**Project 1: GTThreads Project – Interim Report Submission**

**Name:** Raghavendra V. Belapure

**Part I – Understanding Implementation of GTThreads, CFS and O(1) Priority Scheduling Algorithms**

The critical data-structures that are used in the package are shown in below. The following figure shows these data structures, their components and functions that they support. The figure also shows the dependency between these structures.

GTThreads library uses a generic scheduler interface. The specific schedulers implement this interface for the necessary behavior. The scheduling process using signals using this generic scheduler is shown on next page using a flowchart. Working of 3 critical functions of both schedulers is shown in table below.

|  |  |  |
| --- | --- | --- |
| **Function name** | **Completely Fair Scheduler** | **O(1) Priority Scheduler** |
| preempt\_current\_uthread | * Delete process node from red-black tree * Update vruntime * Insert process node again | * Remove the process from the active run-queue * Add it to expired run-queue * When a uthread is finished, add it to a zombie queue |
| pick\_next\_uthread | * Get node from red-black tree with lowest vruntime | * Find highest priority runnable uthread * If found, remove from priority queue * If active queue empty, swap active & expired queues * If both are empty, no more jobs to do |
| Resume\_uthread | * Set interval timer which generates SIGVTALRM which is SIGSCHED * swapcontext() to newly selected process | |



**Part II – Design for a process wide scheduling algorithm**

The proposed design is as follows:

1. A data structure is created which stores a group of uthreads. Such data structure is called a process and a unique process id will be assigned to it.
2. There are different sizes of the matrices. For each type of matrix, we create two processes. One of the processes contains “optimal” number of uthreads (equal to the number of cores in the CPU) while the other process is the “greedy” process (it has more number of uthreads than the number of cores).
3. Each uthread will be a member of one unique group which will be the process that contains that uthread.
4. Every uthread will be assigned to a kthread using a round robin algorithm, without regard to its group membership. This is how the uthreads get mapped to the kthreads in current implementation of GTThreads.
5. Process will be a structure similar to the struct uthread and the process attributes will contain the execution time and information about the timeslice.
6. During the scheduling, the process node will be inserted into the red-black tree with the vruntime for the process as the key for red-black tree.
7. The scheduler will select a process with lowest vruntime and will be chosen for running. The threads inside the process will be chosen in a round-robin and will be run using the corresponding kthread to which those are mapped.
8. This way, the CPU power will be divided fairly among the competing processes without regard to the number of thread they have created. The decision of scheduling the process will be solely based upon the vruntime of the process and not the vruntime of the individual threads.