# Guaranteeing Mutual Exclusion in Transactional Systems

Chakshu Tandon, Shaleen Garg
*Rutgers University*

## Abstract

Typical transactional systems may have hundreds or thousands of transactions being executed at any point in time. Many of these systems operate on shared objects.

The ACID properties of transactions may be violated by errors that are semantic in nature, uncaught by the compiler.

In this paper, we design and implement a library which provides fail-stop enforcement and easy debugging for maintaining mutual exclusion in transactional systems. This guarantees that transactions maintain their ACID properties.

Our low overhead design and easy to merge library can be quickly used by existing transactional systems.

## 1 Introduction

Transactions are a set of instructions with ACID properties. ACID is an acronym for Atomicity, Consistency, Isolation, and Durability.

Transactional systems are systems that run transactions. The world of business is based on transaction of money and services. Any error in these systems can lead to losses in terms of money, data, or time.

Errors in any transactional system can be classified into syntactic errors and semantic errors. A syntactic error arises when the written program has invalid program syntax or similar mechanical fault. Such an error is easy to detect using existing compilers and thus usually easy to correct.

Semantic errors can occur in programs which are syntactically correct, however, do not produce the desired results and may also leave the system in an unusable state. For instance, consider this small transactional system with two concurrent threads, editing a shared variable *A*.

```
1 void Thread1(int A, sem_t mutex)
2 {
3     sem_wait(&mutex);
4     A += 1;
5     sem_post(&mutex);
6     return ;
7 }
```

```
1 int Thread2(int A, sem_t mutex)
2 {
3     //doesn't wait for mutex lock
4     A *= 12;
5     int a = A *5;
6     return a;
7 }
```

**Figure 1:** Threads running concurrently can lead to unexpected results

The program above is syntactically correct, but as one can notice Thread2 does not write to the shared object *A* safely. It does not use the appropriate locks which can leave the system with values in the shared object to be either unexpected or corrupt, which is not desirable. Here, the properties of durability, consistency, and isolation are violated.

Even today, semantic bugs can sneak into production software. This is largely due to lack of compiler support and adequate testing suites for semantic bugs in transactional systems. Moreover, semantic bugs such as violated mutual exclusion are generally triggered due to race conditions. Making sure all the race-conditions are checked requires brute-force tests which are neither easy to implement nor technically sound due to their inefficiency.

Semantic bugs that sneak into production software, at some point, may lead to an inconsistent state. Even with all the available system logs it is difficult to exactly point out the error in the source code. While the logs may show which entity wrote what data, it can not capture race conditions since all the user threads are at the mercy of the operating system's thread scheduler. Such scheduling information is not exposed to user threads. With larger transactional systems, this problem only amplifies.

This paper provides a solution for fail-stop enforcement and easy debugging for maintaining mutual exclusion in transactional systems. While capturing the semantics of a large transactional system is difficult, intuitively, we want to avoid objects being in an inconsistent state. So, there is some light if we can enforce the ACID properties while writing to shared objects.

The first step in such an approach would be to know shared objects in a given transactional system. Section 2 includes why this is a difficult problem. Section 3 provides a work around for this problem.

The second step is to enforce, at runtime, atomic writes to those shared objects. Section 2 talks about the overheads associated with enforcing atomic writes and our efficient solution for it.

## 2 Background

As a method of solving different problems, we shall introduce the various methodologies and technologies we used.

### 2.1 Detecting Shared Data Structure

Consider the program as listed in figure 2. A reasonable presumption about the value $x$ would be 11. However, function $f$ may change the value of x by casting away the constant. It may also store the address of $x$ in a global variable so that function $g$ can alter the value $x$. A compiler has to be very careful when analysing pointers because they can change with program inputs at runtime.

Based on indecisiveness of pointers, compilers cannot be sure that a given object is shared. Therefore, we resort to help from the programmer to identify shared objects in the given program. With our adaptation of transactional mutual exclusion, we eliminate this problem and identify shared objects at runtime. Section 3 goes into further detail.

```
int *addr_x;

void f(int const&);
void g() {
    *addr_x = 2;
}
int main() {
    int x = 10;
    f(x);
    x += 1;
    g();
    cout << x << "\n";
}
```

**Figure 2:** Program showing the difficulty in using pointers for tracing shared objects

## 2.2 Protecting Memory Objects

The Linux operating system has mechanisms to protect a region of memory from stray reads and writes in the calling process. The mprotect() system call applies protection to all the pages in a given range by setting appropriate bits in the page table entries corresponding to those memory pages. This call is process-global in nature, i.e. any protection applied is visible across all the processes.

The mprotect() system call is a powerful tool to control the write permissions of memory objects at various points in the program execution. But such power does not come without heavy performance overheads. For a large contiguous region of virtual memory an mprotect() call incurs the following overheads:

1. A kernel trap.
2. Updates to hundreds or even thousands of page table entries requiring expensive memory lookups.
3. Processor TLB cache invalidations.

A typical mprotect() call spanning roughly a 1000 pages can incur a latency of approximately 25 $\mu s$. The cost of an mprotect() call for non-contiguous memory regions can be significantly higher. Such a cost can become a very large bottleneck for high-frequency permission updates.

Intel-based CPUs have a new hardware page protection mechanism called Memory Protection Keys (MPK). It is a lightweight mechanism to handle per-thread memory permissions. Each physical thread has a user-accessible, thread-local PKRU (protection key rights for userspace) register which can be changed by the individual thread without privileged rights. A thread can therefore change the protections of memory pages in its address space very quickly. The Linux kernel and glibc provide library support to handle protection keys (pkeys) from user space. Updating memory protections does not require a kernel trap, updates to the page tables, or TLB shoot-downs. Permission changes take ~15 ns. In total, the hardware allows for 16 pkeys [1]. It can be noted, that a child thread inherits all the pkeys from its parent.

Since, there is no need for privileged access to change PKRU registers, and thus memory protection, this mechanism is useful for keeping self-discipline rather than enforced-discipline.

## 2.3 Source Code Compliance

Any solution has a set of assumptions and expectations from its users. In order to make compliance easy and effective, we used a well-known source analysis and compilation tool called LLVM [1].

LLVM provides the ability to write front-end passes for many source languages and target various instruction set architectures (ISA). It also has powerful mechanisms to alter the source code as well as make modifications to its architecture-independent and language-independent intermediate representation (IR). Since transactional systems can be quite large, it is reasonable to expect an automated tool to help in compliance. Using LLVM, we have written compiler passes based on our assumptions listed in section 3 to check source compliance and modify programs accordingly.

# 3 Our Design and Implementation

The objective of our approach is to provide fail-stop enforcement of mutual exclusion and easy debugging in transactional systems. To this end, we have implemented a support library along with a compiler pass to meet that objective.

In this section, we shall look at the high-level architecture and implementation details of our proposal.

```
void TypicalCriticalSection(sem_t mutex, ...)
{
    sem_wait(&mutex);
     // Critical section
    sem_post(&mutex);
}
```

**Figure 3:** Lock: Typical Critical Section

```
void Lock&Perm(sem_t mutex, ...)
{
     sem_wait(&mutex);
     PKRU_WRITE_ENABLE
     // Critical section
     PKRU_WRITE_DISABLE
     sem_post(&mutex);
}
```

**Figure 4:** Lock&Perm: New Lock Semantics

## 3.1 High-Level Architecture

The failure to maintain mutual exclusion occurs when some thread does not properly honor the semantics of a critical section, ie. they do not use locks before writing to shared objects.

[1]There are only 15 protection domains available to user applications. Key zero is used for the default protection domain.

In order to enforce mutual exclusion, for each shared object, we associate a pkey used to provide access control to that object. By default, we disable write-access to the pkey so that no stray writes to the shared objects are possible.

Further, we require a new lock semantic for transactional systems called Lock&Perm to enforce transactional security.

Figure 3 shows how mutual exclusion is typically ensured. Here the issue arises form the fact that there is no memory protection if an erroneous thread does not use semaphores correctly.

Figure 4 shows the change in lock semantics in our approach. In addition to semaphores, we have included enabling write permission on shared objects. The combination of the two enforces mutual exclusion even on threads which have semantic bugs in their source code. Even if a erroneous or malicious thread tries to edit shared object without the appropriate locking mechanism (Lock&Perm), it will incur a segmentation fault since the memory is protected.

```
void LockOnly(sem_t mutex, ...)
{
    sem_wait(&mutex);
    // Critical section
    sem_post(&mutex);
}
```

**Figure 5:** Correct semaphore usage but failure to enable write protection

```
void PermOnly(sem_t mutex, ...)
{

    PKRU_WRITE_ENABLE
    // Critical section
    PKRU_WRITE_DISABLE
}
```

**Figure 6:** No semaphore usage with the PKRU updates

```
void NoLockNoPerm(...)
{
    // Critical section
}
```

**Figure 7:** Using neither semaphore nor write protection

With the new lock semantics, there remain three types of erroneous critical sections:

1. Figure 5 shows a scenario where some thread is only using semaphores and no write permission. If, this thread tries to write to a shared object, it will incur a segmentation fault since it does not have write permissions.

2. Figure 6 shows a scenario where a malicious thread is trying to sneak some updates without first acquiring a lock. The following subsection discusses how we mitigate these classes of errors using an LLVM compiler pass.

3. Figure 7 shows a scenario where a malicious thread is blatantly trying to update some shared object without acquiring a lock or updating the pkeys. Again, this thread will incur a segmentation fault since it does not have write permissions.

**Algorithm 1** Define a Shared Object

```
1: mem = Allocate_Memory(size)
2: pkey = pkey_alloc(WRITE_DISABLE)
3: pkey_mprotect(mem, pkey)
4: return Mem
```

Next we shall show, how the mitigation library is structured and how different error scenarios are handled.

## 3.2 Implementation Details

Our library is a combination of functions and an LLVM pass. For user ease, we provide two library calls to:

**Define a shared object** As shown in section 2 we can not automatically detect shared objects in a program. Our library provides a function to declare shared objects. Using this library function, the user specifies objects which are accessed concurrently.

Algorithm 1 describes the library function which is used to define a new shared object. In addition to allocating memory, the function also allocates a new pkey for that object and disables write access to this object.

**Define a Critical Section** To make it easy for the programmer to use the new lock semantics, we provide a library function which is a wrapper around a user-defined critical function. As arguments, it requires the a reference to the shared object as well as a pointer to a user-defined critical function.

Algorithm 2 illustrates the workings of the wrapper function. Here, after acquiring a lock, write access is enabled through the PKRU register for the shared object and the critical function is called. Before returning, write access on the protection key is disabled and the lock is released.

In addition to the two library calls as described above, we use an LLVM front-end pass to check through the source program. We assume that pkeys are not used for any other purposes in the system. So, if there exist any usage of the PKRU register, which are not from our library, then they must be removed since they are considered to be erroneous.

The LLVM pass goes through all the functions, other than those explicitly marked by our library, and warns the programmer of any stray PKRU updates. Stray PKRU writes are ones that do not originate from the library functions. It also edits the source to make the system compliant. If this non-compliance was a part of human error, the programmer will change the program and make it compliant providing easy debugging; else, the software will incur a segmentation fault when the shared object is changed without the required permissions, providing fail-stop enforcement.

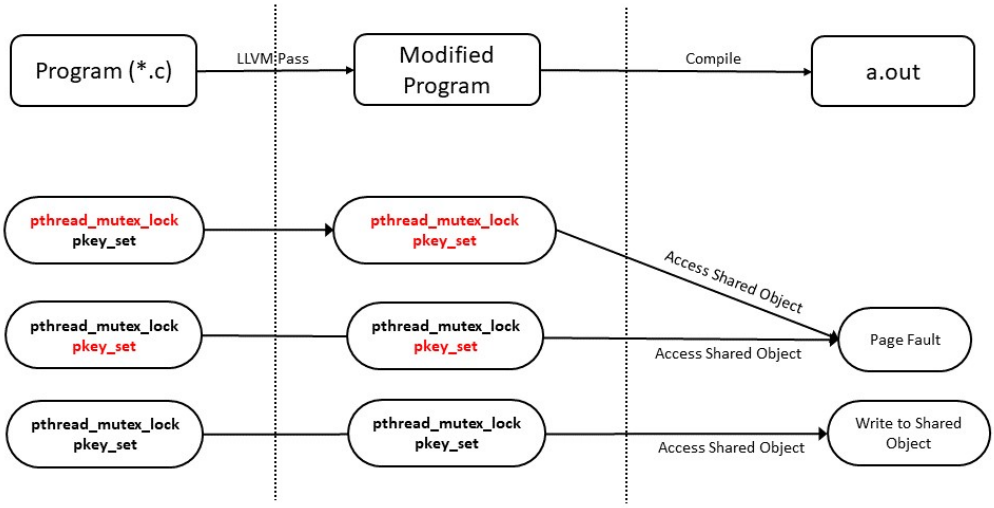


**Figure 8:** Design of our library for transactional systems. We ensure incorrect lock semantics result in a page fault at runtime.

**Algorithm 2** Wrapper: Critical Section

```
1: Acquire_Lock
2: PKRU_WRITE_ENABLE
3: Call critical_function(...)
4: PKRU_WRITE_DISABLE
5: Release_Lock
```

## 4 Evaluation and Limitations

Any system that caters to transactions must maintain their ACID properties. We demonstrate that we meet these requirements in our system.

**Atomicity** Consider two threads operating on two shared objects simultaneously. It is possible that updates to one of the shared objects is persisted but the transaction failed before completion. Our library focuses on providing consistency and isolation, hence relies on existing rollback mechanisms to maintain atomicity. Using our library does not hurt atomicity as the library guarantees that no transaction following all the critical section rules will fail due to the changes made by the library's compiler pass.

**Consistency** Our library ensures that no malicious or erroneous thread is able to mutate shared objects. This is ensured by using additional write protections around shared objects in addition to semaphores and mutexes. Any thread poised to make a shared object inconsistent, receives a segmentation fault, maintaining the objects consistent state.

**Isolation** The library ensures that at no point are two threads able to write to the same shared object simultaneously. This is done by the memory protection checks in LLVM pass. The pass will remove any instance of pkey protection updates which are not preceded with a lock. Hence, at runtime, such a thread will receive a segmentation fault. Therefore, isolation across different concurrent threads is maintained.

**Durability** This library relies on underlying hardware and software features like journaling to provide durability. While, the library does not add any durability guarantees to the transactional system, it does not harm existing durability guarantees.

Given the programmer correctly defines all shared objects, the given approach will prevent bugs in attaining proper mutual exclusion and simplify the endeavor of debugging the program if such semantic errors remain.

Our evaluation is based on the number of source lines to be changed by the programmer and the overheads incurred while executing the program.

```
1 int VanillaCode()
2 {
3     //PreText
4     int *A = malloc(size);
5     sem_wait(&mutex);
6     CriticalFunc(A);
7     sem_post(&mutex);
8     //PostText
9 }
```

```
1 int ModifiedCode()
2 {
3
4     //PreText
5     A *= LibAlloc(size);
6     LibCritical(*CriticalFunc, A);
7     //PostText
8
9 }
```

**Figure 9:** Change in source code needed by the programmer

Figure 9 shows the change in the source code. For small programs it should not be difficult to migrate to our architecture. For large programs, however, the changes may be more significant.

### 4.1 Limitations

There are a few limitations to our approach which arise from hardware and software constraints listed below.

1. Since, we tag and protect each shared object using pkeys, we are limited in the number of shared objects to the number of available hardware pkeys. Most hardware

that supports pkeys, only supports a total of four bits for referencing, i.e. a total of 16 pkeys. We can remove this assumption by using software pkeys such as the ones described in [2].

2. The crux of restricting any illegal write operations to shared objects is based on the removal of pkey operations by the user program. Therefore, our approach does not allow any other usage of pkeys by the user program.

3. Software most often use different shared libraries to offload functionalities in their core logic. Our approach requires all the shared libraries to be compliant by changing source code and recompiling using our LLVM pass.

## 5 Conclusion

We have shown a first order solution which mitigates unexpected behavior due to the absence of correct mutual exclusion guarantees in transactional systems. To this end, our solution provides fail-stop enforcement and easy debugging for such unexpected behavior. Ultimately, we hope this system will be used to protect large-scale transactional systems and enable developers to automatically locate semantic bugs in their programs.

We would like to thank Santosh Nagarakatte and Sudarsun Kannan for their inputs without which this work would not have be possible.

## References

[1] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[2] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.