Realizing Iterative-Relaxed Scheduler in Kernel Space

Master-Arbeit
Sreeram Sadasivam
2662284

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

Concurrency bugs which are often resident in multi-threaded programs with shared memory designs are difficult to find and reproduce. Deterministic multi-threading (DMT) is one such scheme indicated to resolve the above difficulty. But, DMT presents the challenge of having no scheduling constraints. However, currently there are no such techniques that allow to control the schedule of a multi-threaded program on a fine-grained level, i.e, on the level of single memory accesses. A design with a granularity of single memory accesses would help in enforcing the scheduling constraint. This thesis focuses on moving the scheduling decision to kernel space. Thus, improving the execution time of the user program.

In the existing design, we have a thread scheduler and a verification engine. The verification engine primarily focuses on instrumenting the user code and realizing memory accesses made by various user threads. The set of safe schedules are provided by the verification engine for the given user program. The generated execution pattern is later realized with the thread scheduler, when the user program is executed. The scheduler thread is realized in user space. However, there is a problem of the scheduler thread getting context switched when executed in user space. The operating system scheduler might ignore the scheduling constraint set by the user level scheduler. Moving the scheduler task to the kernel space would help to realize the safe scheduling constraints set by the user. With the migration of scheduler module to the kernel space, there arises certain design changes and challenges.

The approach used in the thesis would be bench-marked on various thread conditional programs such as Indexer, Last Zero, Fibonacci and Dining Philosopher's Problem. These programs enforce the verification of correctness in multi-threaded environment. The evaluation is performed on the execution overhead exerted by the transition to a loadable kernel module. The evaluation will also relate to the number of synchronizations taking place when using the ioctl calls. The above comparison would also cover evaluations across instrumented and un-instrumented code. The scaling of thread count to core count is also considered for the above evaluations. The approach presented in this work is expected to reduce the execution overhead and also some shortcomings generated by its counterpart user-space design.

1 Introduction

—introduction comes here—

2.1 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[14].

There are two fundamental approaches used in software verification - dynamic and static software verification[14]. Dynamic software verification is performed in conjunction with the execution of the software. In this approach, the behavior of the execution program is checked- commonly known as Test phase. Verification is succeeding phase also known as Review phase. In dynamic verification, the verification adheres to the concept of test and experimentation. The verification process handles the test and behavior of the program under different execution conditions. Static software verification is the complete opposite of the previous approach. The verification process is handled by checking the source code of the program before its execution. Static code analysis is one such technique which uses a similar approach.

The verification of software can also be classified in perspective of automation - manual verification and automated verification. In manual verification, a reviewer manually verifies the software. Whereas in the latter approach, a script or a framework performs verification.

Software verification is a very broad area of research. This thesis work is focused on automated software verification for multithreaded programming.

2.2 Multithreaded Programming

Computing power has grown over the years. Advancements are made in the domain of computer architecture by moving the computing power from single-core to multi-core architecture. With such advancement, there were needs to adapt the programming designs from a serialized execution to more parallelizable execution. Various parallel programming models were perceived to accommodate the perceived progression. Multithreaded programming model was one of the designs considered for the performance boost in computing[6].

Threads are small tasks executed by a scheduler of an operating system, where the resources such as the processor, TLB (Translation Lookaside Buffer), cache, etc., are shared between them. Threads share the same address space and resources. Multithreading addresses the concept of using multiple threads for having concurrent execution of a program on a single or multi-core architectures. Inter-thread communication is achieved by shared memory. Mapping the threads to the processor core is done by the operating system scheduler. Multithreading is only supported in operating systems which has multitasking feature.

Advantages of using multithreading include:

- Fast Execution
- Better system utilization
- Simplified sharing and communication

- Improved responsiveness - Threads can overlap I/O and computation.
- Parallelization

Disadvantages:

- Race conditions
- Deadlocks with improper use of locks/synchronization
- Cache misses when sharing memory

2.3 Concurrency Bugs

Concurrency bugs are one of the major concerns in the domain of multithreaded environment. These bugs are very hard to find and reproduce. Most of these bugs are propagated from the mistakes made by the programmer[17]. Some of these concurrency bugs include:

- Data Race
- Order violation
- Deadlock
- Livelock

Non-deterministic behavior of threads is one of the reasons for having the among mentioned bugs. Data race and order violation are classified as race condition bugs. Whereas, deadlock and livelock are classified as lack of progress bugs.

2.3.1 Race Condition

Race condition is one of the most class of common concurrency problems. The problem arises, when there are concurrent reads and writes of a shared memory location. As stated above, the problem occurs with non-deterministic execution of threads.

Consider the following example, you have three threads and they share two variables x and y [6]. The value of x is initially 0.

Thread 1	Thread 2	Thread 3
(1) x = 1	(2) x = 2	(3) y = x

Table 2.1: Race condition example

If the statements (1), (2) and (3) were executed as a sequential program. The value of y would be 2. When the same program is split to three threads as shown in the above Table 2.1, the output of y becomes unpredictable. The possible values of y = {0,1,2}. The non-deterministic execution of the threads makes the output of y non-deterministic. Table 2.2 depicts possible executions for the above multithreaded execution.

The above showcased problem is classified as race condition bug. Ordered execution of reads and writes can fix the problem.

2.3.2 Lack Of Progress

Lack of progress is another bug class observed in multithreaded programs. Some of the bugs under this class include deadlocks and livelocks.

Execution Order	Value of y
(3),(1),(2)	0
(3),(2),(1)	0
(2),(1),(3)	1
(1),(3),(2)	1
(1),(2),(3)	2
(2),(3),(1)	2

Table 2.2: Possible executions

Deadlock

Deadlock is a state in which each thread in thread pool is waiting for some other thread to take action. In terms of multithreaded programming environment, deadlocks occur when one thread waits on a resource locked by another thread, which in turn is waiting for another resource locked by another thread. If a thread is unable to change its state indefinitely because the resource requested by it are being held by another thread, then the entire system is said to be in deadlock[7].

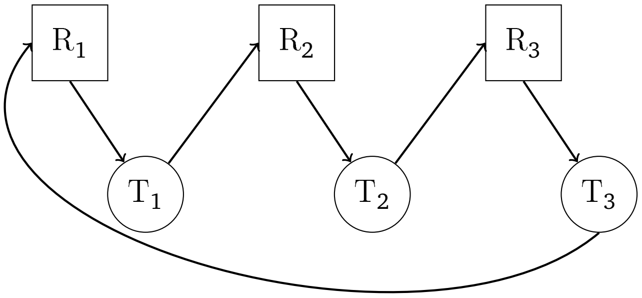


Figure 2.1: Dead Lock Example

In the example depicted in Fig 2.1, we have three threads T_1, T_2, T_3 and three resource instances R_1, R_2, R_3. The figure depicts hold and wait by each threads. Thread T_1 holds resource R_1 and waits for the acquisition of resource R_2 from thread T_2. T_2 cannot relinquish resource R_2, unless it acquires resource R_3 for its progress. But, resource R_3 is acquired by T_3 and is waiting for R_1 from T_1. Thus, making a circular wait of resources. This example clearly explains the dependency of resources for the respective thread progress.

Deadlock can occur if all the following conditions are met simultaneously.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

These conditions are known as Coffman conditions[9].

Deadlock conditions can be avoided by having scheduling of threads in a way to avoid the resource contention issue.

Livelock

Livelock is similar to deadlock, except the state of threads change constantly but, with none progressing. Livelock is special case of resource starvation of threads/processes. Some deadlock detection algorithms are susceptible to livelock conditions when, more than one process/thread tries to take action[17][7]. The above mentioned situation can be avoided by having one priority process/thread taking up the action.

2.4 Model Checking

From section 2.3, it is very clear that there needs to be verification for multithreaded programs. The verification solutions range from detecting causality violations to correctness of execution[11]. Model checking is an example of such a technique. It is used for automatically verifying correctness properties of finite-state concurrent systems[8][3]. This technique has a number of advantages over traditional approaches that are based on simulation, testing and deductive reasoning. When solving a problem algorithmically, both the model of the system and the specification are formulated in a precise mathematical language. Finally, the problem is formulated as a task in logic, namely to verify whether a given structure adheres to a given logical formula. The technique has been successfully used in practice to verify complex sequential designs and communication protocols[8]. Model checker tries to verify all possible states of a system in a brute force manner[2]. Thus, making state explosion as one of the major challenges, which is discussed in detail in section 2.4.1. Model checking tools usually verify partial specification for liveness and safety properties[11]. Model checking algorithms generate set of states from the instructions of a program, which are later analyzed. There is a need to store these states for asserting the number of visits made them are at-most once. There are two methods commonly used to represent states:

- Explicit-state model checking
- Symbolic model checking

Advantages of using model checking:

- Generic verification approach used across various domains of software engineering.
- Supports partial verification, more suited for assessment of essential requirements for a software.
- Not vulnerable to the likelihood that an error is exposed.
- Provides diagnostic information thus, making it suitable for debugging purposes.
- Based on graph theory, data structures and logic thus, making it ‘sound and mathematical underpinning’.
- Easy to understand and deploy.

Disadvantages of using model checking:

- State explosion problem.
- Appropriate for control-intensive applications rather than data-intensive applications.
- Verifies system model and not the actual system.
- Decidability issues when considering abstract data types or infinite state systems.

2.4.1 State explosion problem

The state space of a program is exponential in nature when it comes to number of variables, inputs, width of the data types, etc,. Presence of function calls and dynamic memory allocation makes it infinite[11]. Concurrency makes the situation worse by having interleaving of threads during execution. Interleaving generates exponential number of ways to execute a set of statements/instructions. Thus, having an explosion in state space. There are various techniques used to avoid the state explosion problem.

2.4.2 Explicit-state Model Checking

Explicit-state model checking methods recursively generate successors of initial states by constructing a state transition graph. Graphs are constructed using depth-first, breadth-first or heuristic algorithms. Erroneous states are determined 'on the fly' thus, reducing the state space. A property violation on the newly generated states are regarded as erroneous states. Hash tables are used for indexing the explored states. If there is insufficient, memory lossy compression algorithms are used to accommodate the storage of hash tables[11]. Explicit-state techniques are more suited for error detection and handling concurrency.

2.4.3 Symbolic Model Checking

Symbolic model checking methods manipulate a set of states rather than single states. Sets of states are represented by formulae in propositional logic. It can handle much larger designs with hundreds of state variables. Symbolic model checking uses different model checking algorithms: fix-point model checking(mainly for CTL), bounded model checking(mainly for LTL), invariant checking, etc,. Two main symbolic techniques used - Binary Decision Diagrams(BDD) and Propositional Satisfiability Checkers(SAT solvers). BDDs are traditionally used to represent boolean functions. A BDD is obtained from a Boolean decision tree by maximally sharing nodes and eliminating redundant nodes. However, BDDs grow very large. The issues in using finite automata for infinite sets are analogous. Symbolic representations such as propositional logic formulas are more memory efficient, at the cost of computation time. Symbolic techniques are suitable for proving correctness and handling state-space explosion due to program variables and data types.

2.4.4 Partial Order Reduction

Partial Order Reduction(POR) is a technique used for reducing the size of state space to be searched by a model checking algorithm[20]. This technique exploits the independence of concurrently executed events. Two events are independent of each other when executing them either order results in the same global state[8]. A common model for representing concurrent software is to have it depicted as interleaving model. In interleaving model, we have a single linear execution of the program arranged in an interleaved sequence. Concurrently executed events appear to be ordered arbitrarily to each other. Considering all interleaving sequences would lead to extremely large state space. Constructing full state graph would make the fitting into the memory difficult. Therefore, a reduced state graph construction is used in this technique.

POR exploits the commutativity of concurrently executed transitions, which would result in the same state. Fig 2.2 depicts the commutativity behavior. S, S_1, S_2 and R are various states of a given program and α_1, α_2 represents various transitions. Consider two paths P_1 and P_2. $P_1 = S \rightarrow S_1 \rightarrow R$ and $P_2 = S \rightarrow S_2 \rightarrow R$. P_1 and P_2 reaches the same final state R. Thus, showing us that commutativity of transitions α_1, α_2 on the given example.

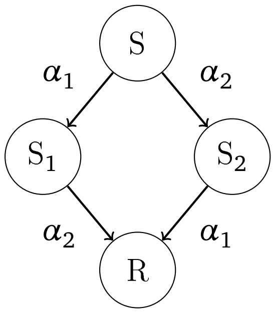


Figure 2.2: Commutativity Example

Partial order reduction derives its motivation from the early versions of algorithms used for partial order modeling of program execution. POR is described as model checking using representatives[19]. Verification is performed using representatives from equivalence classes of behaviors.

The transitions of a system play a major role in the POR. POR is based on the dependency relation that exists between the transitions of a systems. A transition $\alpha \in T$ is enabled in a state s, if there is a state s' such that $\alpha(s,s')$ holds. Otherwise, α is disabled in s. The set of transitions enabled in s is $enabled(s)$. A transition α is deterministic, if for every state s there is at most one state s' such that $\alpha(s,s')$.

A path π from a state s_0 is a finite or infinite sequence.

$\pi = s_0 \rightarrow s_1 \rightarrow ...$

$\alpha_0(s_0,s_1)$, $\alpha_1(s_1,s_2)$ are transitions on the states in path π such that for every i, $\alpha_i(s_i,s_{i+1})$ holds. If π is finite, then the length of π is the number of transitions in π and will be denoted by $\mid \pi \mid$. Purpose of POR is to reduce the number of states, while preserving the correctness of the program. A reduced state graph is generated using depth-first or breadth-first search methods. Model checking algorithm is applied to the resultant graph, which has fewer states and edges.

An independence relation $I \subseteq T \times T$ is a symmetric, anti-reflexive relation such that for $s \in S$ and $(\alpha, \beta) \in I$:

- Enabledness If $\alpha, \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.
- Commutativity $\alpha, \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The dependency relation D is the complement of I, namely $D = (T \times T) \setminus I$. The enabledness condition states that a pair of independent transitions do not disable one another. However, that it is possible for one to enable another. Stuttering refers to a sequence of identically labeled states along a path. In fig 2.2, we have two paths P_1 and P_2 which are stuttering equivalent. Thus, the reduced graph would have fewer number of states and retains the correctness property of the model.

Two main POR techniques which are commonly considered: persistent/stubborn sets and sleep sets. Persistent set technique computes a provably-sufficient subset of the set of enabled transitions in each visited state such that unselected enabled transitions are guaranteed not to interfere with the execution of those being selected. The selected set is called a persistent set. Whereas, the most advanced algorithms are based on stubborn sets. These algorithms exploit information about “which communication objects in a process gets committed to in a given set of operations in future”[13]. Such an information is generally obtained from static code analysis. The sleep set technique exploits information on dependencies exclusively among transitions enabled in the current state along with information recorded about the past of the search. Both the techniques can be used simultaneously and are complementary. Unfortunately, existing persistent/stubborn set techniques suffer from a severe fundamental limitation in the context of concurrent software systems. The non-determinism in the execution of the concurrent programs makes the computation of precision difficult. Sleep sets could be used but, it cannot avoid state explosion. To overcome the above limitations, we have Dynamic POR, which is discussed further in the next section.

2.4.5 Dynamic POR

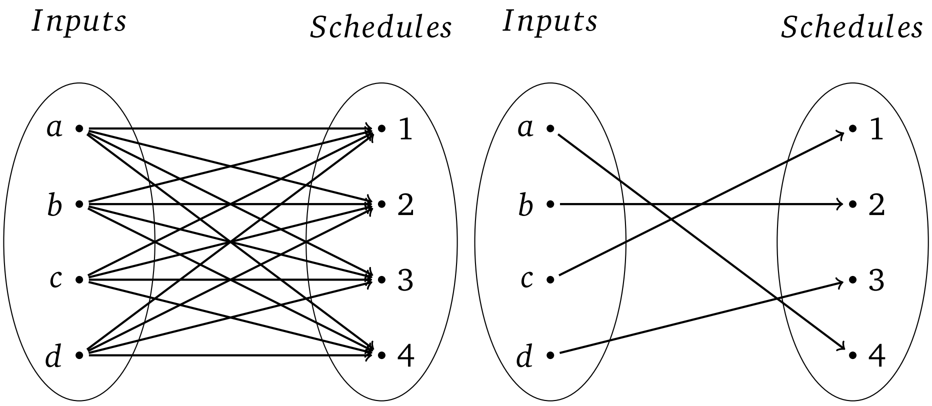
Dynamic POR is a technique, which dynamically tracks interactions between processes and then exploits this information to identify back tracking points where alternative paths in the state space need to be explored[13]. The algorithm works on depth first search in the reduced state space of the system. Dynamic POR helps to calculate dependencies dynamically during the exploration of the state space. It is able to adapt the exploration of the program’s state graph to the precision of having another read or write operation accesses on the same memory location in the same execution path. Dynamic POR algorthm by Flanagan and Godefroid [13], explores single transitions and performs recursive calls subsequently. A persistent set is calculated at each state in the state graph of a system.

2.5 Deterministic Multi-Threading

In section 2.4, we primarily dealt with various ways to verify a program and suggested various techniques, which could be used in a multi-threaded environment. In this section, we discuss about a different approach to deal with the verification of multithreaded programs. In sections 2.2 and 2.3, we discussed about the non-determinism offered by multithreaded programming designs and the bugs associated with them. One way to detect and avoid bugs is to have a constrained scheduling of threads thus, adhering to deterministic execution. Deterministic multi-threading is an approach used to bring in determinism in the execution of multi-threaded programs.

Fig 2.3a depicts a mapping of inputs to possible scheduling pattern adopted by the multi-threaded program. By bringing in deterministic mapping as shown in fig 2.3b, we have direct mapping between inputs and schedules. Having such mapping provides us the opportunity to determine erroneous executions. Such a mapping facilitates to determine concurrency bugs in the program execution. There are many frameworks - CoreDet[4], Parrot[10], Kendo[18], DThreads[16], Grace[5], which adheres to this principle. Some of these frameworks are discussed further in the next chapter.

2.6 Iterative Relaxed Scheduling



(a) Traditional Mapping

(b) Deterministic Mapping

Figure 2.3: Input to Schedule mappings in multithreaded execution

3 Related Work

3.1 COREDET

3.2 PARROT

3.3 KENDO

3.4 DTHREADS

3.5 GRACE

4 Approach

In this chapter, we address the approach used for realizing IRS in kernel space. In the first section, we discuss a theoretical design. In the later sections, we address the potential challenges related to its implementation and the perceived prototypes.

4.1 Theoretical Design

The key component of this thesis is the scheduler. Scheduler handles the scheduling of various user level threads based on their memory access permissions. Memory access permissions are perceived by traces. The traces are realized simple graphs with nodes. Each node denotes a shared memory event for a thread which can be a read or a write event.

4.1.1 Vector Clock

Vector clock is an algorithmic design motivated from Lamport logical clocks [12]. It is used to detect causality violations and generating a partial ordering of events in a distributed system. A vector clock is an array of N logical clocks corresponding to N processes/threads. Vector clocks allow for the partial causal ordering of events. The following definition holds:

- $VC(x)$ denotes the vector clock of event x, and $VC(x)_a$ denotes the component of that clock for process a.
- $VC(x) < VC(y) \iff \forall a[VC(x)_a \lneq VC(y)_a] \land \exists b[VC(x)_b < VC(y)_b]$
- $x \rightarrow y$ indicates event x happened before event y. It is defined as : if $x \rightarrow y$, then $VC(x) < VC(y)$

4.2 Design Challenges

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.
- API -Application Programming Interface.

4.2.1 Mapping UTID to Task Object

UTID is passed to kernel space via a custom proc file and invoking scheduler API - $get\_current\_task()$. This function returns a task struct object. In kernel space, we can have a mapping of UTID to the obtained task struct object. User defined thread ID (UTID) is required to be communicated to the scheduler. And the mapping of task struct to UTID needs to be realized, in order for the scheduling to be done right. The custom registration proc file communicates the UTID to the kernel space. The user defined thread writes the UTID in the above proc file, which would trigger a callback to the write function in the kernel space module. 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 scheduler module and registration module.

4.2.2 Data Structure for mapping UTID to Task Object

The mapping of UTID - task object 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. The task object is obtained by invoking $get\_current\_task()$ function during registration call. The data structure is created to store the mapping of UTID to task object. 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, possible design choices include - linked list, arrays. There is a complexity associated in accessing a node in the linked list, which is O(n).

4.2.3 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 in many ways[15]. Some of them are:

- ProcFS - Virtual file system for handling process and thread information base. Useful for small and short communications.
- Netlink - Special IPC scheme between kernel space and user space which uses sockets. Portable design and
- System call - 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. Useful for large transactions of data.
- Signals - Unidirectional communication. Communicated from kernel space to user space.
- Upcall - Execute a certain function defined in the user space from kernel space.
- IOCTL - Used primarily for input and output operations in between user space and kernel space. It is an extension of character device implementation. It uses simple read and write system calls for communication purposes. It can be realized as an alternative for system call.

Assessing the requirements for the implementation, IOCTL seems to be a perfect fit for all the interactions required for a scheduler module. System call implementation requires the building of the entire kernel source tree and they are very difficult to debug and develop. IOCTL provides the possibility for a plug and play design.

4.2.4 Mapping the trace object to kernel space

The proposed design uses vector clocks as an outline for trace implementation. The traces generated as graphs are mapped to kernel space a struct object. Graphs are realized as graphviz files. Parsing of graph is required in order to be mapped to the kernel space. The parsed graph is passed to the kernel space as a long string via a custom proc file. Currently, there is no automated method existent in this thesis to generate a graph string from a graphviz file. We generate the trace string manually and pass it as an input to the custom proc file when the user program starts.

4.2.5 Trace verification inside user program vs kernel space scheduler

On occurrence of a shared memory event, the respective callbacks (BeforeMA() & AfterMA() - Before memory access and After memory access) 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 IOCTL calls for accessing the kernel module. In a monolithic kernel architecture, most of the IOCTL calls are blocking synchronous calls to the kernel space. Having too many IOCTL calls would increase the scheduler overhead on the program execution. One solution is to make IOCTL calls when there is an actual need of a context switch. The user space threads would assess the trace based on which the IOCTL calls for the kernel space scheduler would be made. We discuss about such a solution in two prototypes used in the implementation section.

Note - Non-preemptiveness indicated in this section and the rest of the document is in regard to the scheduler implemented in this thesis and not the OS Scheduler.

4.2.6 Yield to scheduler vs Preemptive 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 preemptive 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.

4.2.7 Vector clock design for finding the event in the trace

Before a shared memory access 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.

4.3 Synchronization Designs

We classify the designs in two classes for our convenience. The classification is based on the checking for memory permission in user space. The first class has no checking for memory permission in the user space and the second has a proxy checking in user space for memory permission.

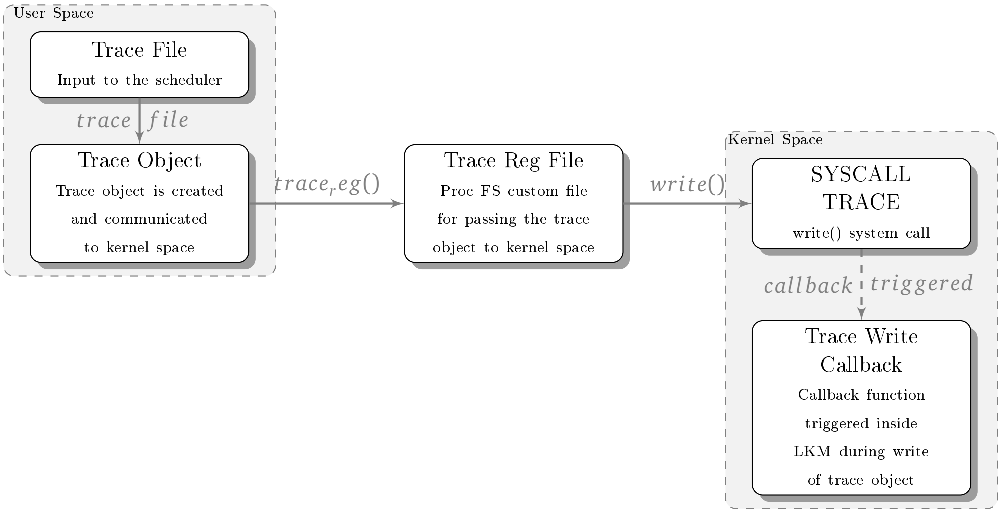
4.3.1 Design with no checking in user space

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

Design with no additional scheduler thread

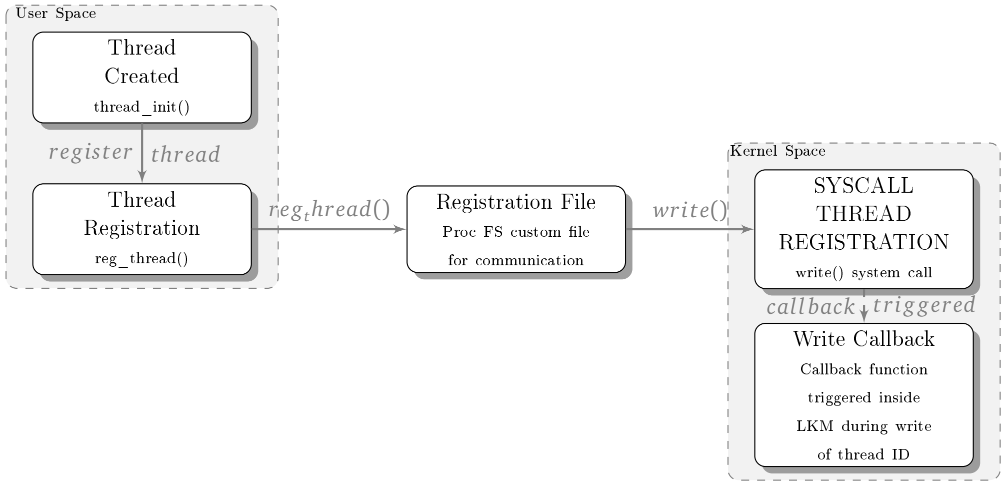
The design described in this section addresses the use of no additional scheduler thread.

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.

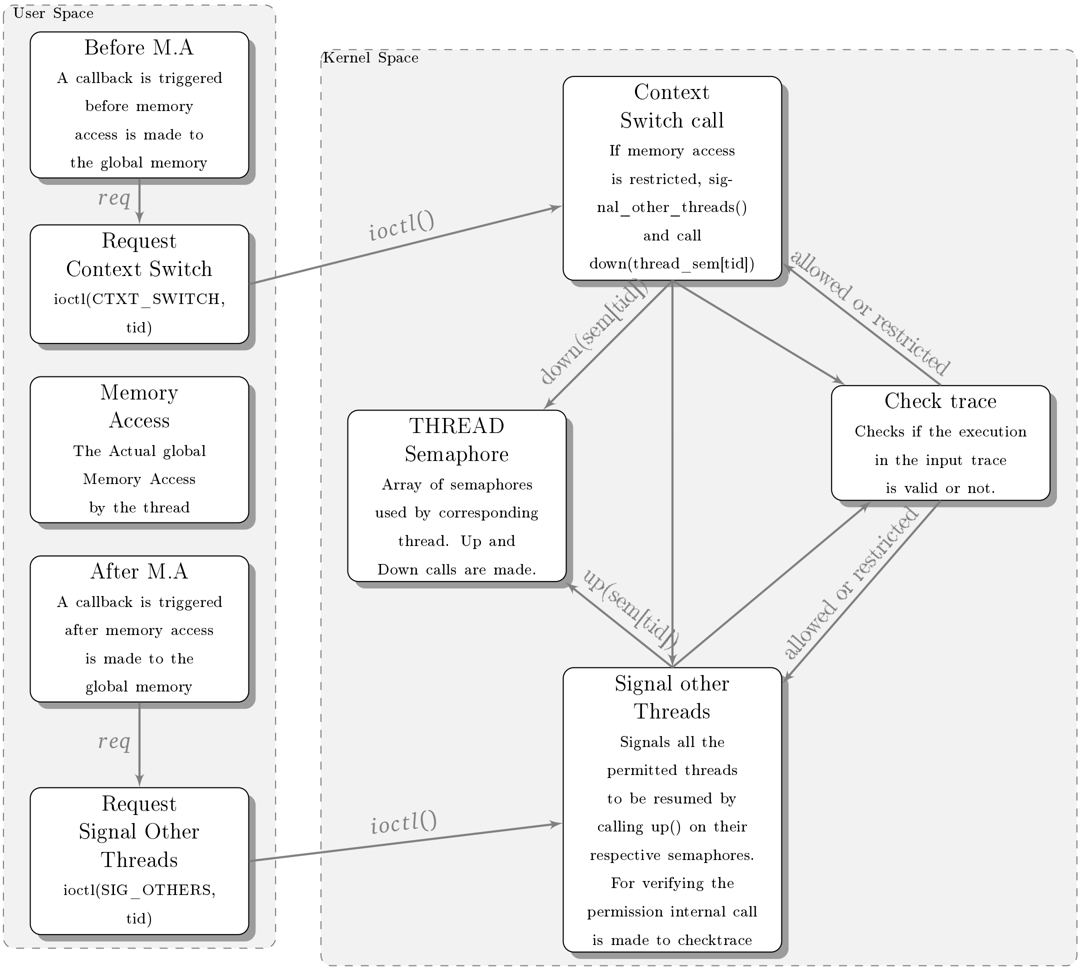
Thread Registration



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.


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])
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
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
}
```

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 - 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
        up(threads_sem[i])$\forall$
        //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
```

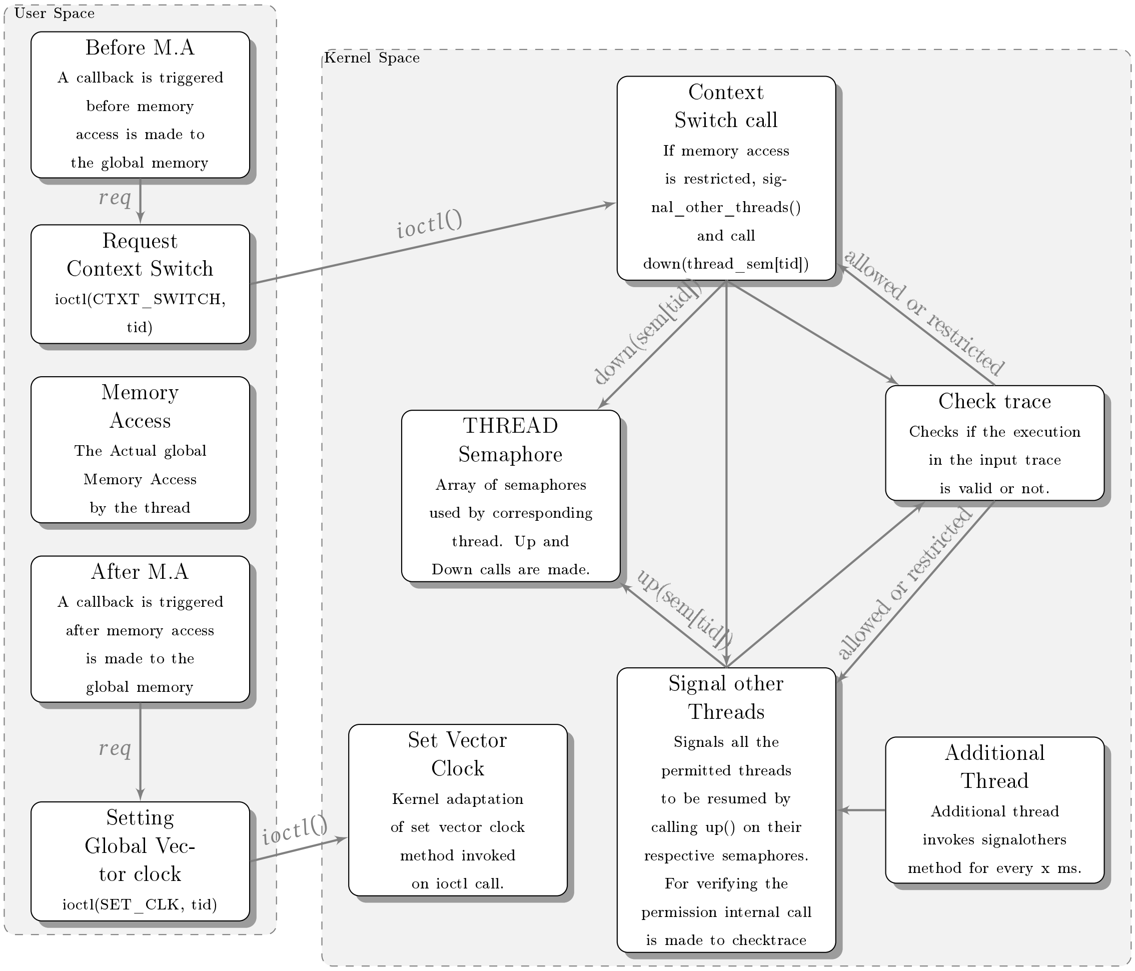
```
                    wait_queue[tid-1] = 1; //sets the thread inline for waiting
                    //critical section ends.

                    ctxt_switch_thread(tid);

          }
}
```

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])
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 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
```

4.3.2 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.

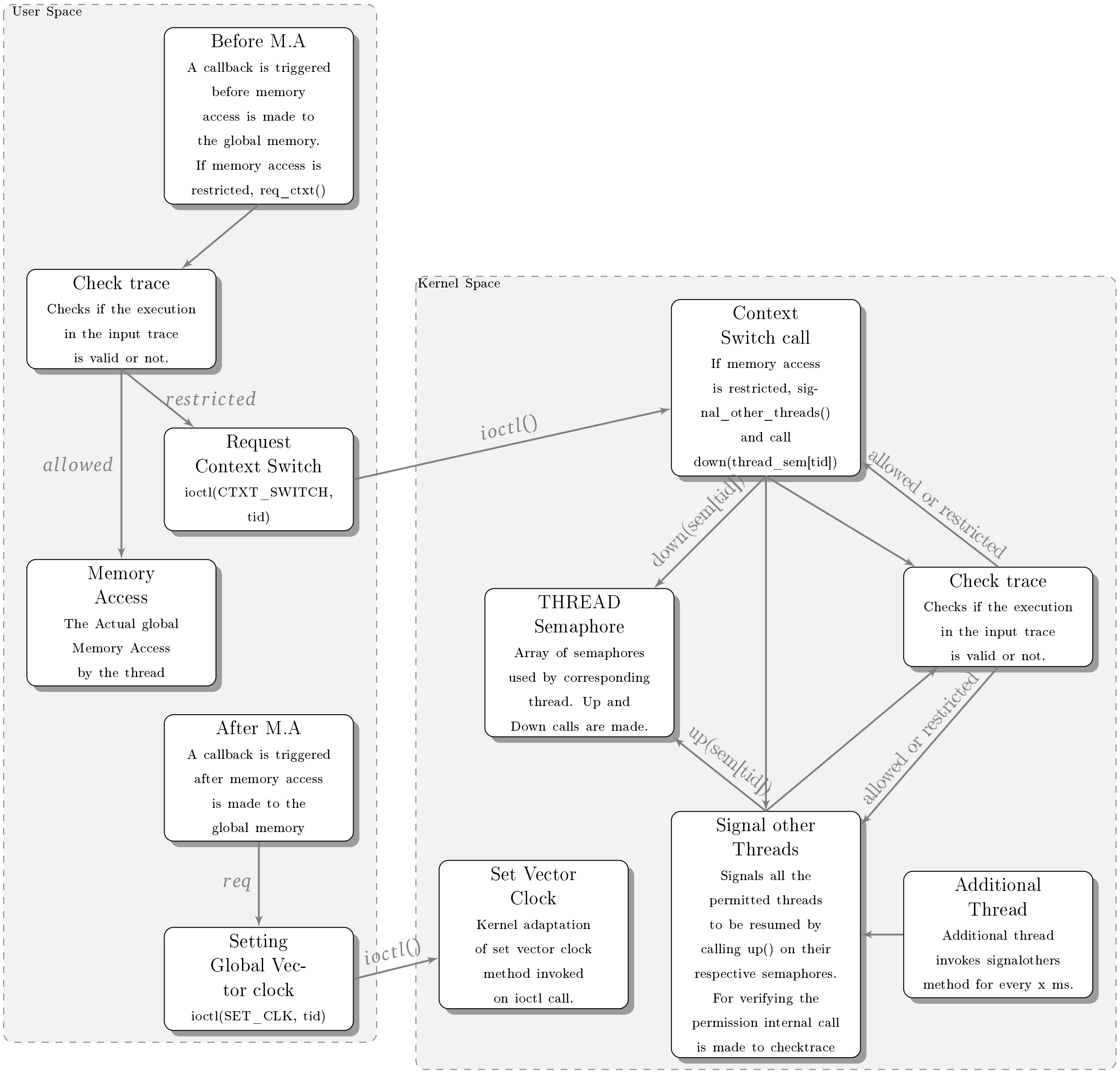
Design with no additional scheduler thread

Without an additional thread in kernel space, the design would require a signaling function inside AfterMA(), similar to the one used in Design 4.3.1. 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.

Design with an additional scheduler thread

The scheduler implementation is similar to one defined in the section 4.3.1. 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])
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 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
```

4.3.3 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 4.3.1

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 4.3.1

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
```

5 Evaluation

In this chapter, we provide an evaluation proof of using IRS in kernel space. All the evaluations are done in regard with the user space implementation.

5.1 Setup

The evaluation is performed on a virtual machine running on cluster with a maximum of 128 cores. The virtual machine can use from two to eight processor cores which is used for the scaled up evaluation. The virtual machine is based on Intel Xeon Family of processors configured with Ubuntu 16.10 as the operating system. It is configured with 4GB RAM and 80GB hard disk. The virtual machine is configured with the LLVM-CLANG 3.9, GCC 4.9 and Boost 1.6.2.

5.2 Evaluation Metrics

- Execution Overhead - Evaluation is done between the IRS user space solution vs kernel space solution. Execution overhead generated by either of the designs with respect to the un-instrumented execution is used as an evaluation criteria.
- Number of valid synchronization calls - It is used to realize the number of voluntary calls made to kernel space for synchronization purposes. It is primarily the number of IOCTL calls under the command - context_switch, signal_all_other_threads or set_clock.

5.3 Benchmarks

We use four different bench-marking programs for the evaluation of this thesis. The bench-marking programs include:

- Fibonacci - Program runs with two threads computing Fibonacci numbers for 25 iterations per thread.
- Last Zero - Program runs with 16 threads [1].
- Indexer- Program runs with 15 threads [13].
- Dining Philosophers Problem - Program runs with 16 threads. These threads are classified as odd and even philosophers thread. In this benchmark, only one class of philosopher thread is active at any point of time. This benchmark is motivated from the solution presented in Silberschatz et al. [21].

5.4 Voluntary kernel level calls

We evaluate the number of voluntary calls made to kernel space for synchronization. The evaluation is done across all six prototypes. The benchmark used for the evaluation is Fibonacci. The Fibonacci benchmark presents different levels of memory constraints via its traces. It has three traces providing 98 constraints, 44 constraints and 24 constraints respectively.

	Prototype 1-4	Prototype 5-6
Trace-1	300	175
Trace-2	300	150
Trace-3	300	150

Table 5.1: Number of IOCTL calls

	Prototype 1-4	Prototype 5-6
Trace-1	150	27
Trace-2	150	0
Trace-3	150	0

Table 5.2: Number of context switch calls

From the tables 5.1 and 5.2, it is really evident that prototypes 5 and 6 reduce the number of calls made to kernel space. Prototypes 5 & 6 are expected to provide better performance compared other prototypes when there are less dependencies between threads even with high number of memory events. Let us consider the Fibonacci benchmark, it has two threads with a total of 75 shared-memory events per thread. Thus, making a total of 150 memory events. For every memory event, prototypes 1-4 trigger IOCTL calls to kernel space for context_switch, signal_all_other_threads or set_clock. Therefore, having number of IOCTL calls as 300. In case of prototype 5-6, we have a proxy checking in user space which drastically reduces the calls to kernel space for additional synchronization. The set_clock ioctl command is the only call made consistently for every memory access when using prototypes 5-6.

	Proto-1	Proto-2	Proto-3	Proto-4	Proto-5	Proto-6
Trace-1	406.833	454.785	385.416	455.745	277.793	275.343
Trace-2	367.199	520.352	352.506	509.843	160.266	160.307
Trace-3	351.029	416.653	333.704	412.206	152.425	153.06

Table 5.3: Execution overhead(%) when compared with plain execution of Fibonacci

Table 5.3 presents the reasoning of using prototypes 5-6. It shows the execution overhead is drastically reduced for the above mentioned prototypes. Reduction in the number of IOCTL calls makes a huge difference in the execution overhead. Prototypes 5-6 performs the best, when the following condition holds: $num\_memory\_constraints << total\_memory\_events$.

5.5 Scaled-up Evaluation

In this evaluation, we understand the merits and demerits in the performance of the realized prototypes and the two user space IRS implementations. For this evaluation, we use three benchmarking programs - last zero, indexer and dining philosophers problem. We scale the core count from two to eight processor cores and monitor the changes the performance overhead across the three benchmarks for various implementations.

6 Conclusion

—conclusion comes here—

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. ACM SIGPLAN Notices, 49(1):373–384, 2014.

[2] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. Principles of model checking. MIT press, 2008.

[3] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. Systems and software verification: model-checking techniques and tools. Springer Science & Business Media, 2013.

[4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In ACM SIGARCH Computer Architecture News, volume 38, pages 53–64. ACM, 2010.

[5] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In ACM sigplan notices, volume 44, pages 81–96. ACM, 2009.

[6] Richard H Carver and Kuo-Chung Tai. Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs. John Wiley & Sons, 2005.

[7] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing, 17(4):461–483, 2005.

[8] Edmund M Clarke, Orna Grumberg, and Doron Peled. Model checking. MIT press, 1999.

[9] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. ACM Computing Surveys (CSUR), 3(2):67–78, 1971.

[10] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 388–405. ACM, 2013.

[11] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(7):1165–1178, 2008.

[12] Colin Fidge. Logical time in distributed computing systems. Computer, 24(8):28–33, 1991.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In ACM Sigplan Notices, volume 40, pages 110–121. ACM, 2005.

[14] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. Fundamentals of software engineering. Prentice Hall PTR, 2002.

[15] Ariane Keller. Tldp - communication between user space and kernel space. URL `http://wiki.tldp.org/kernel_user_space_howto`.

[16] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 327–336. ACM, 2011.

[17] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. A study of concurrency bugs and advanced development support for actor-based programs. arXiv preprint arXiv:1706.07372, 2017.

[18] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. ACM Sigplan Notices, 44(3):97–108, 2009.

[19] Doron Peled. All from one, one for all: on model checking using representatives. In International Conference on Computer Aided Verification, pages 409–423. Springer, 1993.

[20] Doron Peled. Ten years of partial order reduction. In Computer Aided Verification, pages 17–28. Springer, 1998.

[21] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating system concepts essentials. John Wiley & Sons, Inc., 2014.