Design decisions and challenges for fine-grained scheduler in kernel space

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

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.

Design Decisions and Challenges

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()*.

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.

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).

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.

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.

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.

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.

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.