Enabling Precedence Constrained Task Scheduling in LITMUS^{RT}

By
Ashish Prasad
Digvijay Singh
Shashank Sharma

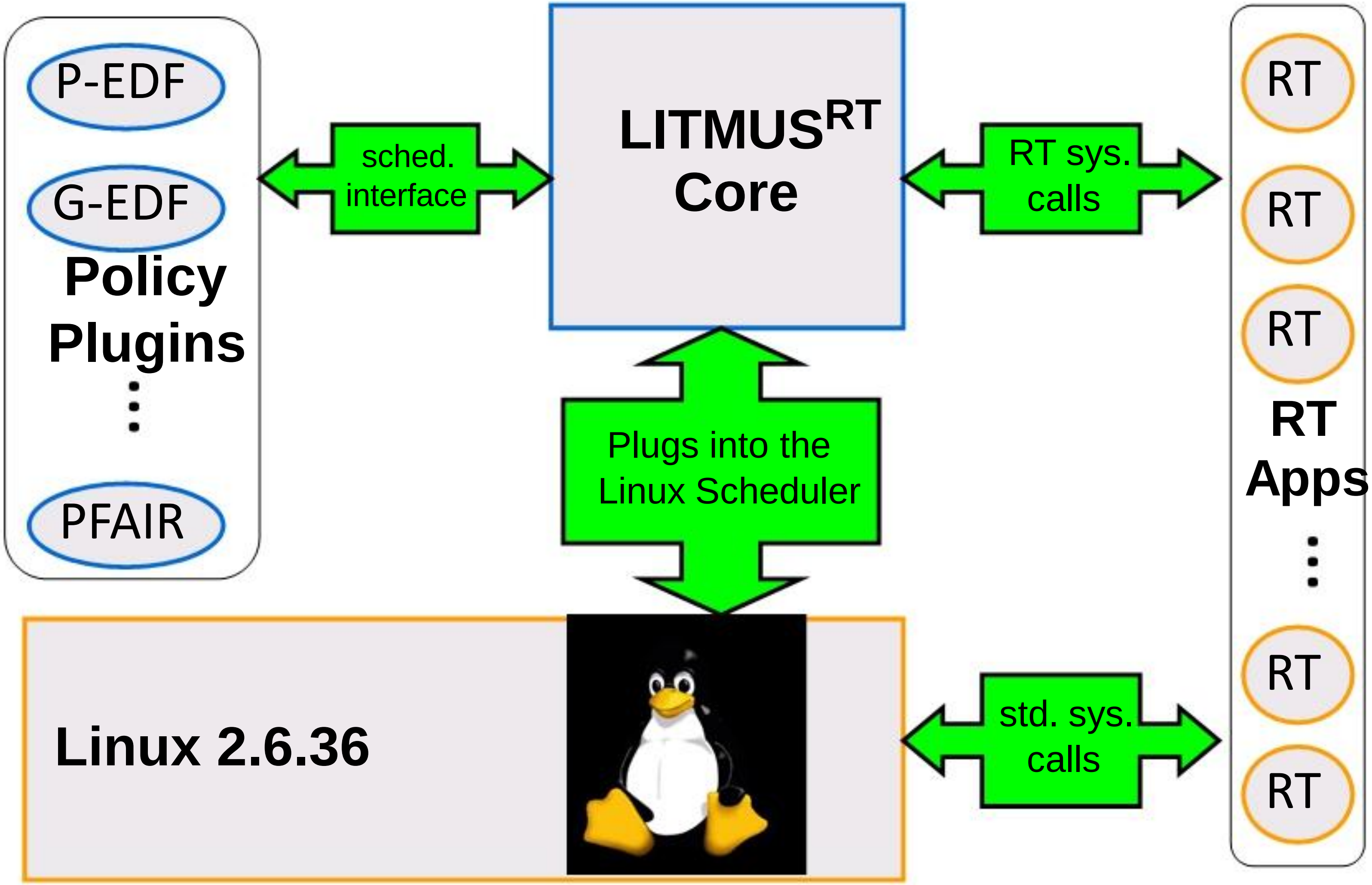
Index

- LITMUS^{RT} : Design and Architecture
- Task model
- Data Structure
- GSN-EDF
- User Interface
- System Call
- Kernel Interface
- Result
- Challenges

LITMUS^{RT}

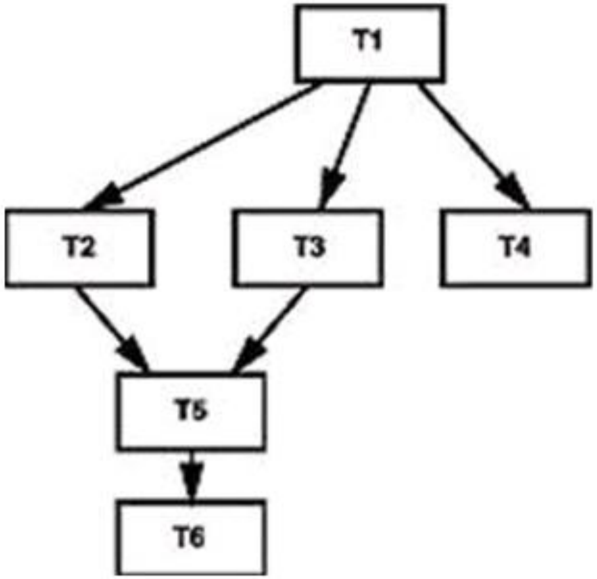
- The LITMUS^{RT} patch is **a (soft) real-time extension of the Linux kernel** with a focus on multiprocessor real-time scheduling and synchronization. The Linux kernel is modified to support the sporadic task model and modular scheduler plugins. Clustered, partitioned, and global scheduling are included, and semi-partitioned scheduling is supported as well.
- The primary purpose of the LITMUS^{RT} project is to **provide a useful experimental platform for applied real-time systems research**.
- The current version of LITMUS^{RT} is **2012.1** and is based on Linux 3.0

LITMUS^{RT} Design



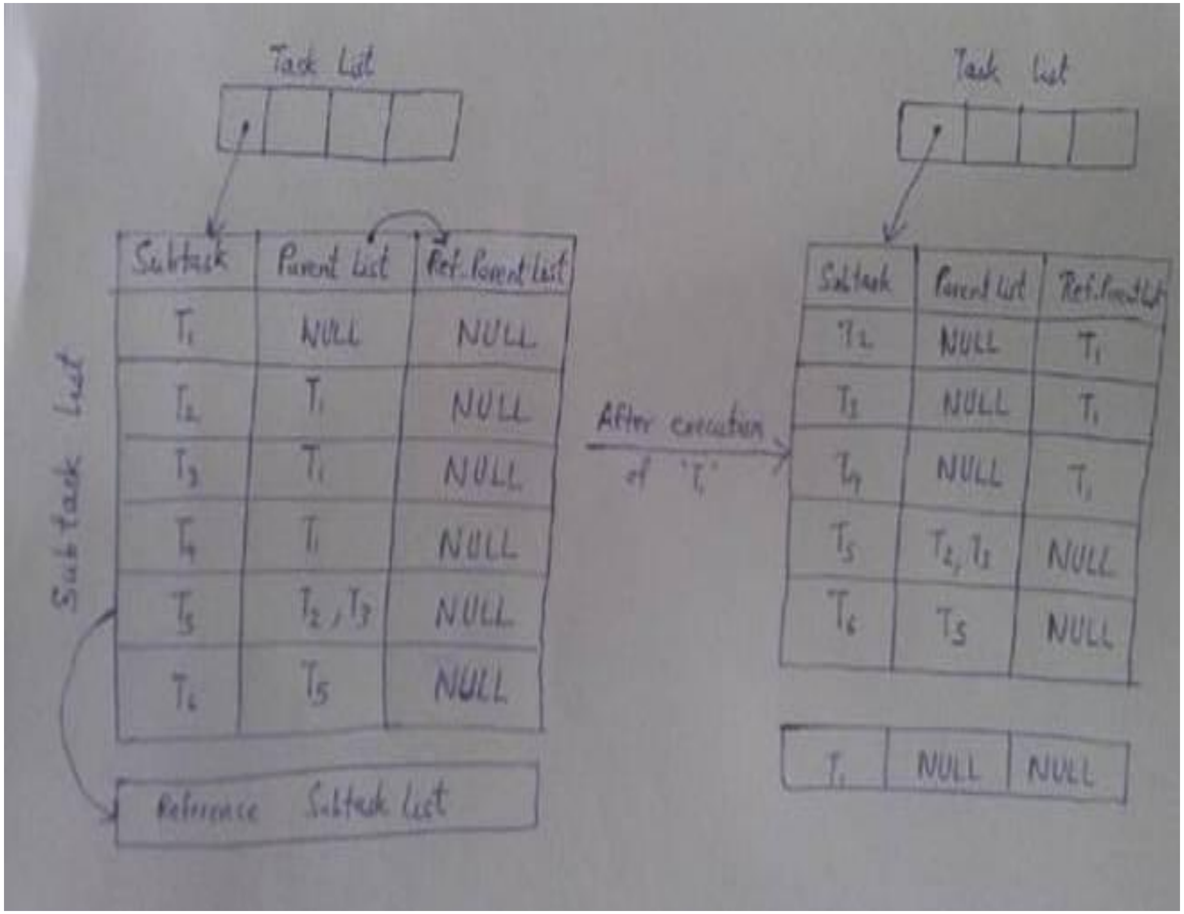
Dependent Constrained Tasks

- Directed Acyclic Graph
- Source Node
- Sink Node
- Dependency



Data Structure

- Task List
- Task
- Subtask

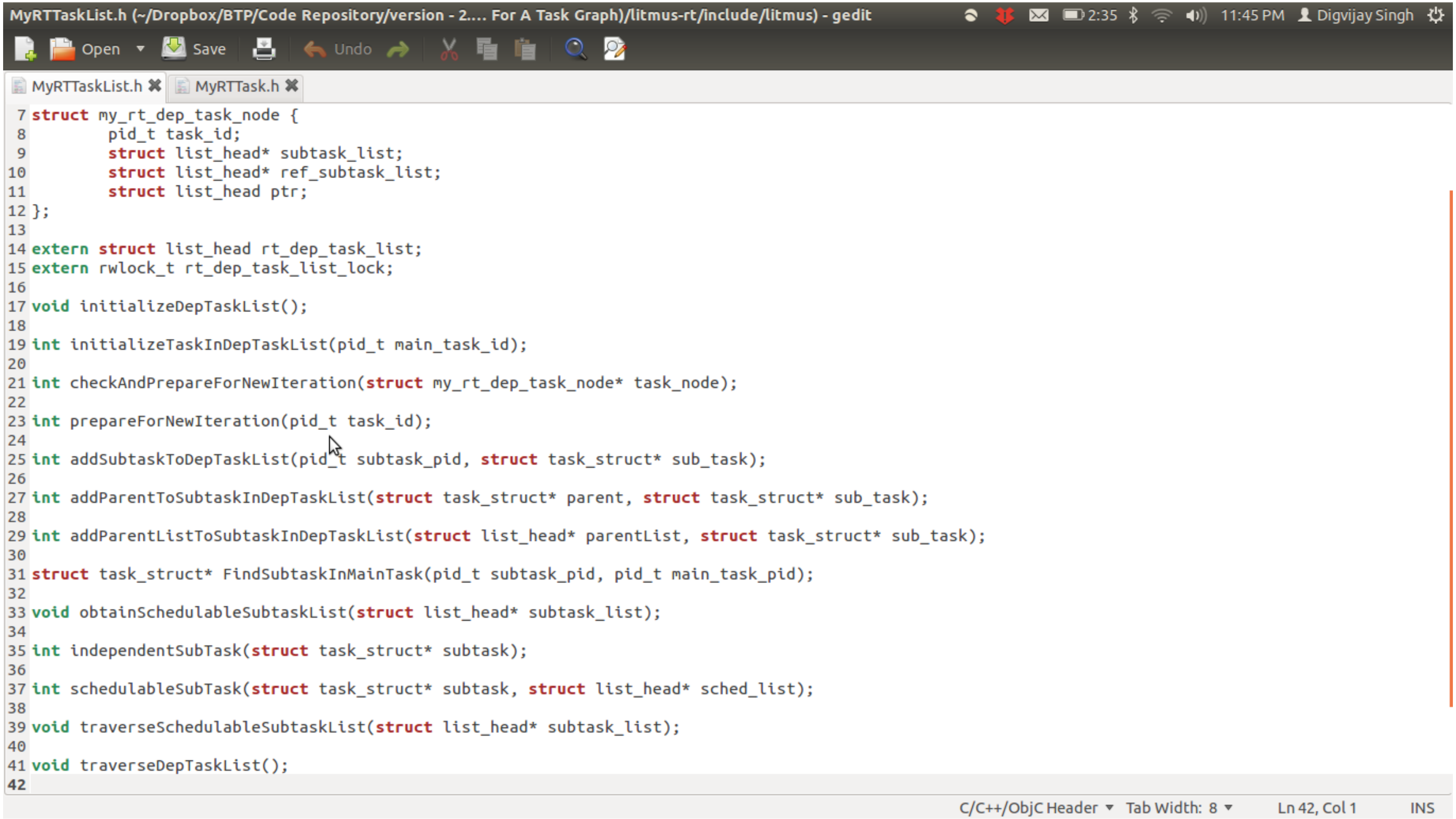


Subtask	Parent List	Ref. Parent List
T_1	NULL	NULL
T_2	T_1	NULL
T_3	T_1	NULL
T_4	T_1	NULL
T_5	T_2 , T_3	NULL
T_6	T_5	NULL

Subtask	Parent List	Ref. Parent List
T_2	NULL	T_1
T_3	NULL	T_1
T_4	NULL	T_1
T_5	T_2, T_3	NULL
T_6	T_5	NULL

T_1	NULL	NULL

Code Snippet

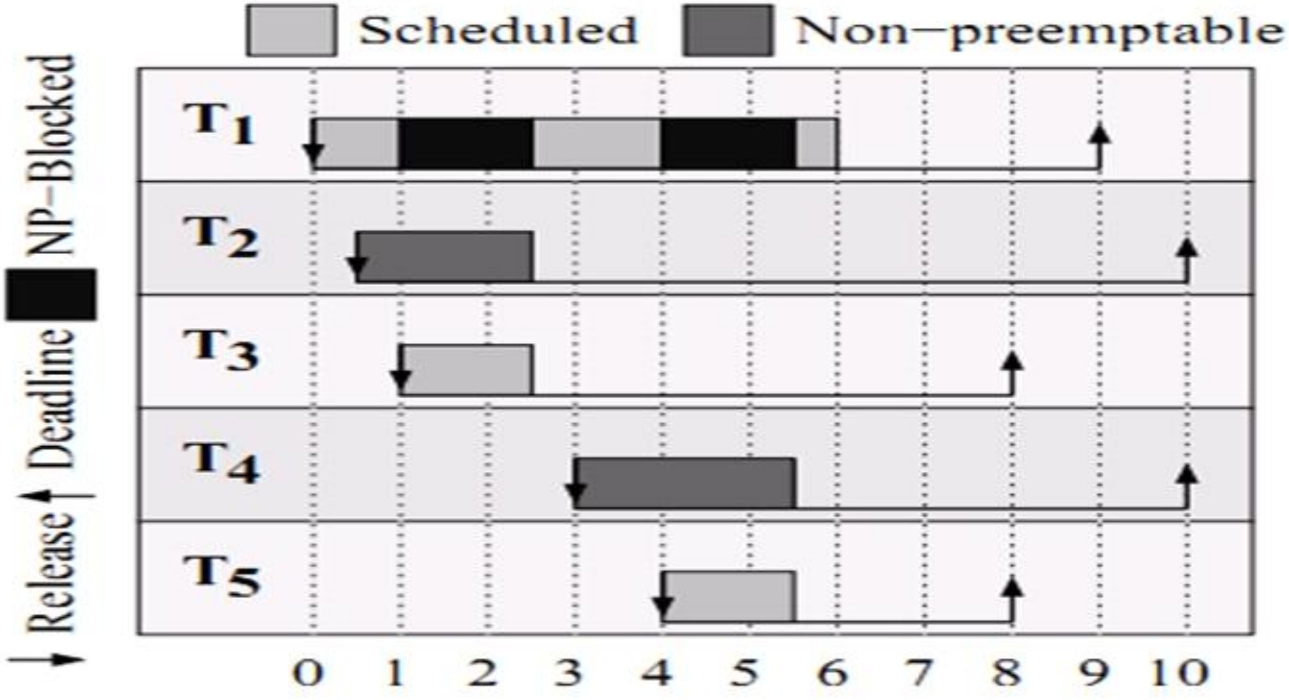


MyRTTaskList.h (~/Dropbox/BTP/Code Repository/version - 2.... For A Task Graph)/litmus-rt/include/litmus) - gedit

```c
struct my_rt_dep_task_node {
        pid_t task_id;
        struct list_head* subtask_list;
        struct list_head* ref_subtask_list;
        struct list_head ptr;
};

extern struct list_head rt_dep_task_list;
extern rwlock_t rt_dep_task_list_lock;

void initializeDepTaskList();

int initializeTaskInDepTaskList(pid_t main_task_id);

int checkAndPrepareForNewIteration(struct my_rt_dep_task_node* task_node);

int prepareForNewIteration(pid_t task_id);

int addSubtaskToDepTaskList(pid_t subtask_pid, struct task_struct* sub_task);

int addParentToSubtaskInDepTaskList(struct task_struct* parent, struct task_struct* sub_task);

int addParentListToSubtaskInDepTaskList(struct list_head* parentList, struct task_struct* sub_task);

struct task_struct* FindSubtaskInMainTask(pid_t subtask_pid, pid_t main_task_pid);

void obtainSchedulableSubtaskList(struct list_head* subtask_list);

int independentSubTask(struct task_struct* subtask);

int schedulableSubTask(struct task_struct* subtask, struct list_head* sched_list);

void traverseSchedulableSubtaskList(struct list_head* subtask_list);

void traverseDepTaskList();

```

C/C++/ObjC Header ▾ Tab Width: 8 ▾ Ln 42, Col 1 INS

Why-GSN-EDF

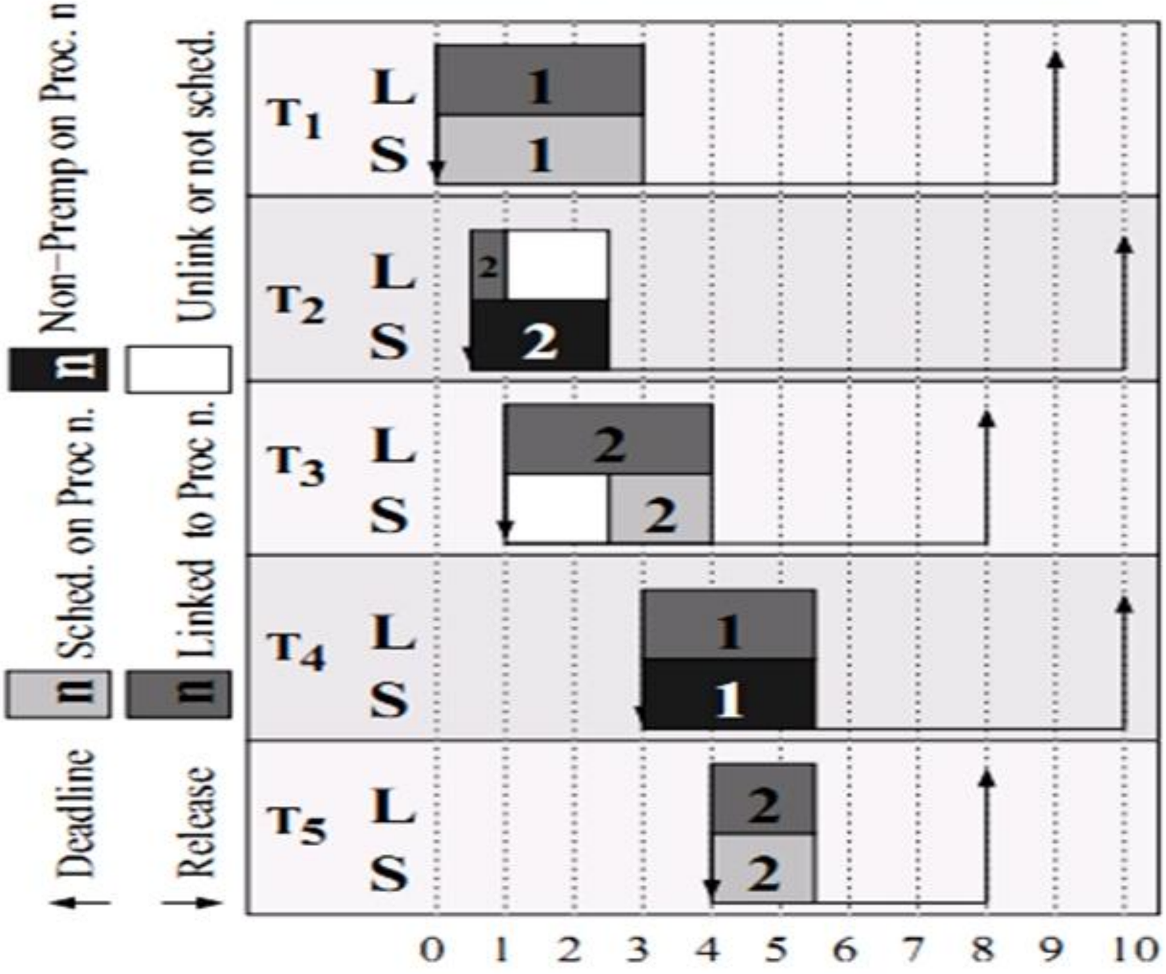


- T2 and T4 are non-preemptive
- Priority:
    - T1>T2, T1>T4
    - T1<T3, T1< T5

GSN-EDF Algorithm

- Jobs can become non preemptable for short durations of time.
- A job T_{ij} is only blocked by another non-preemptable job when T_{ij} is either released or resumed, and that such blocking durations are reasonably constrained.
- A job T_{ij} is non-preemptively blocked at time t iff T_{ij} is one of the highest-priority runnable jobs and it is not scheduled at t because a lower-priority non-preemptable job is scheduled instead.
- Each processor has a scheduled and linked job.
- Handle cases of Job arrival, Job completion and conversion of a non-preemptable task to a preemptable one.

GSN-EDF Advantage:

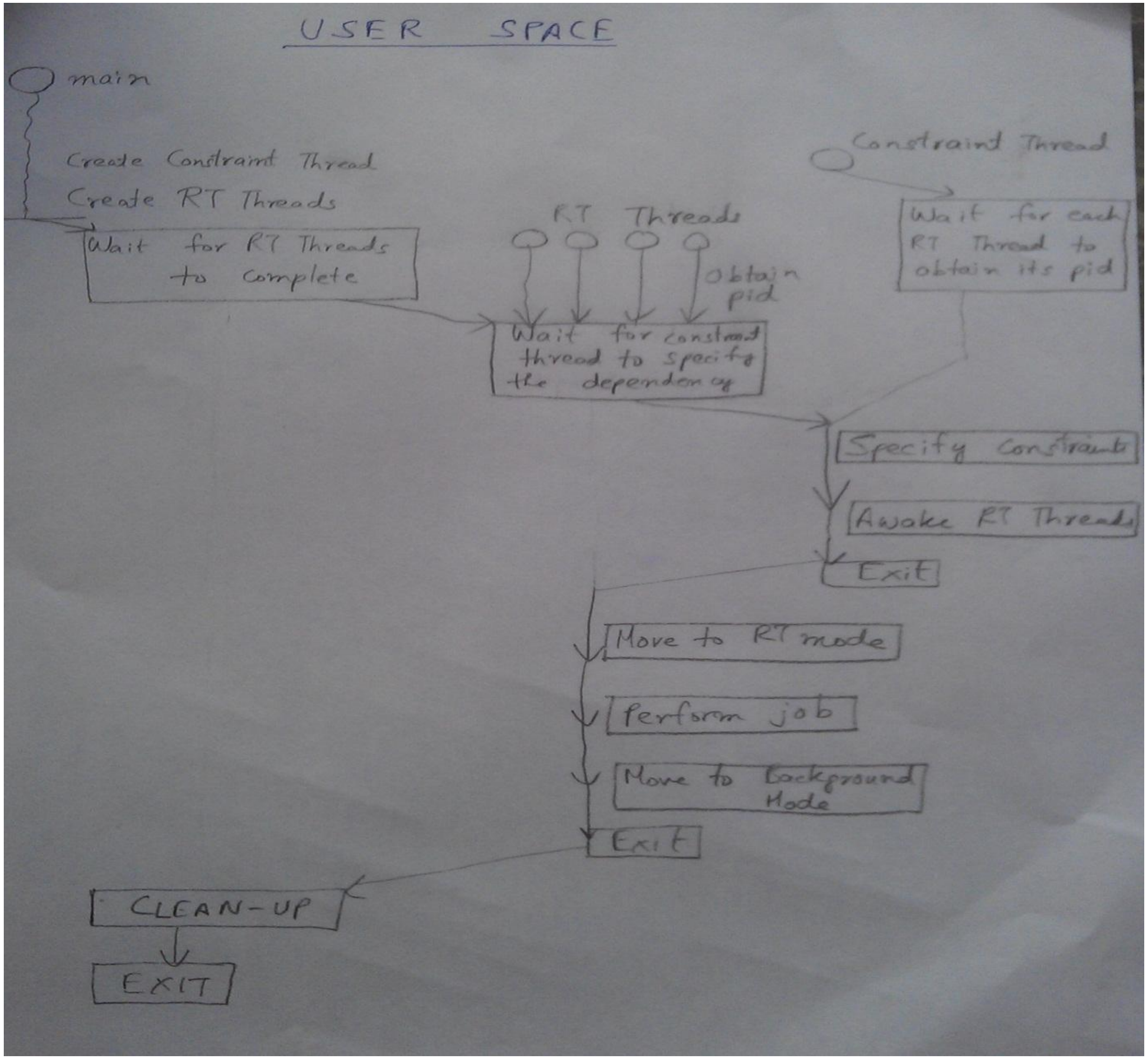


OUR CHANGES

The changes done are in the following three sections

- User Space
- System calls
- Kernel Space

User Interface



USER SPACE-I

- **Creation of threads-**In the user space, we create multiple threads. Each thread via the inbuilt system call (getid()), obtains its PID and this PID is used by our syscall to initialize a subtask in the data structure
- **Dummy Thread-**Apart from the RT threads, we created a dummy thread to specify the constraints.

USER SPACE-II

- **Lock**-Dummy thread waits on a lock until all the threads obtain their pids. The threads wait till all the constraints are specified and then resume to accomplish their tasks in real time mode.
- **Deletion of Threads -**Once the task gets completed, the corresponding subtasks are removed from the data structure at kernel side via our defined syscall.

SYSCALLS

- **Syscall-1 long sys_set_main_task_pid(pid_t subtask_pid, pid_t main_task_pid**
- **Syscall -2: long sys_init_dep_subtask (pid_t subtask_pid)**
- **Syscall-3:long sys_add_parent_to_subtask(pid_t parent_pid, pid_t subtask_pid)**
- **Syscall -4: long sys_exit_dep_task (pid_t main_task_pid)**

KERNEL SPACE

- Only a released task whose parent list is empty is eligible for scheduling.
- From the data structure we implemented, we maintain a list of schedulable tasks (whose parent list is empty).
- If an arrived job is released and has all preceding subtasks executed, it passed to the ready queue for scheduling. Else, we keep it in a waiting list.
- When a task is executed, we refresh the schedulable tasks list and add the waiting schedulable tasks to ready queue.

CHALLENGES

- Kernel compilation : Unstable kernel – 2.6.36
- Understand the control flow. (No documentation available for Litmus code.)
- Making changes in the code without breaking the existing flow.
- Multithreaded programming: shared resourced, race conditions, thread blocking, condition variables and locks.

Demonstration

Thank you.