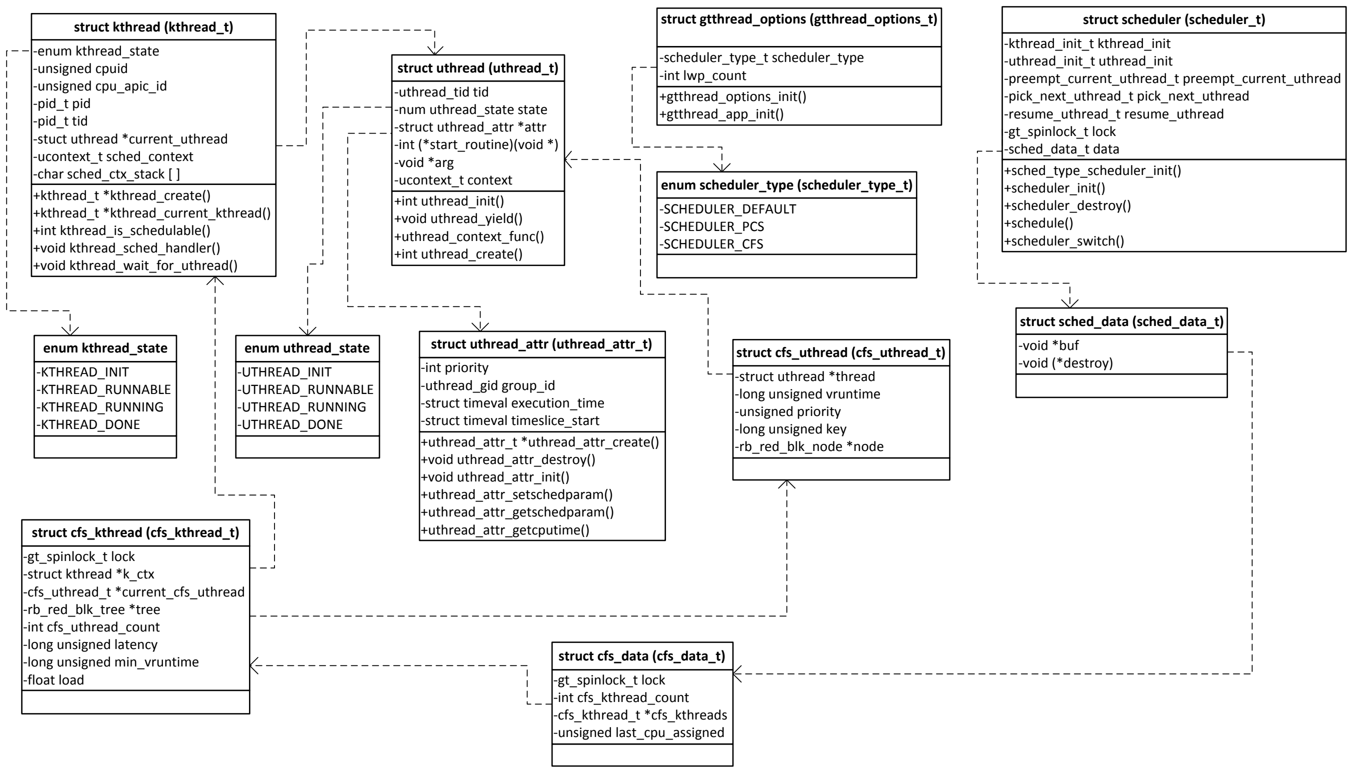
Project 1: GTThreads Project – Interim Report Submission

Name: Raghavendra V. Belapure

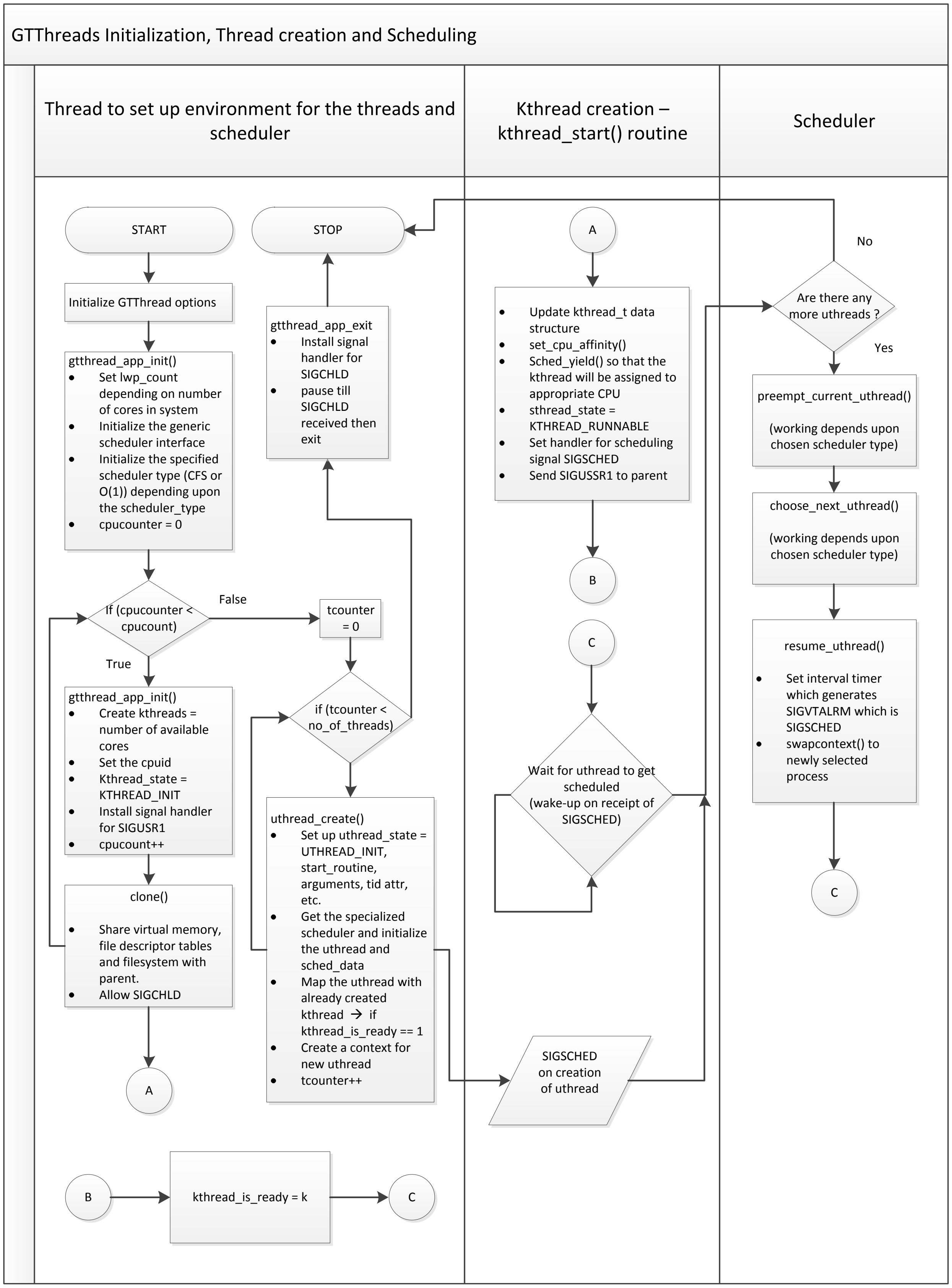
<u>Part I</u> – Understanding Implementation of GTThreads, CFS and O(1) Priority Scheduling Algorithms

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.


struct kthread (kthread_t)
-enum kthread_state
-unsigned cpuid
-unsigned cpu_apic_id
-pid_t pid
-pid_t tid
-stuct uthread *current_uthread
-ucontext_t sched_context
-char sched_ctx_stack [ ]
+kthread_t *kthread_create()
+kthread_t *kthread_current_kthread()
+int kthread_is_schedulable()
+void kthread_sched_handler()
+void kthread_wait_for_uthread()
struct uthread (uthread_t)
-uthread_tid tid
-num uthread_state state
-struct uthread_attr *attr
-int (*start_routine)(void *)
-void *arg
-ucontext_t context
+int uthread_init()
+void uthread_yield()
+uthread_context_func()
+int uthread_create()
struct gtthread_options (gtthread_options_t)
-scheduler_type_t scheduler_type
-int lwp_count
+gtthread_options_init()
+gtthread_app_init()
struct scheduler (scheduler_t)
-kthread_init_t kthread_init
-uthread_init_t uthread_init
-preempt_current_uthread_t preempt_current_uthread
-pick_next_uthread_t pick_next_uthread
-resume_uthread_t resume_uthread
-gt_spinlock_t lock
-sched_data_t data
+sched_type_scheduler_init()
+scheduler_init()
+scheduler_destroy()
+schedule()
+scheduler_switch()
enum scheduler_type (scheduler_type_t)
-SCHEDULER_DEFAULT
-SCHEDULER_PCS
-SCHEDULER_CFS
struct sched_data (sched_data_t)
-void *buf
-void (*destroy)
enum kthread_state
-KTHREAD_INIT
-KTHREAD_RUNNABLE
-KTHREAD_RUNNING
-KTHREAD_DONE
enum uthread_state
-UTHREAD_INIT
-UTHREAD_RUNNABLE
-UTHREAD_RUNNING
-UTHREAD_DONE
struct uthread_attr (uthread_attr_t)
-int priority
-uthread_gid group_id
-struct timeval execution_time
-struct timeval timeslice_start
+uthread_attr_t *uthread_attr_create()
+void uthread_attr_destroy()
+void uthread_attr_init()
+uthread_attr_setschedparam()
+uthread_attr_getschedparam()
+uthread_attr_getcputime()
struct cfs_uthread (cfs_uthread_t)
-struct uthread *thread
-long unsigned vruntime
-unsigned priority
-long unsigned key
-rb_red_blk_node *node
struct cfs_kthread (cfs_kthread_t)
-gt_spinlock_t lock
-struct kthread *k_ctx
-cfs_uthread_t *current_cfs_uthread
-rb_red_blk_tree *tree
-int cfs_uthread_count
-long unsigned latency
-long unsigned min_vruntime
-float load
struct cfs_data (cfs_data_t)
-gt_spinlock_t lock
-int cfs_kthread_count
-cfs_kthread_t *cfs_kthreads
-unsigned last_cpu_assigned

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	


GTThreads Initialization, Thread creation and Scheduling
Thread to set up environment for the threads and scheduler
Kthread creation – kthread_start() routine
Scheduler
START
Initialize GTThread options
gtthread_app_init() • Set lwp_count depending on number of cores in system • Initialize the generic scheduler interface • Initialize the specified scheduler type (CFS or O(1)) depending upon the scheduler_type • cpucounter = 0
If (cpucounter < cpucount)
False
True
gtthread_app_init() • Create kthreads = number of available cores • Set the cpuid • Kthread_state = KTHREAD_INIT • Install signal handler for SIGUSR1 • cpucount++
clone() • Share virtual memory, file descriptor tables and filesystem with parent. • Allow SIGCHLD
A
STOP
gtthread_app_exit • Install signal handler for SIGCHLD • pause till SIGCHLD received then exit
tcounter = 0
if (tcounter < no_of_threads)
uthread_create() • Set up uthread_state = UTHREAD_INIT, start_routine, arguments, tid attr, etc. • Get the specialized scheduler and initialize the uthread and sched_data • Map the uthread with already created kthread → if kthread_is_ready == 1 • Create a context for new uthread • tcounter++
B
kthread_is_ready = k
C
A
• Update kthread_t data structure • set_cpu_affinity() • Sched_yield() so that the kthread will be assigned to appropriate CPU • sthread_state = KTHREAD_RUNNABLE • Set handler for scheduling signal SIGSCHED • Send SIGUSSR1 to parent
B
C
Wait for uthread to get scheduled (wake-up on receipt of SIGSCHED)
SIGSCHED on creation of uthread
No
Are there any more uthreads ?
Yes
preempt_current_uthread() (working depends upon chosen scheduler type)
choose_next_uthread() (working depends upon chosen scheduler type)
resume_uthread() • Set interval timer which generates SIGVTALRM which is SIGSCHED • swapcontext() to newly selected process
C

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.