Realizing Iterative-Relaxed Scheduler in Kernel Space

Master-Arbeit
Sreeram Sadasivam
2662284

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik

Fachgebiet Dependable Embedded
Systems and Software
Prof. Neeraj Suri Ph.D

Realizing Iterative-Relaxed Scheduler in Kernel Space
Master-Arbeit
2662284

Eingereicht von Sreeram Sadasivam
Tag der Einreichung: 16. Marz 2018

Gutachter: Prof. Neeraj Suri Ph.D
Betreuer: Patrick Metzler

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Dependable Embedded Systems and Software
Prof. Neeraj Suri Ph.D

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Master-Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 16. Marz 2018 Sreeram Sadasivam

Contents

List of Figures

List of Tables

Abstract

Abstract comes here...

1 Background

—background information related to thesis comes here—

1.1 Software Verification

Software programs are becoming increasingly complex. With the rise in complexity and technological advancements, components within a software have become susceptible to various erroneous conditions. Software verification have been perceived as a solution for the problems arising in the software development cycle. Software verification is primarily verifying if the specifications are met by the software.

1.1.1 Automated Software Verification for Concurrent Programs

Partial Order Reduction

Lipton

Dynamic POR

1.2 Model Checking

1.3 Symbolic Execution

1.4 Iterative Relaxed Scheduling

1.5 Deterministic Multi-Threading

2 Related Work

—related work comes here—

3 Design Challenges

3.1 Overview

This document addresses various design variants and challenges embedded with the proposed solution of moving a fine-grained scheduler to kernel space.

Note

In the progression of this document, we would be using certain acronyms to indicate certain meanings. Some of them are:

- UTID - User defined thread ID which is relative inside the user program.
- RTID - Real Thread ID which is assigned within the proc file system for any thread created within the user land.
- TaskID - All threads are internally realized as tasks in kernel space and are allocated with an identifier which is task ID.

3.2 Design Decisions and Challenges

3.2.1 Mapping UTID to RTID

The user defined thread ID is mapped to the real thread ID created in the user space. The presence of the RTID can be realized by accessing the corresponding RTID folder location inside the proc file system. There are system call defined in Linux operating systems for obtaining the thread ID of any given thread. getttid() is one such function. Based on the thread library used, suitable functional implementation for thread ID extraction can be used. For POSIX (Portable Operating System Interface) based thread library (PThread) we can use the pthread.self().

3.2.2 Registration of UTID to scheduler

User defined thread ID (UTID) is required to be communicated to the scheduler. And the mapping of task ID to UTID needs to be realized, inorder for the scheduling to be done right. The custom registration proc file communicates the UTID-RTID mapping to the kernel space. The user defined thread writes the mapping of UTID - RTID in the above proc file, which would trigger a callback to the write function in the kernel space module. The invocation to the registration module can be done two ways.

Method 1

Single thread can collect the information about mapping of UTID - RTID of all the threads running the user program context. And this thread can invoke the registration module in kernel space.

This would require all the threads to communicate their mapping of UTID - RTID individually to the collector thread. The concept is similar to Gather function in MPI. The collector thread would be blocked until all threads have communicated their mapping information.

Method 2

Threads will be created based on the user's choice. On thread creation, the threads would invoke the registration module individually. This method would require a definition of synchronization block inside the kernel space since, multiple write function calls are invoked. Multiple threads are accessing the registration module. The synchronization is also required between the context_switch module and registration module.

3.2.3 Data Structure for mapping UTID to task ID

The mapping of UTID - task ID is realized, when the registration of a UTID to the scheduler is done. In the registration, the user thread is required to pass the UTID and RTID. The task ID is obtained by passing the RTID to pid_task() function. The data structure is created to store the mapping of UTID to task ID. An item in the data structure is created whenever a registration of UTID takes place. An item is otherwise accessed during the invocation of context_switch() function. In a user space environment, there are solutions such as dictionary mapper or even hash table designs. Since the mapping is coherent in the kernel space, there is only one design choice - linked list. There is a complexity associated in accessing a node in the linked list, which is O(n).

3.2.4 Communication between user thread and kernel space scheduler during context switch

With the transition of scheduler to kernel space, there is a need of having a communication design to interact between the user program and kernel space scheduler. The communication can be dealt with many ways. Some of them are:

- ProcFs - Virtual file system for handling process and thread information base.
- Netlink - Special IPC scheme between kernel space and user space which uses sockets.
- Syscall - Functional implementation mainly meant to communicate some data or perform a specific service in kernel space.
- CharacterDevice - Special buffering interface provided for communicating with character device driver setups.
- Mmap - Fastest way of copying data between kernel space and user space without explicit copying.
- Signals - Unidirectional communication. Communicated from kernel space to user space.
- Upcall - Execute a certain function defined in the user space from kernel space.

3.2.5 Mapping the trace object to kernel space

Trace object inside the framework is needed to be mapped on to the kernel space with the same memory mapping. Currently, the trace object implemented in the framework is realized as a class with many member variables and functions. For realizing the same in the kernel space, the object needs to be remapped in the kernel space when it is received as an object. Since, there are no classes in C but only structures.

3.2.6 Trace verification inside user program vs kernel space scheduler

On occurrence of an event (in this case a memory access of a global variable), the respective callbacks from the user program would trigger a system call to the kernel module. Such a design would facilitate towards a non-preemptive scheduler. By overcoming the additional synchronization overhead existent in the user space design, we encounter the problem of invoking system calls for accessing the kernel module. In a monolithic kernel architecture, most of the system calls are blocking synchronous calls to the kernel space. Having too many system calls would increase the scheduler overhead on the program execution. One solution is to make system calls when there is an imminent context switch (expected thread switch in the provided safe schedule). The user space threads would assess the safe schedules or traces based on which the system calls for the kernel space scheduler would be made.

3.2.7 Yield to scheduler vs Pre-emptive scheduler

The current implementation uses a non-preemptive design for the scheduler. The design uses the verification of memory access event and performs yield to scheduler when the access to memory is not permitted. A pre-emptive design would reduce the communication between user space and kernel space during context switch but, would increase the same for every memory access events. With such an implementation, it would require the kernel space to be able to detect the memory access events of the global memory used by the user space threads. Considering the complexity of its implementation and lack of existing solutions such a design would be not feasible to implement.

3.2.8 Vector clock design for finding the event in the trace

Before a memory access (events triggered on accessing a global memory) is made, the user thread triggers a callback - BeforeMA() (in short before memory access). The callback internally triggers a yield to scheduler if the memory access is not permitted. The memory access permission is determined by checking the trace object. The timeline of the event is required to be addressed during the checking with the trace. The event timeline can be determined by having a vector clock design. The same vector design needs to be used inside the kernel space as well, for its trace verification function.

4 Designs

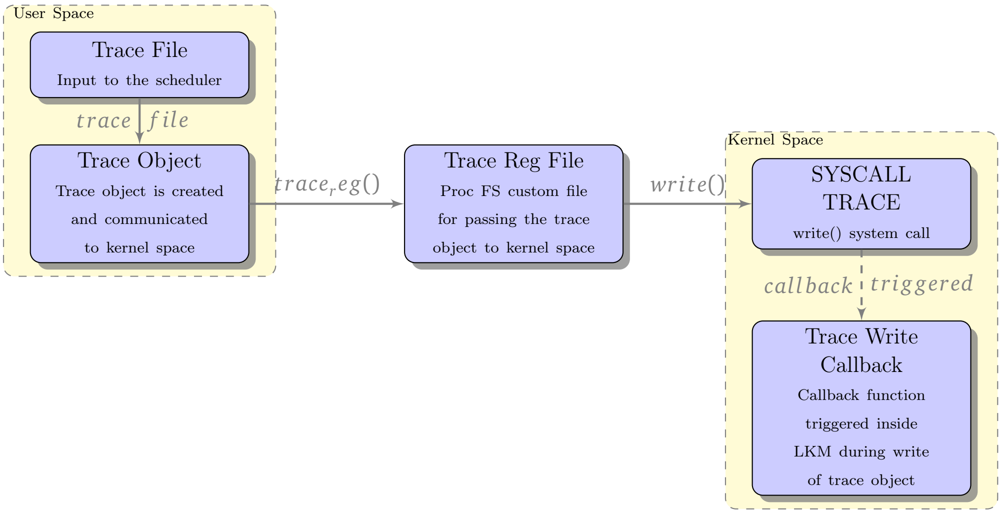
4.1 Design with no checking in user space

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

4.1.1 Design with no additional scheduler thread

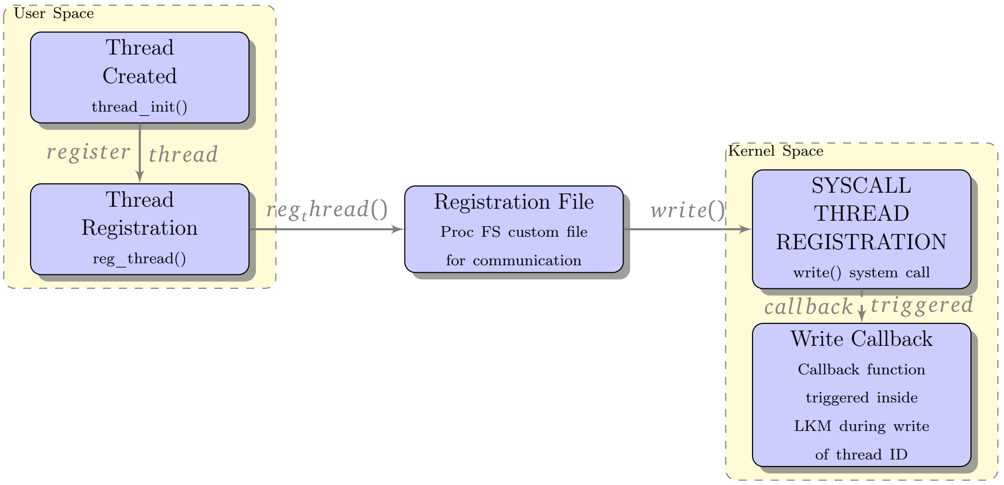
The design described in this section addresses the use of no additional scheduler thread.

Trace Registration


User Space
Trace File
Input to the scheduler
trace file
Trace Object
Trace object is created and communicated to kernel space
trace_reg()
Trace Reg File
Proc FS custom file for passing the trace object to kernel space
write()
Kernel Space
SYSCALL TRACE
write() system call
callback triggered
Trace Write Callback
Callback function triggered inside LKM during write of trace object

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.

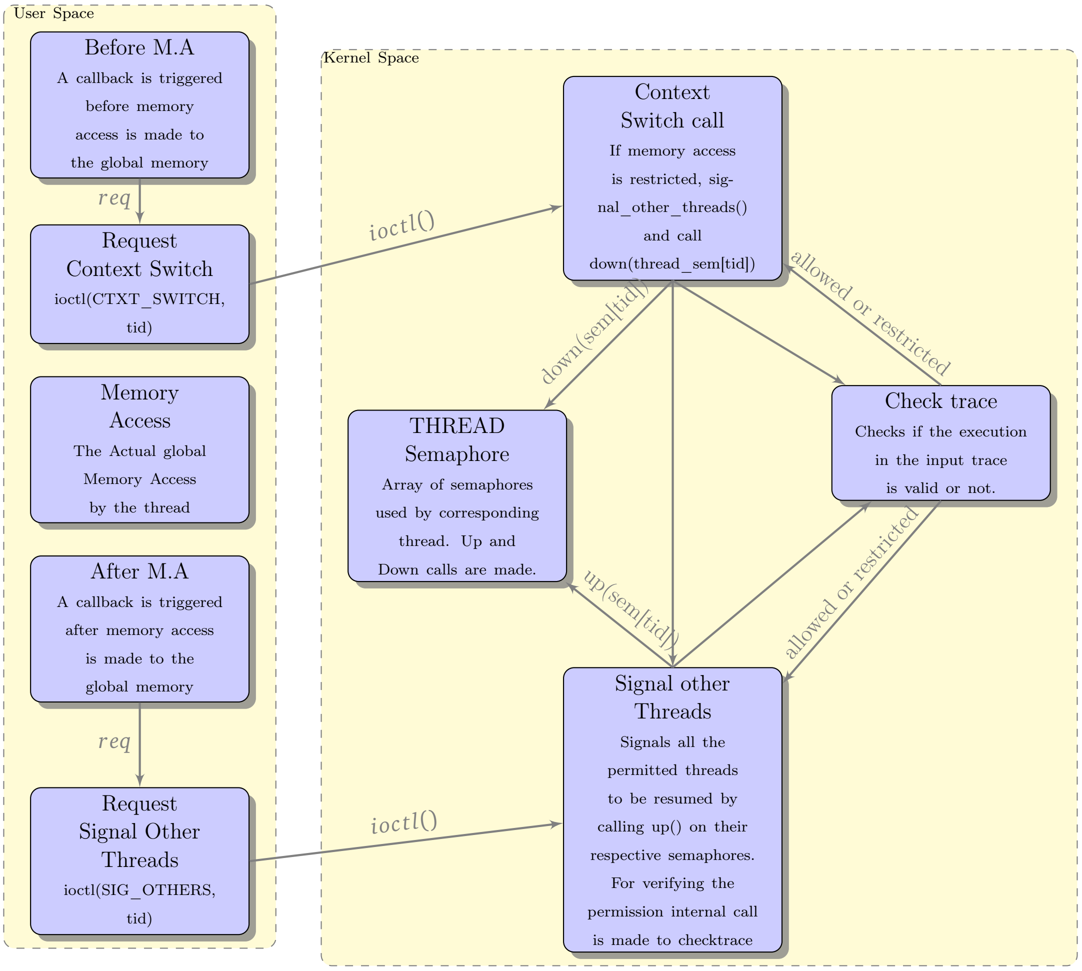
Thread Registration



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.


User Space
Before M.A
A callback is triggered before memory access is made to the global memory
req
Request Context Switch
ioctl(CTXT_SWITCH, tid)
Memory Access
The Actual global Memory Access by the thread
After M.A
A callback is triggered after memory access is made to the global memory
req
Request Signal Other Threads
ioctl(SIG_OTHERS, tid)
Kernel Space
ioctl()
Context Switch call
If memory access is restricted, signal_other_threads() and call down(thread_sem[tid])
down(sem[tid])
allowed or restricted
THREAD Semaphore
Array of semaphores used by corresponding thread. Up and Down calls are made.
Check trace
Checks if the execution in the input trace is valid or not.
up(sem[tid])
allowed or restricted
ioctl()
Signal other Threads
Signals all the permitted threads to be resumed by calling up() on their respective semaphores. For verifying the permission internal call is made to checktrace

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_sem[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

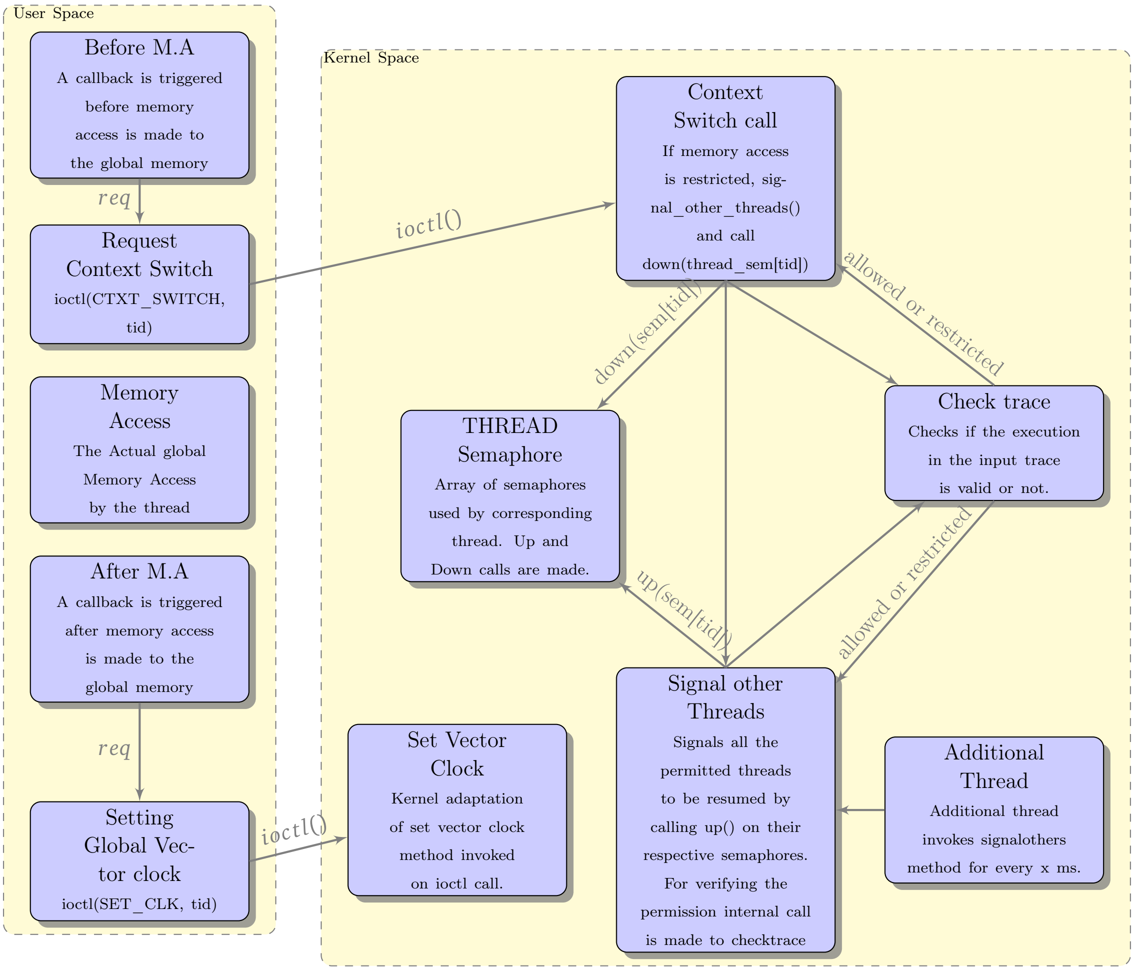
```
/* This method is triggered whenever ioctl commands are issued from the user space
    */
ioctl_access(IOCTL_CMDS cmd) {
        switch(cmd) {
                case CTXT_SWITCH:
                        req_ctxt_switch(thread_id);//requests for context switch
                        break;
                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);

        }
}
```

4.1.2 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


User Space
Before M.A
A callback is triggered before memory access is made to the global memory
req
Request Context Switch
ioctl(CTXT_SWITCH, tid)
Memory Access
The Actual global Memory Access by the thread
After M.A
A callback is triggered after memory access is made to the global memory
req
Setting Global Vector clock
ioctl(SET_CLK, tid)
ioctl()
ioctl()
Kernel Space
Context Switch call
If memory access is restricted, signal_other_threads() and call down(thread_sem[tid])
down(sem[tid])
allowed or restricted
THREAD Semaphore
Array of semaphores used by corresponding thread. Up and Down calls are made.
Check trace
Checks if the execution in the input trace is valid or not.
up(sem[tid])
allowed or restricted
Set Vector Clock
Kernel adaptation of set vector clock method invoked on ioctl call.
Signal other Threads
Signals all the permitted threads to be resumed by calling up() on their respective semaphores. For verifying the permission internal call is made to checktrace
Additional Thread
Additional thread invokes signalothers method for every x ms.

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
```

4.2 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.

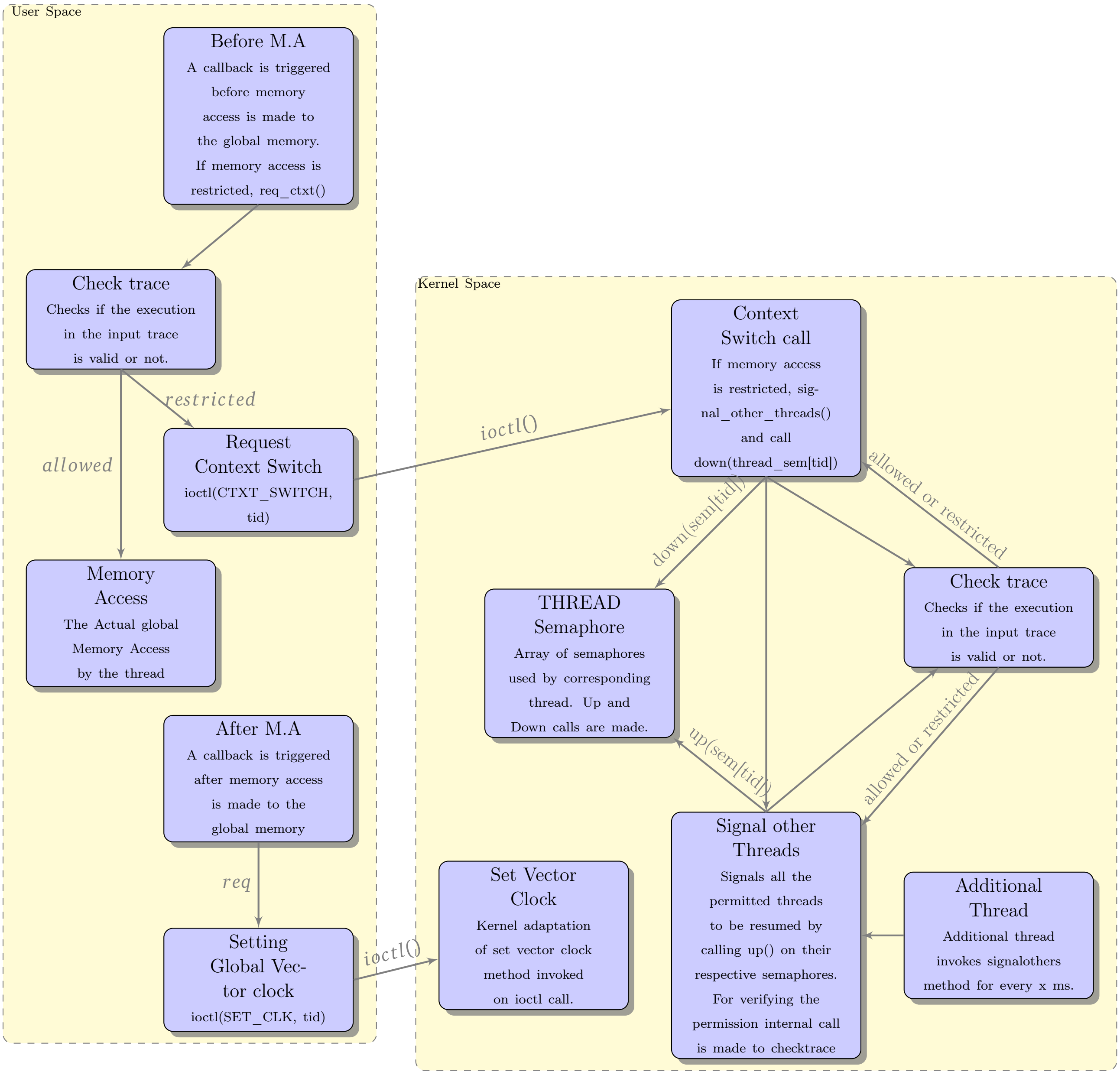
4.2.1 Design with no additional scheduler thread

Without an additional thread in kernel space, the design would require a signaling function inside AfterMA(), similar to the one used in Design 4.1.1. 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.

4.2.2 Design with an additional scheduler thread

The scheduler implementation is similar to one defined in the section 4.1.2. Key difference is the additional checking for memory permissions in the user space.

Memory assessment block


User Space
Before M.A
A callback is triggered before memory access is made to the global memory. If memory access is restricted, req_ctxt()
Check trace
Checks if the execution in the input trace is valid or not.
restricted
allowed
Request Context Switch
ioctl(CTXT_SWITCH, tid)
Memory Access
The Actual global Memory Access by the thread
After M.A
A callback is triggered after memory access is made to the global memory
req
Setting Global Vector clock
ioctl(SET_CLK, tid)
ioctl()
Kernel Space
ioctl()
Context Switch call
If memory access is restricted, signal_other_threads() and call down(thread_sem[tid])
allowed or restricted
down(sem[tid])
THREAD Semaphore
Array of semaphores used by corresponding thread. Up and Down calls are made.
Check trace
Checks if the execution in the input trace is valid or not.
up(sem[tid])
allowed or restricted
Set Vector Clock
Kernel adaptation of set vector clock method invoked on ioctl call.
Signal other Threads
Signals all the permitted threads to be resumed by calling up() on their respective semaphores. For verifying the permission internal call is made to checktrace
Additional Thread
Additional thread invokes signalothers method for every x ms.

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
```

4.3 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 4.1.1

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 4.1.2

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
```