

# Implementation designs for fine-grained scheduling in kernel space

Sreeram Sadasivam  
M.Sc Distributed Software Systems  
TU Darmstadt, Germany  
sreeram.sadasivam@stud.tu-darmstadt.de

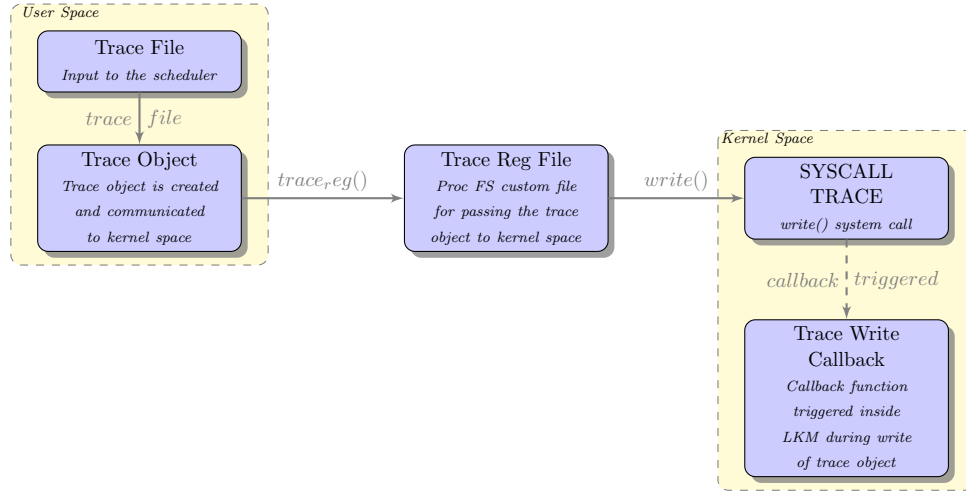
## 1 Design with no checking in user space

In the following designs, we address the use of check permission of memory access method entirely in Kernel space.

### 1.1 Design with no additional scheduler thread

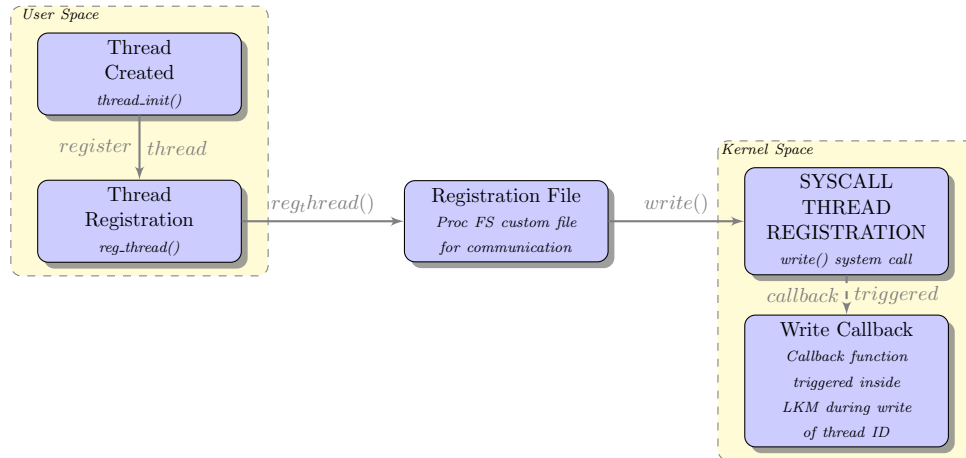
The design described in this section addresses the use of no additional scheduler thread.

#### Trace Registration



The trace file is passed on as an input for the scheduler. In the above flow diagram, the trace file is read by the main user thread at the start of its execution. It parses the file, creates and passes the trace object to the kernel space as string via a custom file created in the proc file system.

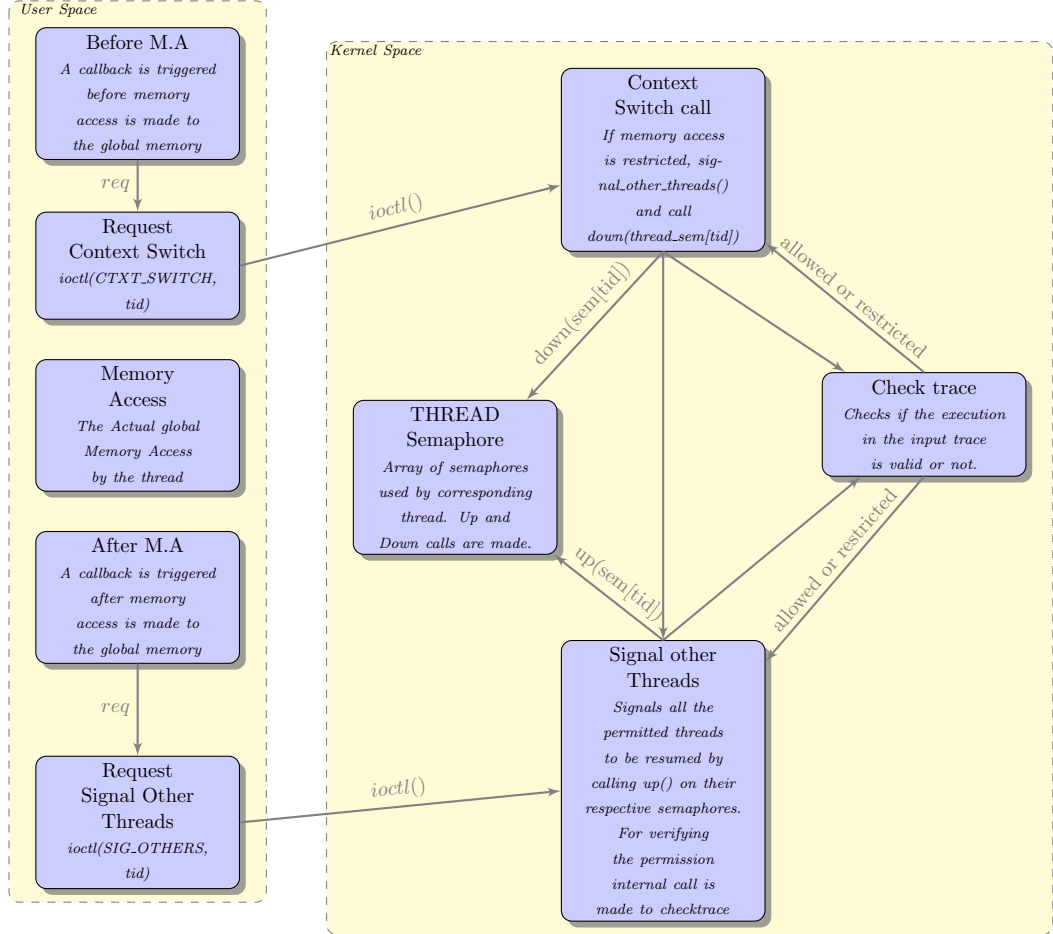
## Thread Registration



In the above picture, the registration block happens when a user thread is created. The registration happens via a custom proc file system.

## Memory Assessment

Prior to any global memory access, the given design would invoke `IOCTL` command with `CTXT_SWITCH` and thread id of the thread which addressed the memory event as its parameters.



## Pseudo Implementation

Data Types Section used by user space and kernel space

```
enum IOCTL_CMDS {
    GET_CURR_CLK = 1,
    CTXT_SWITCH = 2,
    SIGNAL_OTHER_THREADS = 3,
    RESET_CLK = 4,
    SET_MY_CLK = 5
}

enum mem_access{
    e_ma_restricted = 0,
    e_ma_allowed    = 1
}

struct vec_clk {
    int clocks[THREAD_COUNT],
}

struct trace_node {
    thread_id_t tid;
    vec_clk clk;
    int valid;
}

struct trace {
    trace_nodes trace_obj_arr[TRACE_LIMIT];
}
```

Check Permission for memory access

```
mem_access check_mem_acc_perm(vec_clk* curr_vec_clk, vec_clk*
    trace_inst, thread_id_t tid) {

    int i;
    if(trace_inst->clocks[tid-1] == curr_vec_clk->clocks[tid-1])
    {
        for i in range(0, THREAD_COUNT)
        {
            if(i!=(tid-1))
            {
                if(trace_inst->clocks[i] <= curr_vec_clk->clocks[i])
                {
                    continue;
                }
                else
                {
                    return e_ma_restricted;
                }
            }
        }
    }
}
```

```

        }
    }
}
else if(trace_inst->clocks[tid-1] < curr_vec_clk->clocks[tid-1])
{
    return e_ma_restricted;
}
return e_ma_allowed;
}

```

### User Space Implementation

```

BeforeMA() {
    ioctl(CTXT_SWITCH, thread_id);
}

AfterMA() {
    ioctl(SIGNAL_OTHER_THREADS, thread_id);
}

reset_clock() {
    ioctl(RESET_CLK);
}

//This method is defined by the thread library which is used by
//the user
thread_create_impl(thread t) {
    t ->thread_init(tid);
    t ->thread_exec(thread_function);
}

thread_function() {
    reg_thread(); //This method increments a threadcount
                //variable in kernel space.
    ....
    Before_MA(); //function triggered before accessing
                //the global memory
    Mem_Access(); //global memory access permitted for
                //the thread
    AfterMA(); //function triggered after
                //accessing the global memory
    ....
    thread_exit()
}

trace_reg() {

```

```

        fd = open("/proc/trace_reg", O_RDWR);
        close(fd);
    }

    main() {
        trace_reg()
        thread t = thread_create(tid, thread_function);
        //thread_create_impl() is called internally
        .....
        t.join();
        return EXIT_SUCCESS;
    }

```

### Kernel Space - General module definitions

```

semaphore threads_sems[THREAD_COUNT];
int wait_queue[THREAD_COUNT];
trace trace_obj;
vec_clk curr_clk;
int thread_count = 0;

module_init() {
    for i in range(0, THREAD_COUNT) {
        init(threads_sems[i] = 0;
            wait_queue[i] = 0;
            curr_clk[i] = 0;
        }
        alloc_ioctl_device(); //method used to allocate ioctl
                               device.
    }

    trace_reg_callback() {
        //The method parses the trace which is passed as string
        and stores in trace_obj
    }

    reg_thread_callback() {
        thread_count++;
    }
}

```

## Kernel Space - IOCTL

```
/* This method is triggered whenever ioctl commands are issued
   from the user space */
ioctl_access(IOCTL_CMDS cmd) {
    switch(cmd) {
        case CTXT_SWITCH:
            req_ctxt_switch(thread_id); //requests
            for context switch
            break;
        case SIGNAL_OTHER-THREADS:
            Increment_curr_clk(thread_id); //this
            will increment the current clk for
            the given thread id.
            signal_all_other_threads(thread_id);
            break;
        case GET_CURR_CLK:
            get_curr_clk(); //returns the current
            vector clock.
            break;
        case RESET_CLK:
            reset_clk(); //reset the current vector
            clock to zero.
            break;
        case SET_CURR_CLK:
            set_curr_clk(clk); //sets the current
            vector clock with the clk received.
    }
}

//Methods of interest with respect to the ioctl cmds
mem_access check_mem_access_with_trace(thread_id_t tid) {
    ...
    //method internally calls check_mem_acc_perm() with
    current clock time and uses the first valid
    instance vector clock registered for a given thread
    in the trace array.

    //returns e_ma_allowed e_ma_restricted based on the
    check_mem_acc_perm()
}

ctxt_switch_thread(thread_id_t tid) {
    down(threads_sem[tid-1]); //perform semaphore down
    operation respective semaphore.
    /**if the value is already 0 when performing the down,
    the thread waits until the value is positive.**/
}
```

```

signal_all_other_threads(thread_id_t tid) {
    //critical section for wait queue
    for i in(0,THREAD_COUNT) {
        if(i!=(tid-1)) {
            if(check_mem_access_with_trace(i+1) ==
               e_ma_allowed) {
                /**Performs up operation on the
                 respective thread
                 semaphore.*/
                up(threads_sem[i]);
                wait_queue[i]=0;
            }
        }
    }

    //critical section ends.
}

req_ctxt_switch(thread_id_t tid) {
    if(check_mem_access_with_trace(tid) ==
       e_ma_restricted) {

        signal_all_other_threads(tid);

        //critical section for waitqueue
        wait_queue[tid-1] = 1; //sets the thread inline
                               for waiting
        //critical section ends.

        ctxt_switch_thread(tid);
    }
}

```

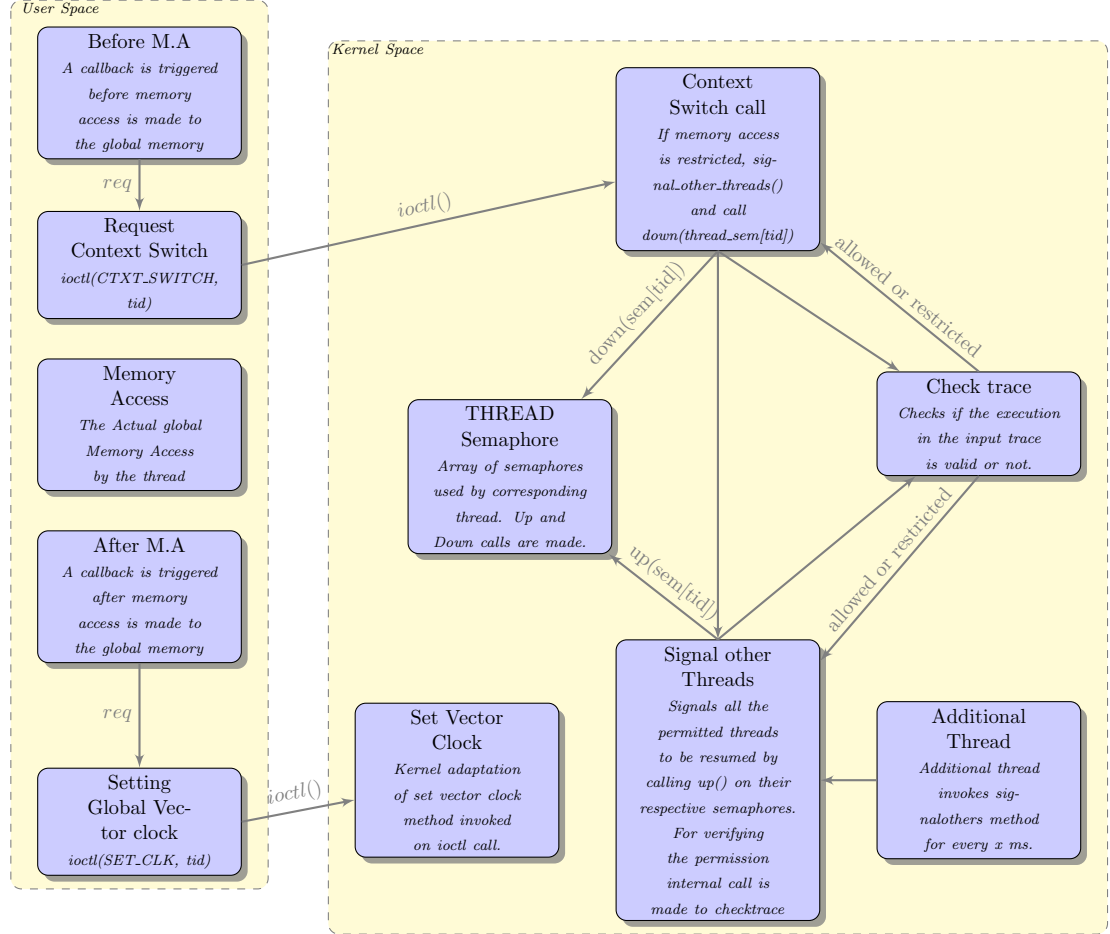
## 1.2 Design with an additional scheduler thread

In this design, we create an additional scheduler thread primarily addressing the signaling mechanism pertained in the previous design. By having an additional scheduler thread, we move the entire signaling system to the scheduler thread. Thus, reducing the execution overhead encountered in the user space thread for signaling other threads.

The major change from the previous design apart from additional thread is in the memory assessment block.



## Memory assessment block



## Pseudo Implementation

The major changes are in kernel space code. However, there are minor variations in the *AfterMA()* in user space.

### User Space Implementation

```
//Rest of the code remains the same

AfterMA() {
    ioctl(SET_MY_CLK, thread_id);
}

//Rest of the code remains the same
```

### Kernel Space - General module definitions

```
//code remains the same

signal_permitted_threads() {
    //critical section for wait queue
    for i in(0,THREAD_COUNT) {
        if(wait_queue[i] == 1) {
            if(check_mem_access_with_trace(i+1) ==
               e_ma_allowed) {
                /**Performs up operation on the
                 respective thread
                 semaphore.**/
                up(threads_sem[i]);
                wait_queue[i]=0;
            }
        }
    }
    //critical section ends.
}

module_init() {
    //code remains the same

    kernel_thread tk = create_kernel_thread(
        signal_permitted_threads)
    tk->setTimerCallForEvery(x) //this method will make
        call to signal permitted threads for every x ms.
}

//code remains the same
```

## 2 Design with checking in user space

In the following designs, we address the use of check permission of memory access method both in User Space and Kernel space.

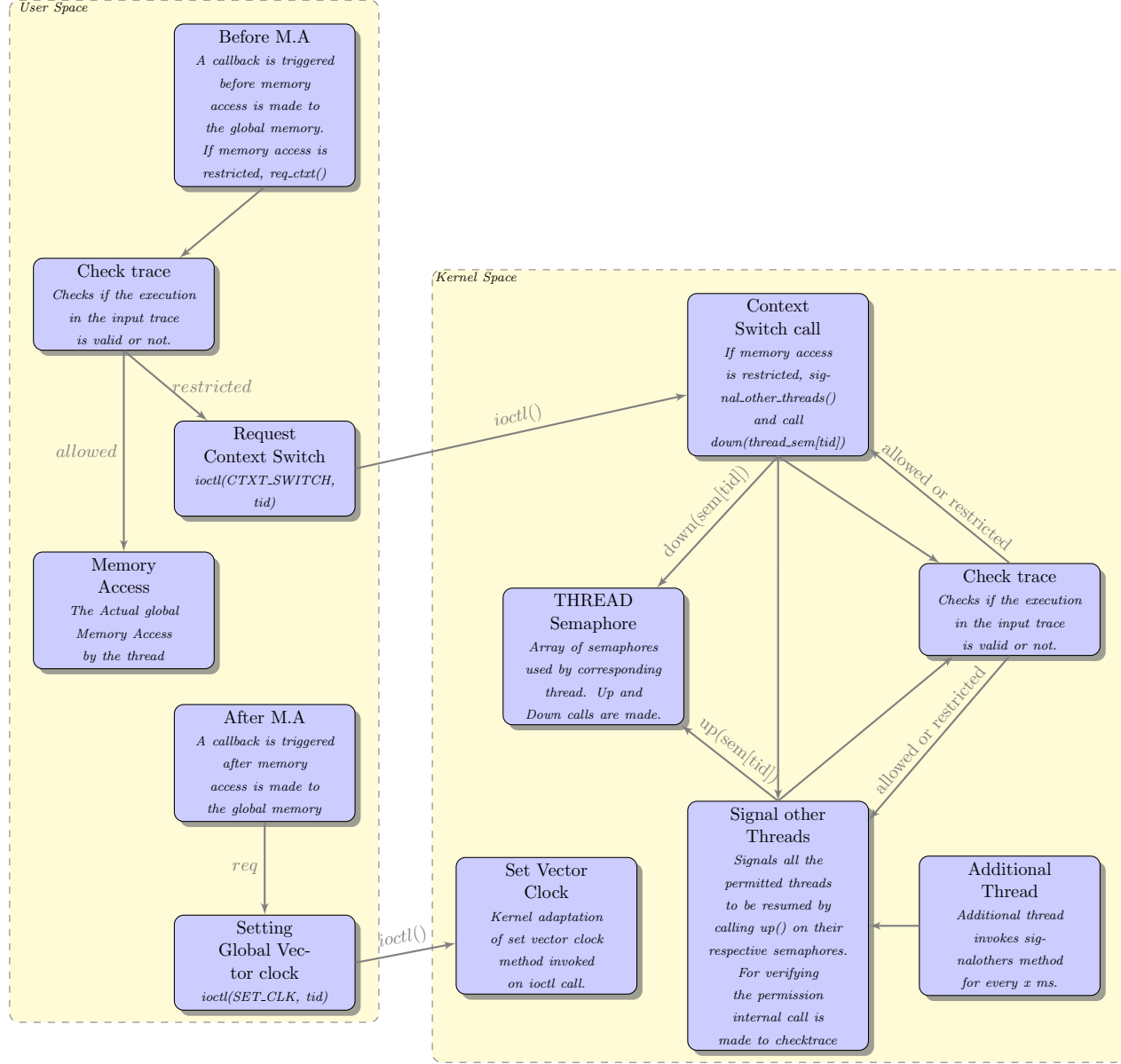
### 2.1 Design with no additional scheduler thread

Without an additional thread in kernel space, the design would require a signaling function inside *AfterMA()*, similar to the one used in Design 1.1. Triggering a signaling mechanism is an additional overhead on the thread calling the *AfterMA()*. Therefore, such a design is not a wise choice when considering the performance metrics such as execution time.

### 2.2 Design with an additional scheduler thread

The scheduler implementation is similar to one defined in the section 1.2. Key difference is the additional checking for memory permissions in the user space.

## Memory assessment block



## Pseudo Implementation

The major changes are in user space code.

## User Space Implementation

```
//Rest of the code remains the same

mem_access ma_status[THREAD_COUNT];
vec_clk curr_clk_time;

initialize_vec_clock() {
    for i in range(0, THREAD_COUNT)
    {
        curr_clk_time.clocks[i] = 0;
    }
}

BeforeMA() {
    ma_status[thread_id-1] = check_mem_access_with_trace(
        thread_id);
    if(ma_status[id-1] == e_ma_restricted) {
        ioctl(CTXT_SWITCH, thread_id);
    }
}

AfterMA() {
    ioctl(SET_MY_CLK, thread_id);
    curr_clk_time.clocks[thread_id-1]++;
}

//Rest of the code remains the same
```

### 3 Variant in blocking implementation

In the previous designs, the blocking was done using semaphores. In the variant design, we use the combination of *schedule()* and *wake\_up\_process()* functions provided by the Linux scheduler APIs. The kernel level tasks associated for the provided user level threads are moved from running queue to wait queue by initially setting the task status as *TASK\_INTERRUPTIBLE* and yielding the processor by invoking *schedule()*. The task added in wait queue is later resumed, when *wake\_up\_process(sleeping\_task)* is invoked by another task. On calling the *wake\_up\_process(sleeping\_task)*, the task status for *sleeping\_task* is set as *TASK\_RUNNING*. It would be pushed to run

queue and executed in future by the operating system scheduler on the basis of scheduler class and priority of tasks in run queue.

### Variant Pseudo Code for Design 1.1

#### Kernel Space - General module definitions

```
//code remains the same.
typedef struct {
    int is_waiting;
    struct task_struct *my_task;
}wait_queue_threads_t;

static wait_queue_threads_t wait_queue[THREAD_COUNT];

module_init() {
    for i in range(0,THREAD_COUNT) {
        wait_queue[i].is_waiting = 0;
        wait_queue[i].my_task = NULL;
        curr_clk[i] = 0;
    }
    alloc_ioctl_device();//method used to allocate ioctl
device.
}

//code remains the same
```

#### Kernel Space - IOCTL

```
//code remains the same

ctxt_switch_thread(thread_id_t tid) {
    //critical section for wait queue
    wait_queue[tid-1].is_waiting = 1;
    wait_queue[tid-1].my_task = current;
    set_current_state(TASK_INTERRUPTIBLE);
    //critical section ends
    schedule();
}

signal_all_other_threads(thread_id_t tid) {
    //critical section for wait queue
    for i in(0,THREAD_COUNT) {
        if(i!=(tid-1)&&(wait_queue[i].is_waiting==1)) {
```

```

        if(check_mem_access_with_trace(i+1) ==
           e_ma_allowed) {
            wait_queue[i].is_waiting = 0;
            wake_up_process(wait_queue[i].
                           my_task);
        }
    }
}

//critical section ends.
}

```

## Variant Pseudo Code for Design 1.2

### Kernel Space - General module definitions

```

//code remains the same.
typedef struct {
    int is_waiting;
    struct task_struct *my_task;
}wait_queue_threads_t;

static wait_queue_threads_t wait_queue[THREAD_COUNT];

module_init() {
    for i in range(0,THREAD_COUNT) {
        wait_queue[i].is_waiting = 0;
        wait_queue[i].my_task = NULL;
        curr_clk[i] = 0;
    }
    alloc_ioctl_device();//method used to allocate ioctl
                        device.
}

//code remains the same

```

### Kernel Space - General module definitions

```

//code remains the same

signal_permitted_threads() {
    //critical section for wait queue
    for i in(0,THREAD_COUNT) {
        if(wait_queue[i].is_waiting==1) {

```

```

        if(check_mem_access_with_trace(i+1) ==
           e_ma_allowed) {
            /**Performs up operation on the
              respective thread
              semaphore.**/
            wait_queue[i].is_waiting = 0;
            wake_up_process(wait_queue[i].
                           my_task);
        }
    }

    //critical section ends.
}

//code remains the same

```