the output parameter*/
selectTaskId(taskId);
/* set the task status = RUNNING for the selected task*/
setTaskStatus(taskId,Running);
/* Get the time slice of the selected taskId*/
*timeSlice = getTimeSlice(taskId);
/* Each thread should access its own currently selected task */
if(threadId == 1)
    strcpy(curTaskId_schd1,taskId);
else if(threadId == 2)
    strcpy(curTaskId_schd2,taskId);
if(strlen(taskId) > 0)
{
    printf("\n Thread%d picks the task %s",threadId,taskId);
    printTDS();
}
else if(strlen(taskId) == 0)
{
    if(nActiveTasks == 0)
    {
        printf("\n Thread%d completed the task %s",threadId,taskId);
        printTDS();
    }
    printf("\n Thread%d waiting for READY task\n",threadId);
}
}
/*Calculate and set wait time for each task */
void calculateWaitTime()
{
    int i = 0;
    for(i = 0;i < nTasks;i++)
    {
        if( ods[i].turnAroundTime > 0)
            ods[i].waitTime = ods[i].turnAroundTime - ods[i].T
    }
}
/*Calculate and set context switch for each task */
void calculateContextSwitches()
{
    int i = 0;
    for(i = 0;i < nTasks;i++)
    {
        if(ods[i].Nchild == 0)
        {
            // Process context switch
        }
    }
}
ods[i].contextSwitches = (ods[i].Tx/FIXED_TIME_SLICE) - 1;  
}
else if(ods[i].Nchild > 0)  
{ /*parent tasks:: Context switches = Number of children + context switches required for remaining Tx time after 
releasing all children*/
ods[i].contextSwitches = (ods[i].Nchild - 1) + ((ods[i].Tx - 
ods[i].maxTchildTime)/FIXED_TIME_SLICE);
}
/* Calulate and set turn around time for each task */
void calculateTurnAroundTime(int threadId)
{
    int i = 0;
    float waitTime = 0.0,timeSlice = 0.0;
    char taskId[10]={};
    if(threadId == 1)
    {
        strcpy(taskId,curTaskId_schd1);
        waitTime = waitTime_schd1;
        timeSlice = timeSlice_schd1;
    }
    else if(threadId == 2)
    {
        strcpy(taskId,curTaskId_schd2);
        waitTime = waitTime_schd2;
        timeSlice = timeSlice_schd2;
    }
    for(i = 0;i < nTasks;i++)
    {
        if(strcmp(ods[i].Tid,taskId) == 0)
        {
            ods[i].turnAroundTime = waitTime + timeSlice;
            waitTime = waitTime + timeSlice;
        }
    }
    if(threadId == 1)
    {
        waitedTime_schd1 = waitTime;
    }
    else if(threadId == 2)
    {
        waitedTime_schd2 = waitTime;
    }
}
/* Main function*/
int main()
{
    pthread_t scheduler_1, scheduler_2;
    int retValue=0;
    int timeSlice=0;
    void *exit_status;
    int threadId1=0,threadId2=0;
    /*initialise the mutex*/
    pthread_mutex_init(&lock,NULL);

memset(&tds, 0, sizeof(tds));
/* initialise the outputDataStructure*/
memset(&ods, 0, sizeof(ods));
/* Read the task inputs from the stdin*/
retValue = ReadTaskInputs();
if(retValue != SUCCESS)
{
    printf("\nError in reading the Inputs");
    return 1;
}
/*create scheduler1 and 2*/
threadId1 = 1;
retValue = pthread_create(&scheduler_1,NULL,TaskScheduler,&threadId1);
if(retValue != SUCCESS)
{
    printf("\nError in creating thread 1");
    return 1;
}
threadId2 = 2;
retValue = pthread_create(&scheduler_2,NULL,TaskScheduler,&threadId2);
if(retValue != SUCCESS)
{
    printf("\nError in creating thread 2");
    return 1;
}
/* wait till threads(scheduler1/2) are exited*/
retValue = pthread_join(scheduler_1,&exit_status);
if(retValue != SUCCESS)
{
    printf("\nError in joining thread");
    return 1;
}
retValue = pthread_join(scheduler_2,&exit_status);
if(retValue != SUCCESS)
{
    printf("\nError in joining thread");
    return 1;
}
calculateWaitTime();
calculateContextSwitches();
printOutputTDS();
/* destroy the mutex*/
pthread_mutex_destroy(&lock);
return 0;

/* Freeing the memory*/
void freeResources()
{
    int i = 0, j = 0;
    if(input.Tchild != NULL)
        free(input.Tchild);
    if(input.TchildStartTime != NULL)
        free(input.TchildStartTime);
    for(i=0;i<nTasks;i++)
    {
        for(j=0;j<256;j++)
        {
            if(tds[i].Tchild[j] != NULL)
                free(tds[i].Tchild[j]);
        }
    }
}
if(tds[i].TchildStartTime != NULL)
    free(tds[i].TchildStartTime);
}

void printTDS()
{
    int i=0,j=0,k=0;
    for (i=0;i<nTasks;i++)
    {
        printf("\nTid = %s  Tx = %0.2f Nparent = %lu Nchild = %lu priority = %0.2f taskStatus = %d",tds[i].Tid,tds[i].Tx,tds[i].Nparent,tds[i].Nchild,tds[i].priority,tds[i].taskStatus);
        printf("  Tchild = ");
        if(tds[i].Nchild == 0)
        {
            printf("N/A");
        }else
        {
            for(j=0;j<tds[i].Nchild;j++)
                printf("%s ",tds[i].Tchild[j]);
        }
    }
}
printf(" TchildStartTime = ");
    if(tds[i].Nchild == 0)
    {
        printf("0.0");
    }
    else
    {
        for(k=0;k<tds[i].Nchild;k++)
            printf("%0.2f  ",tds[i].TchildStartTime[k]);
    }

    printf("\n\n");
}

void printOutputTDS()
{
    int i = 0,j = 0,k = 0;
    printf("\n Output :");
    for (i = 0;i < nTasks;i++)
    {
        printf("\nTid = %s ExecutionTime = %0.2f  TurnAroundTime = %0.2f WaitTime = %0.2f contextSwitches = %d",ods[i].Tid,ods[i].Tx,ods[i].turnAroundTime,ods[i].waitTime,ods[i].contextSwitches);
    }
    printf("\n");
}

/*Validate the input*/
int validateInput(char* inputBuffer)
{
    int i = 0,retValue = SUCCESS,j = 0;
    int isChildAvail = 0,NchildCount = 0;
    float Tx = 0.0,totalTchildTime = 0.0;
    char *temp;
    char *Tchild;
    char* TchildStartTime;
    if(inputBuffer != NULL)
    {
        temp = strtok(inputBuffer,"t");
        while(temp != NULL)
        {
            if(i == 0)
            {
                if(strcmp(temp," ") == 0)
                {
                    retValue = 1;
                    return retValue;
                }
            }
            else if(i == 1)
            {
                if(strcmp(temp," ") == 0)
                {
                    retValue = 1;
                    return retValue;
                }
                else
                {
                    Tx = atof(temp);
                }
            }
            i++;
        }
    }
}
else if(i == 2)
{
    if(strcmp(temp," ") == 0)
    {
        retValue = 1;
        return retValue;
    }
}
else if(i == 3)
{
    if(strcmp(temp," ") == 0)
    {
        retValue = 1;
        return retValue;
    }else if(atoi(temp) > 0)
    {
        NchildCount = atoi(temp);
        isChildAvail = 1;
    }
}
else if(i == 4)
{
    if(isChildAvail == 1 && strcmp(temp," ") == 0)
    {
        retValue = 1;
        return retValue;
    }
    else if(isChildAvail == 1)
    {
        Tchild = strdup(temp);
        Tchild = strtok(Tchild," ,");
        while(Tchild != NULL)
        {
            Tchild = strtok(NULL," ,");
            j++;
        }
        if(NchildCount != j)
        {
            retValue = 1;
            return retValue;
        }
    }
}
else if(i == 5)
{
    if(isChildAvail == 1 && strcmp(temp," ") == 0)
    {
        retValue = 1;
        return retValue;
    }
    else if(isChildAvail == 1)
    {
        temp = strtok(temp," ,");
        while(temp != NULL)
        {
            temp = strtok(NULL," ,");
        }
    }
}
TchildStartTime = strdup(temp);
if(TchildStartTime != NULL)
    totalTchildTime = totalTchildTime + atof(TchildStartTime);
        
    temp = strtok(NULL,",");
        
    if(totalTchildTime > Tx)
    {
        retValue = 1;
        return retValue;
    }
        
    temp = strtok(NULL,"	");
i++;
    
return retValue;

/* Create the Task Data Structure*/
int createTaskDataStructure(int index)
{
    int retValue = 0;
    
    strcpy(tds[index].Tid,input.Tid);
tds[index].Tx = input.Tx;
tds[index].Nparent = input.Nparent;
tds[index].Nchild = input.Nchild;
    retValue = copyChildInfo(index);
tds[index].priority = -1;
tds[index].taskStatus = NotReady;
    
    return retValue;
}

/* Process and Copy the Tchild,TchildstartTime info into TDS table from input*/
int copyChildInfo(int index)
{
    int i=0;
    float childStartTime = 0.0;
    char* temp = NULL,*str=NULL;
    
    /* alloc memory for tchild[]*/
    for(i=0;i<256;i++)
    {
        tds[index].Tchild[i] = (char*)(malloc (sizeof(char)*10));
        memset(tds[index].Tchild[i],0,sizeof(char)*10);
    }
    
    /* separate the string input.Tchild*/
    temp = strdup(input.Tchild);
temp = strtok(temp,",");
i = 0;
    while(temp!=NULL)
    {
        strcpy(tds[index].Tchild[i],temp);
temp = strtok(NULL,",");
i++;
    }
    tds[index].TchildStartTime = (float*) malloc( 256 * sizeof(float));
memset(tds[index].taskTimeSlice,0,1024*sizeof(float));
memset(tds[index].TchildStartTime,0,256*sizeof(float));
    temp = strdup(input.TchildStartTime);
temp = strtok(temp,",");

}
i = 0;
while(temp != NULL)
{
    str = strdup(temp);
    if(str != NULL)
    {
        childStartTime = atof(str);
        tds[index].TchildStartTime[i] = atof(str);
        if(i > 0)
        {
            tds[index].taskTimeSlice[i] = childStartTime - tds[index].taskTimeSlice[i-1];
        }
        else if(i == 0)
        {
            tds[index].taskTimeSlice[i] = childStartTime;
        }
        temp = strtok(NULL, ",");
    i++;
}
return 0;
}

/* Get the max task exec time(Tx max) from teh tds..Needed for calculating the priority*/
float maxExecutionTime()
{
    int i=0,j=0;
    float maxTx=0;
    for(i = 0; i < nTasks; i++)
    {
        if(tds[i].Tx > maxTx)
        {
            maxTx = tds[i].Tx;
        }
    }
    return maxTx;
}

/*Setting the task status to READY for all the tasks in table which has NO parents/incompleted tasks*/
void setTaskStatusForTable()
{
    int i;
    for(i = 0; i < nTasks; i++)
    {
        if(tds[i].Nparent == 0 && tds[i].taskStatus != Completed && tds[i].taskStatus != Running)
        {
            //printf("nSetting the task status to READY!!!!!!!!!!!!!!!");
            tds[i].taskStatus = Ready;
        }
    }
}

/* Set the individual task status */
void setTaskStatus(char* taskId,int status)
{
    int i;
    for(i = 0; i < nTasks; i++)
    {
        if(strcmp(taskId,tds[i].Tid)== 0)
        {
            tds[i].taskStatus = status;
            if(status == Completed)
Compute the Priority */
void calculatePriority()
{
    int i,j;
    float temp,maxTx=0;
    maxTx = maxExecutionTime();
    for(i = 0; i < nTasks; i++)
    {
        temp = maxTx - tds[i].Tx;
        if(maxTx != 0)
            tds[i].priority = tds[i].Nchild + (temp/maxTx);
        if(tds[i].taskStatus == Completed)
            tds[i].priority = -1;
    }
}

/* Create the output Data Structure*/
int createOutputDataStructure(int index)
{
    strcpy(ods[index].Tid,input.Tid);
    ods[index].Tx = input.Tx;
    ods[index].Nchild = input.Nchild;
    if(input.Nchild > 0)
    {
        ods[index].maxTchildTime = tds[index].TchildStartTime[input.Nchild - 1];
        printf("\nmaxTchildTime = %f",ods[index].maxTchildTime);
    }
    return 0;
}