PROJECT REPORT ON

Process Scheduler Using Min Heaps and Red Black Trees

Submitted By:

Ankit Dave
UFID: 9211-5520

For:
Course: Advanced Data Structures (COP5536)

28th March, 2018

Department of Computer and Information Science Engineering
University of Florida, Gainesville, FL
32608

Objective

The purpose of the project was rot simulate a process scheduler, similar to the ones in operating systems. In this project, the process was simulated using a text file, which gave the input on when a process came in, how long it intended to use the CPU time. The program, of-course was designed to not know about the incoming process until the time stamp of the incoming process matched the simulated time stamp of the program. The actual process scheduler was a Min Heap data structure. This needed to be designed was speed and accuracy. The data stored in the heap needs to be simple and the heap must give out the process that has taken up the least CPU time (or executed time) until then. The background data of the process is stored in a RED BLACK TREE.

Design

The design of the project can be broken down into 4 parts:

1. The Scheduler

This is the class in the project, that schedules the next process. It is designed to simulate a min heap data structure. In this the data stored is the executed time and a pointer to the node in the other data structure that has the information of the process itself. The reason for a small data structure, is so that the speed is maximised. The scheduler cannot take a lot of time, otherwise what time will the actual process get.

This heap is also designed to have a self doubling capacity as need comes. If the amount of process that need to be scheduled turns to be too large, the heap dynamically doubles in size to accommodate all data. This means the heap, if of size 2 will go to 4, followed by 8, then 16……etc.

The scheduler is the simplest and the ultimate authority in deciding which process goes next.

2. Task List

This particular part is built on a RED BLACK TREE. This data structure is very powerful and simulates a quick environment to get large chunks of data, and data in some query. All the data of the process, like ID, time, total time etc. get stored in this. This supports many operations as may be required by the scheduler or the interrupt handler. Searching is especially quick in such a structure and hence most suitable for this as well. When a process is initlised, it is fist put into the data structure. This has all the information, about the process, and all the information on what and where the information needed by the process might be. The scheduler hence, has pointers to nodes in this data structures. this allows process to have quick access to the details of it self. On the other hand, an external/sudo user trying to get data from this will use the function provided in this class to access it. This prevents the data from being tampered by any user. Only the Job List class has capacity to change data within itself. Any external attempt is attracted by using functions like search, which return a copy of the data queried, and hence prevent changes that might lead to data loss.

3. Interrupt Handler

While implementing a Round Robin Algortihm. It is paramount to make sure that the interrupts are called and handled in an appropriate and non risky manner. The virtual OS itself interrupts any process in the CPU, if that process has executed for 5ms. It the calls the scheduler to give it the next process to put in the CPU (which is the process that has run for the least about of time up until then). This also handles cases when a CPU is given up by the process since it no longer needs more time.

The scheduling algorithm uses a flag “isocuupied” to make sure that no other process or function can cause any problem to the process that is in the CPU. If the flag is set to 1, only and only the interrupt handler has access to the CPU to add and remove processes.

4. Module integration

This is helper and substantial background codes, that ask and give data and reform process needed for quick retrieval of data and scheduling of the process. These background models are the hard code that make the entire system run seamlessly.

Data Structures and function capabilities

MinHeap:

Constructor to iptilize all the variables at time=0:
```
MinHeap();
```

Insert new process that needs to be scheduled:
```
insert(mhnode &data)
```

Remove a process that no linger needs CPU time:
```
Delete(mhnode &data);
```

Helper function to keep the data structure in a way that the minimum value can be return in O(n) time:

```
Heapify(int root);
```

Increase the size of the Heap:
```
resize();
```

Function for user to get the private data:
```
getcursize();
gettotalelements();
puttotalelements(int val);
```

Print the current state of the Heap:
```
print();
```

Helper function to perform large operations:

```
    int parent(int i)
    int left(int i)
    int right(int i)
    int min(int a, int b)
    swap(mhnode &x, int &y);

           mhnode extractmin();
            int setglobaltime(long gt);
```

Data that is not accessible (abstracted to prevent mishandling):-

The node with pointer to the data:

```
    mhnode *arr;
```

size of the heap at any point:

```
    int total_size;
```

count of the process that need to be scheduled:

```
    int current_count;
```

current time stamp:

```
    long global_time;
```

2. RED BLACK TREE

Pointer to the data in memory

```
    rbtnode *root;
```

Time

```
    static long global_time;
```

Constructor:

```
    RBT();
```

Add new process and it’s data:

```
    bool insert(rbtnode* i_job);
```

Find a process with particular parameters:

```
    rbtnode* search(rbtnode* s_job);
```

Remove a data structure:

```
    bool remove(rbtnode* r_job);
```

Helper function:

```
rbtnode* maketree(rbtnode* i_job);
static int putgloabltime(long gt);
bool RL(rbtnode* i_job);
bool RR(rbtnode* i_job);
void PrintJobs(int val, int flag, ofstream &result);
int TravRootnxt(rbtnode *tmp, int val, int flag, ofstream &result, int a);
int TravRootprev(rbtnode *tmp, int val, int flag, ofstream &result, int a);
void PrintJobRange(int start, int end,ofstream &result);
int PrintJobRangecall(int start, int end, rbtnode* tmproot, ofstream &result, int a);
```

3. Nodes

```
rbtnode{
    jobID;
    colour;
    total_time;
    executed_time;
    rem_time;
        };


struct mhnode{
      data value:executed_time;
     pointer to main data
     }
```

FINAL FUNCTIONING

The design was very successful. The scheduler works perfectly. If a series of instances are provided and commands given for the processor to do its work of scheduling, the actual time taken by the scheduler is less that 1/10000 of the total time it gives out to the process. The complex yet simple data structures allow for quick access and the redundancies in the code allow for huge data and process to be put into the scheduler and get results seamlessly.