`myThread`: User Transparent Multi-Processes Programming Library

Tianyi Liu
Computer Science Department
University of Texas at San Antonio
San Antonio, TX, USA
tianyi.liu@my.utsa.edu

Abstract

Mutithreaded programming is an efficient way to enhance performance of applications. However, multithreaded programs always have data race problem, which makes the result of parrallel programs be undeterminnistic. Undeterministic feature can be a fital weakness for precise critical applications and even induce disasters in some cases. To deal with this problem, this technique report present `myThread`, a run time library, which could convert threads into processes silently, and the whole process is transparent to programers.

When we use processes to implement threads, we should guarantee the processes can also share the memory regions that threads share in multithreaded programs. Also, we should reinitialize the mutex locks, conditon varialbles and barriers to make them shared among processes. The experiment result indicates that `myThread`can be comparable with `pthread` and in some cases can be even better.

Keywords Multithreaded Programs, Multi-Processes, pthread, Mmap, Thansparent

1 Introduction

Mutithreading is an effective way to enhance performance of applications. It tries to fully use of the resources in modern computer system. However, Multithreaded programs also have to face some problems, such as data race, underministic executing flow, and deadlock. There are two reasons for such phenomenons. The first reason is that multithreaded programs share global data and heap, and different threads can access others' stacks if they have the valid addresses. The second reason is that there is no guarantee of the executing order among different threads, and they may access the same memory locations with different orders.

Because of the underministic characteristic, the output of multi-threaded programs may vary if we run a program for several times, which is really a serious problem in precise sensitive computing taskes and finance or economy related affairs, and it may even induce disasters because of the undeterministic results.

OSCourse'17, Nov 01–22, 2017, San Antonio, TX, USA
2017.

There exit some systems to deal with the underministic problem. CoreDet [1] is a system to check every underministic operations, such as memory accesses, synchronizations. If the memory location or synchromization is owned by a thread itself, then the thread execute the command directly. Otherwise, it will wait and check again in next stage. CoreDet is really a useful design for avoiding undeterministic behaviors, but it is slow and unstable. Dthread [3] is another system designed to solve undeterministic problems of multithreads. In Dthread system, it just implement threads as processes. In the executing stage, all the child processes are independent from each other. And in the commit stage, all the processes do the synchronization work. The Dthread library is much faster than CoreDet and generally as faster or even faster as pthreads. Figure 1 discribes the differences between multithreaded programs and muti-processes.

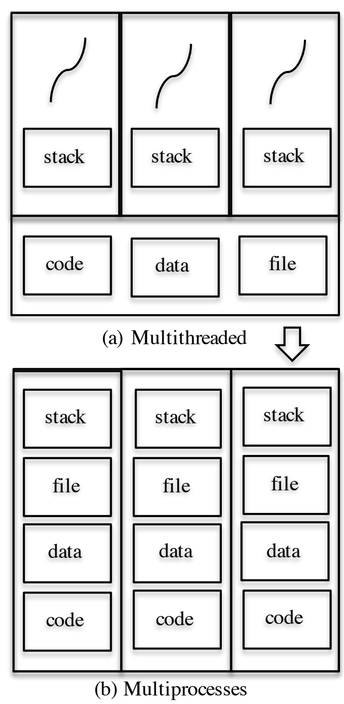

Figure 1. Change mutithreaded program into multiprocess.

In this work, we also implement threads as processes, which is transparent to programmers. We provide a library, named `libmythread`, to replace the `pthread`. Then programmers can use `-lmythread` flag when compiling their multi-threaded programs.

Outline:

The remainder of this report is organized as follows. Section 2 gives an overview of our work, including what features our library support and After that, Section **??** presents the detailed implementation, including the functions that we use when trying to support specific characteristics. Finally, Section 3 presents experimental results of `myThread`. Possible limitations and future extensions of `myThread` are discussed in Section 4.

2 Implementation

This section describes the design of `myThread`. First, we will have an overview of how `myThread` is implemented. Second, we will talk about the detail implementation from 3 aspects: How to intercept pthread_create to produce processes; How to share memory among the processes; How to implement synchronization.

2.1 Overview of Implementation

Figure 2 describes the overview of the whole process. The implementation has two steps. The first step is initialization, during which the library functions are wrapped and heap region, global region are also allocated in virtual address space.

To implement the goal of replacing threads with processes transparently to programmers, we should intercept all pthread related functions that appear in user's programs and then replace them with our own functions implemented in our library. The functions that we intercepted is shown in Table 1. The left column shows the original functions in pthread, and the right column shows the new functions that implemented in our `myThread` library.

As show in Figure 2, when the program calls a function in pthread library in run time, the funtions will be linked to the corresponding functions in our library. In this way, we can prevent pthread_create function from creating multiple threads, but creating multiple processes. Since multiprocess program should mimic the behavior of multithreaded programs, we should guarantee different processes sharing their heap, global, and dynamic library files. Another problem is to implement synchronization. Originally, the locks are shared among threads, but now we should initialize the locks to be shared among multiple processes. All the details of the design will be discussed in Section 2.2

2.2 Implementation Details

In this section, we will discuss the concrete technologies that we use to implement threads using processes.

Table 1. Comparasion of Library Functions

pthread functions	mythread functions
pthread_create()	thread_create()
pthead_join()	thread_join()
pthread_cancel()	thread_cancel()
pthread_mutex_init()	thread_mutex_init()
pthread_mutex_lock()	thread_mutex_lock()
pthread_mutex_unlock()	thread_mutex_unlock()
pthread_mutex_trylock()	thread_mutex_trylock()
pthread_mutex_destroy()	thread_mutex_destroy()
pthread_cond_init()	thread_cond_init()
pthread_cond_wait()	thread_cond_wait()
pthread_cond_signal()	thread_cond_signal()
pthread_cond_broadcast()	thread_cond_broadcast()
pthread_cond_destroy()	thread_cond_destroy()
pthread_barrier_init()	thread_barrier_init()
pthread_barrier_wait()	thread_barrier_wait()
pthread_barrier_destroy()	thread_barrier_destroy()

2.2.1 How to Intercept pthread_create to Create Processes

As mention in Section 2, our library intercepts the functions in pthread. So, when the programmers try to create threads, they will call the pthread_create function in their programs. In our implementation, we use a clone() system call to replace the pthread_create(). When we use the clone() function, we need to pass 4 parameters to it. The first parameter is a function pointer, which points to the function that the new process will execute. The second parameter is bottom address of a stack. We use the mmap() systemcall to allocate a large range for the stack, caculate the end address and then pass the address to the clone() function. For the third parameter, we set the child process to share I/O, file system and files with the parent process. The last parameter is the arguments that the child process need, and we just use the default arguments that programmers input when they call pthread_create function.

To ensure that the stacks of processes can be accessed by parent process or sibling processes, we use the MAP_SHARED flag when we calling the mmap() systemcall to allocate stack space. Another thing that we should mention is that we did not make the child processes to share the stack of parent process. We will leave it for future work.

2.2.2 How to Share Memory Mappings between Processes

Since we use processes to implement threads, we should make the replacement to be transparent to users. From Figure 1, we learned that all the memmory regions should be shared among the threads. That is to say, heap, global, library mappings should be shared, and there is only one copy in memory. The special case is stack. Because each thread has its own

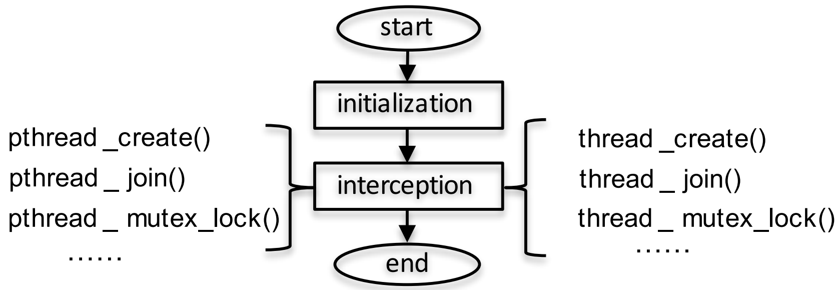

Figure 2. Implementation overview of myThread. The first step is initialization when the library is loaded into memory. The second step is interception at run time.

stack, and different threads can access not only their own stacks, but also stacks of others'. So, if we try to implement multithreaded programs using processes, there should be several stacks regions in memory, and all the stack reginos can be accessed by all the threads.

To implement the goals above, we tried to disable the copy-on-write characteristic of multiprocess programs. Our method is to use "cat /proc/pid/maps" command to get the memory mapping of a parent process. Before we call the clone() function, we just remaped all the memory regions to be shared except the code region. The detail process is described as Remmaping Algorithm in Algorithm 1.

Algorithm 1 Remmaping Algorithm

---

regions = get all the mapping regions of parent process except code and stack
for each *region* $\in$ *regions* **do**
  backup = maping a new region with same SIZE;
  copy the data in *region* and save it in *backup*;
  current = remap the *region* area with MAP_SHARED;
  copy the data from *backup* to *current*;
end for

Since we have already talked about how to deal with stacks in Section 2. After all the above steps, we have completed most part of the taskes about implementing threads using processes. The last thing that we should do is to support synchronization, which we will talk about in detail in Section 2.2.3.

2.2.3 How to do Synchronization

In multithreaded programs, we always use lock() to define a critical region to access memory variables in an atomical way. We also use locks and conditional variables together to avoid the possible deadlock proglems. So, in this section, we will discuss how to support synchronization among processes and make the multithreaded programs look like multiple processes.

We just intercept all the functions related to synchronization the same way that we deal with pthread_create. And then, we change the attributes of locks, conditional variables, and barriers to be shared among different processes. the method that we use is shown in Table 2.

Othrer than lock, conditional variable and barrier, we also need to implement pthread_join to synchronize each threads. So, we just use waitpid() to wait the progresses to expire. In order to implement pthread_cancel(), we just call kill() systemcall to stop a perticular process.

3 Evaluation

3.1 Experimental Setup

We choose a 16-core server to do all the experiments. The server has two sockets with 256GB of memory and Intel(R) Xeon(R) CPU E5-2640 processors. For each processor, it has 256KB L1, 2MB L2, and 20MB L3 shared cache. The version of operating system is Linux3.19.0 and all applications were compiled using gcc 4.9.2 with -O2 flag.

Evaluated Applications: myThread is evaluated on a subset of PARSEC [2]: blacksholes, dedup, pfscan and freqmine for performance. We also get the scalibility data by changing the thread numbers.

3.2 Performance

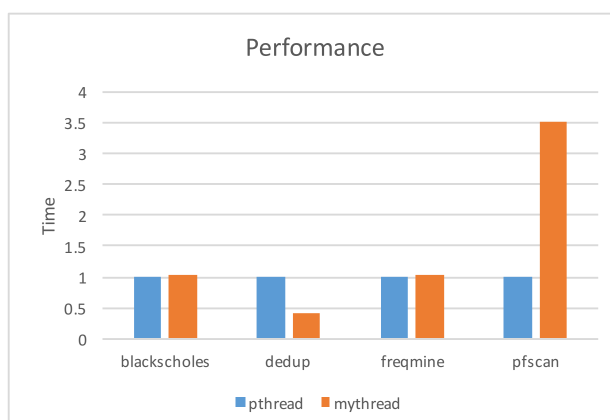

Figure 3. Performance comparasion of pthread and mythread on 4 different applications.

Table 2. Synchronization Implementation

Synchronization Funtion	Actual Implementation
pthread_mutex_init(&mutex, NULL)	thread_mutexattr_init(&mutex_attr) thread_mutexattr_setpshared(&mutex_attr, PTHREAD_PROCESS_SHARED) thread_mutex_init(&mutex, &mutex_attr)
pthread_cond_init(&cond, NULL)	thread_condattr_init(&cond_attr) thread_condattr_setpshared(&cond_attr, PTHREAD_PROCESS_SHARED) thread_cond_init(&cond, &cond_attr)
pthread_barier_init(&barier, NULL)	thread_barierattr_init(&barier_attr) thread_barierattr_setpshared(&barier_attr, PTHREAD_PROCESS_SHARED) thread_barier_init(&barier, &barier_attr)
pthead_join(tid)	waitpid((pid_t) tid)

From Figure 3, we can learn that `mythread` is comparable to `pthread` library. `mythread` performs better than `pthread` in `dedup`, but is worse in `pfscan`.

A possible reason is that isolation of memory regions among multiple threads reduces false sharing problems in dedup applications a lot. However, pfscan application may have little false sharing cases in cache.

Scalability To evaluate the scalability of `mythread`, we increase the thread number from 4 to 32, and we get the executing time of `mythread` vs. `pthread`. The performance scalability is show in Figure 4.

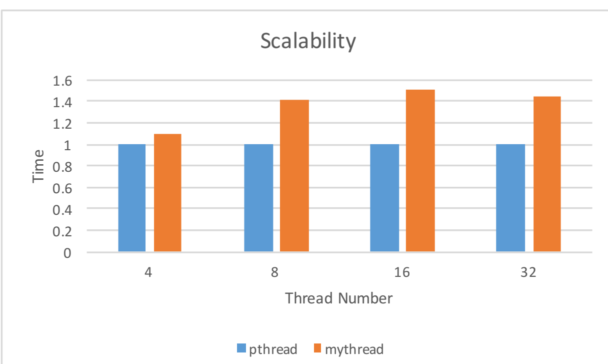

Figure 4. Performance scalability comparasion between `pthread` and `mythread` when increasing thread number from 4 to 32.

As show in Figure 4, we also use pthread as baseline. When changing the thread number from 4 to 32. The average relative executing time increase for the `mythread`. This is easy to under stand. If the thread number equals to process number, and they are both to be 1, the performance of process and thread should be the same. When we increase the thread and process number, the performance of multithreaded programs should be better at more cases. However, multiple processes can make our program more deterministic. In some precise critical applications, deterministic is even more important.

4 Limitation and Future Work

This section shows some limitations of `iReplayer` and possible extensions in the future.

Firstly, `myThread` only shares stack among child processes. Main process's stack is not shared with child processes, which means child processes can only read parent process's stack, but can not write. The semantic is not consistent with real multithreaded programs.

Secondly, `myThread` must have some potiential bugs to deal with, since it can only run some of the applications in PARSEC, not all of them. To the other PARSEC applications, there are still deadlock problems.

Thirdly, comparing to Dthread work, we only convert the threads into processes, which is transparent to progrmers. We did not implement commit stage designed in Dthread paper and our evaluation work is not sufficient.

5 Acknowledgment

This work was a course project released by Dr.Tonging Liu in his operating system class, 2017. In fact, this project is a simpler version of Dthread project, which was conducted Dr.Liu when he pursing his PhD in UMass Amherst. I would like to express my thanks to him for his patience and harsh training.

References

[1] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/1736020.1736029

[2] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

[3] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587