# rThread : A Lightweight Dual-level Thread Library

Yujie REN
*Department of Computer Science, Rutgers University*
*renyj1991@gmail.com*

## Abstract

Multi-threading programming model has become more important than before since modern general purpose processors have more cores than ever. In addition, SMT (Simultaneous Multithreading) further exploits the potential for parallel in a single core. `pthread` - POSIX standard thread library is a good existing interface of multi-threading in UNIX-like operating systems. However, `pthread` uses a 1:1 thread model, which means a thread created by `pthread` will mapped to a kernel thread. This thread model could take advantage of multi-core but failed to provide a flexibility for users to choose their own thread scheduling mechanism.

Thus, we present a dual-level thread library - rThread in this paper, which takes advantage of both kernel-level thread and user-level thread supporting 1:1, 1:N and M:N mapping. rThread works as follows: each kernel thread created by rThread could either running only one user threads like `pthread` do or running in a loop, grabs any runnable user-level threads in the ready queue to execute and quickly switching context between different user-level threads according to user specific scheduling algorithms. Evaluation on rThreads shows that rThread achieved equivalent performance compared with `pthread` and have advantage in context switching overhead over `pthread`.

## 1. Introduction

The emergence of multi-core processors motivates the demands of multi-threaded programming model, which means different processes could run in parallel. Moreover, SMT (Simultaneous Multithreading) technology enables the processor to issue instructions of different threads at the same time on one physical core. Simultaneous multithreading combines the multiple-issue-per-instruction features of modern superscalar processors with the latency-hiding ability of multithreaded architectures.[1] SMT is now widely used in main stream desktop and server processors such as Intel Nehalem and recently published AMD ZEN micro-architecture.

To fully utilize the multi-core and SMT computing resource, many multi-threading programming interfaces have been developed such as POSIX thread and `NPTL`. They uses 1:1 thread model - a new created thread is mapped to a kernel-level thread [2][3]. And a kernel-level thread is a schedule-able entity in Linux kernel. This model takes the advantage of SMT; When the number of threads created by `pthread` or `NPTL` does not exceed the maximum SMT number of the processor, say hyper-threading number in Nehalem micro-architecture, all of them are likely to be executed in parallel.

The advantages of 1:1 thread model is obvious. First, it could maximize the usage of multi-core and SMT system. Second, if one thread of `pthread` or `NPTL` is blocked by I/O, other's are not affected. However, this model still have some drawbacks in some aspects; First, the scheduling is totally in kernel mode, users have no idea how they're scheduled; Second, scheduling overhead is rather high because every context switch between different kernel-level threads, jumping backforth between kernel mode and user mode is required [4]. What about the 1:N model (many user threads map to one kernel thread)? The advantages are just the disadvantages of 1:1 model. Low schedule overhead but limited usage of multi-core and SMT. Moreover, if one thread is blocked by I/O, others mapped to the same kernel thread are also blocked.

rThread combines the advantages of both kernel-level threads and user-level threads adopting a hybrid and flexible mapping relationship between user-level threads and kernel-level threads according to users specification. In addition, rThread is pretty lightweight and covers basic functionalities of a thread library.

The second section will cover the overview and design philosophy of rThread library. The third section will present architecture of rThread including design of user-level threads and synchronization design. The forth section will discuss study and evaluation on rThread performance. The fifth one lists some limitations and future works.

## 2. rThread Overview

Our work on rThread includes the basic operations of thread library: *init, create, destroy, yield, exit, schedule, mutex and condition variable*. In addition, we also studied performance of rThread using different benchmarks and compare it to `pthread`.

**Kernel-level thread and User-level thread:** In Linux kernel, both process and thread are schedule-able entities, and they're all encapsulated in a data structure called *task*. The main difference between a process and a thread is that a newly created process have a separate virtual address space to its father process while a thread belongs to a process share the virtual address space. That is to say, communication between threads with the same *tgid* has less overhead than that between processes. To create a kernel-level thread, *clone()* system call need to be invoked. To use *clone()*, we need to assign a chunck of memory space, set *SIGCHLD*, *CLONE_SIGHAND*, *CLONE_VM* and *CLONE_PTRACE* flags, and pass a function pointer pointing to which function we need to execute in the newly created kernel thread.

User-level threads are totally running in user space and invisible to linux kernel. Thus, we need to design TCB (Thread Contrl Block) for user-level threads which contains thread id, thread status, user-space executing context and stack pointer. The Unix standard provides one more set of functions to control the execution path. These functions were part of the original *System V* API and by this route were added to the Unix API. Linux also adopted this. Each user-level thread has its own context data structure (*ucontext_t*). And a bunch of functions: *makecontext()*, *getcontext()*, *setcontext()* and *swapcontext()*.

**Executing Mode:** Compared with POSIX thread and `NPTL`, rThread support flexible mapping relationship between kernel-level threads and user-level threads. For 1:1 mapping model, rThread creates a kernel thread that executes the function we need to execute just like what `pthread` does. For 1:N mapping model, rThread creates a kernel thread that executes the user-level thread scheduler and N user-level threads; The scheduler could do fast context switching between user-level threads according to what scheduling algorithm the user takes. For M:N mapping model, user-level threads are thrown into a user-level thread queue, and each kernel-level thread runs in a loop, grabs any run-able user-level threads in the user-level thread queue.

**User-level thread scheduling:** Users could choose their own scheduling algorithms of user-level threads. If each task has the same priority, round-robin [5] could be used; Users could also select time slice by themselves. In order to leverage the response time for both long-time jobs and short-time jobs, multi-level feedback queue could be used according to job types. Currently, this two scheduling algorithms are available in rThread library. In the future work, other scheduling algorithms may be added.

**Thread Synchronization:** We only introduce basic synchronization mechanism between user-level threads including *mutex lock* and *condition variable*. As we know, we could implement condition variable by using Linux native semaphore [6]; But we'll introduce a lightweight implementation in next section. Mutex lock is a blocking lock when a thread fails to grab the mutex, it goes to sleep until the thread holding the mutex releases it. Condition Variable is very similar; it need the help of mutex. When a thread is waiting on a condition variable, it release the mutex, and rehold the mutex again when being waken up.

In addition, for a N:M thread model, the user-level thread ready queue is a critical section. We don't want to see that two kernel-level threads running the same user-level thread, it's not only wasteful but also likey yielding non-determinism and wrong output. Thus, it's important to force serialization on accessing user-level thread ready queue.

## 3. rThread Architecture and Detail

This section describes rThread design details of important data structures and algorithms and implementation.

### 3.1 User-level Thread Design

User-level Threads are totally running in user space which means during its lifecycle it's user that is responsible for its creation, synchronization, scheduling and destroy. In this section, we will discuss on how we design user-level threads and several vital data structures along with user-level thread.

#### 3.1.1 TCB and Queue ADT

As we know, TCB (Thread Control Block) is an important data structure for a thread which keeps the status, identification and executing control information of a Thread. The following struct is the TCB for rThread user-level threads.

```
typedef struct threadControlBlock {
    rthread_t   tid;     /* Thread ID     */
    threadStatus status; /* Thread Status */
    ucontext_t context;  /* Thread Contex */
    void *stack;         /* Stack pointer */
    ...                  ...
} _tcb;
```

Thread id is the unique identifier for each user-level thread; it's self-increasing when a user-level thread is created while thread id of a kernel thread is given by Linux kernel. The *tid* of a kernel thread is quite important when doing context switching and we will cover this later. Thread status includes *created*, *running*, *ready* and *blocked*. Figure 1 shows an automaton of state diagrams of user-level threads. As we described in section 2, *ucontext_t* is used to keep track of current executing context of the corresponding user-level thread.

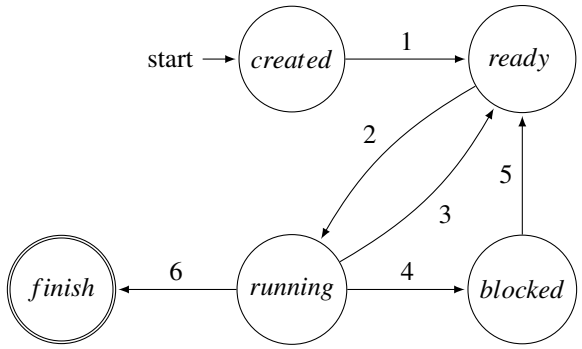


Figure 1: State diagram of rThread user-level thread

Figure 1 shows state diagram of user-level thread. 1) A thread is created and being put into ready queue. 2) A thread is chosen by the scheduler and is going to execute. 3) A running thread is yielding or using up its time slice. 4) A thread failed to grab mutex or waiting for a condition variable or waiting on I/O. 5) A thread is waken up and ready to run. 6) A thread is finished.

Queue is the basic data structure for user-level threads. Ready queue contains threads ready to executing and block queue contains threads are blocked by a certain mutex lock or condition variable; In addition, we may also use queues of different data types. Thus, it's preferrable to make it an abstract data type which resembles the Template Class in C++. By defining each element in the queue be a void pointer, we could explicit type casting when handling different types of queue. For a thread queue, the base data type is a pointer to its *ucontext_t*.

### 3.1.2 User-level Thread Scheduling Algorithm

The key-advantage of user-level thread over kernel-level thread is that users could choose their own scheduling algorithm for user-level threads. Research on task scheduling algorithms have achieved a lot progresses. Here we will introduce four well-known scheduling algorithms for user-level threads. In our implement, we only finished Round-Robin and Multilevel Feedback Queue.

**Round-Robin:** Round-Robin, considered as the most widely adopted CPU scheduling algorithm, is known for its equalty - each thread have the same time quantum. However, if time quantum chosen is too large, the response time of the processes is considered too high. On the other hand, if this quantum is too small, it increases the overhead of the CPU. Thus, several improvements have already been made on improve the performance on Round-Robin scheduling algorithm [7][8].

**Multilevel Feedback Queue:** Compared to Round-Robin, Multilevel Feedback Queue allows tasks moving between queues. This movement is facilitated by the characteristic of the CPU burst of the process. Multilevel Feedback Queue favors not only short jobs but also I/O bound jobs.

**Borrowed-Virtual-Time:** Borrowed-Virtual-Time (BVT) Scheduling provides low-latency for real-time and inter active applications yet weighted sharing of the CPU across applications according to system policy, even with thread failure at the real-time level, all with a low-overhead implementation on multiprocessors as well as uniprocessors. It makes minimal demands on application developers, and can be used with a reservation or admission control module for hard real-time applications [9].

**Lottery Scheduling:** Lottery scheduling provides efficient, responsive control over the relative execution rates of computations. Such control is beyond the capabilities of conventional schedulers, and is desirable in systems that service requests of varying importance, such as databases, media-based applications, and networks [10].

To implement scheduling algorithm, a set of Linux runtime library function is needed including *sigaction*, *setitimer*.

### 3.1.3 Mutex Lock and Condition Variable

This section gives detailed description on rThread synchronization mechanism - *mutex lock*, *condition variable* design. In addition, we also give a possible solution to handle priority inversion problem.

**Mutex Lock:** Mutex is a mechanism that allows only one process or thread hold the lock, others will wait until the current owner release the lock. Figure 2 describes the finite state machine for how mutex lock works.

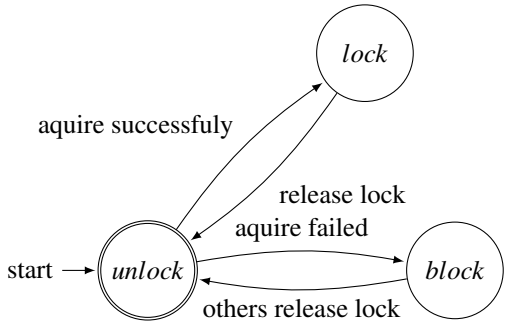


Figure 2: Finite State Machine for rThread mutex lock

As Figure 2 shows, when trying to lock the mutex, only one thread will success and others goes to *block* state. When the current mutex holder releases the mutex, the first thread in the block queue are waken up and being put into running queue. In order to ensure only

one thread hold the lock, an hardware-supported atomic primitive *__sync_lock_test_and_set()* is needed. The reason using blocking lock in rThread mutex is that frequent *test-and-set* operation is a big waste for CPU resource.

```
typedef struct rthread_mutex_t {
    _tcb *owner;      /* owner of mutex */
    uint lock;        /* lock */
    Queue wait_list;  /* block queue */
} rthread_mutex_t;
```

The code above is the data structure for rThread mutex lock. One significant different between mutex and unary semaphore is that mutex must have a owner and only the owner could release the lock. rThread Library currently provides four functions of mutex: *init*, *lock*, *unlock*, *destroy*.

**Priority Inversion:** Priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks. Considering in a preemptive real-time operating system, there're three tasks: $A$, $B$ and $C$. The priority sequence of the three is $A > B > C$; Both of $A$ and $C$ need to access a critical section $M$. Now, $C$ comes first, get the mutex and entering $M$; Then comes $A$, obviously, $A$ will be blocked for failing to grab mutex. Finally, $B$ comes, preempt $C$ for running. Now, $B$ is running while $A$ will be blocked until $C$ release the lock. The problem is $C$ is also blocked. Thus, $A$ is very likely that it will wait for quite a while, which is absolutely dangeous in a real-time system. Well-know RTOS *VxWorks* in NASA Pathfinder explorer used to undergo a priority inversion issue [11].

rThread also supports real-time task scenario. Since rThread is responsible to scheduling user-level threads, there's no way to disable interrupt in rThread. Thus, to prevent priority inversion problem, we use priority inheritance - whenever a high priority task has to wait for some resource shared with an executing low priority task, the low priority task is temporarily assigned the priority of the highest waiting priority task for the duration of its own use of the shared resource, thus keeping medium priority tasks from pre-empting the (originally) low priority task, and thereby affecting the waiting high priority task as well. Once the resource is released, the low priority task continues at its original priority level [12].

**Condition Variable:** Condition Variable is another synchronization mechanism using *wait*/*signal* scheme. Threads waiting on a condition variable go to sleep until this condition variable is available. The condition variable is a little bit like mutex lock but mostly, they are different. If several tasks are all waiting for a certain condition, they are put into the waiting queue. When this condition variable is now signaled, only one task waiting on the queue is going to resume. And we could also do broadcast to let everybody in the waiting queue to resume in order.

Typically, condition variable and mutex lock are used together. When a mutex lock is acquired by a task and need to wait for a certain condition, then the mutex lock was released and that task will go to sleep. When another task signal that task, the mutex lock will be held again.

#### 3.1.4 Other Implementation Detail

So far, we've discussed the core structure of rThread user-level threads. In practice, to be a robust programming interface, there're a lot of issues we also need to take into consideration.

**Memory Leak:** Memory Leak has been a notorious issue in C/C++ programs since they're born because C/C++ programmers need to manage dynamic-allocated memory chuncks by themselves. This seems to be an inevitable issue in a programming language level for C/C++ gives programmers rights to manipulate memory by themselves.

In rThread implementation, when creating either a user-level thread or a kernel-level thread, we need to use *malloc()* to get a chunck of memory in heap as our activation record. However, when the function finishes, it cannot call *free()* automatically. In addition, it's not user-friendly and even impossible for programmers using rThread Library to release the activation record by themselves. Thus, we add a wrapper function in rThread Library to manage the activation record of user-level thread. In the wrapper function, user-level thread executing function is invoked and thread status is set to be *running*. When it finishes, we still put the context of wrapper function into ready queue but set it status to *finish*. The scheduler will call *free()* for any user-level threads whose status is finished.

**User-level Thread Queue:** As we described in section 3.1.1, we use an abstract data type in queue. For a user-level thread queue, no matter a run queue or a wait queue, we use the pointer to *ucontext_t* as the element. When accessing other fields in TCB, we also provide a *GET_TCB()* macro to find the container of a member.

**Deadlock and Livelock:** Deadlock and Livelock are two classic issue in designing thread concurrency. For user-level threads in rThread library, deadlock and livelock problems should not exist because before entering the critical section, threads are forced to use atomic operations to check availability. Thus, hold and wait condition does not meet. For possibly deadlock issues in swapping between *ucontext_t*, we've not sure for it depends on how *ucontext_t* is implemented. So far, in our benchmark and testing, no deadlock issues exists.

### 3.2 Interaction with Kernel-level thread

rThread provides a flexible interface on user-level thread and kernel-level thread mapping. For a 1:N or M:N mapping, kernel-level thread runs user-specific scheduler, grabs user-level threads in ready queue to execute. When there's multiple kernel-level threads, we need to be really cautious on handling context-switch between user-level threads and the scheduler function that inside a kernel-level thread. This section will describe how we handle this problem by using a global hash table.

#### 3.2.1 M:N Mapping Running Protocol

For a M:N mapping scheme, we have M kernel-level threads and N user-level threads (Typically, N>M). User-level threads forms a run queue at the beginning, each kernel-level thread grabs user-level threads to execute. To ensure kernel-level threads access run queue in a correct manner, we use a spin lock here by using *__sync_lock_test_and_set()* because accessing run queue (dequeue and enqueue) are quite short operations; Blocking locks will bring extra overhead and performance issue here [13]. Figure 3 shows how the protocol works.

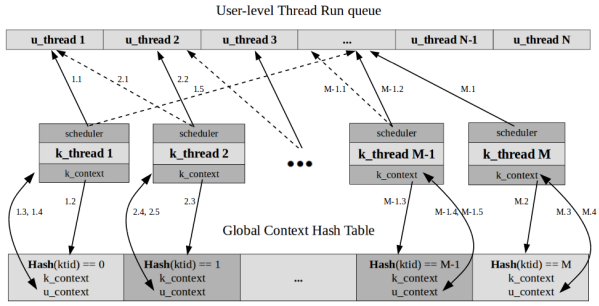


Figure 3: M:N mapping running protocol

When the time slice of a user-level thread is running out, it will be put back in the run queue based on scheduling algorithm. That is to say, during the lifecycle of a user-level thread, it shall be executed in one kernel-level thread or several different kernel-level thread. Now, the problem comes. When a kernel-level thread is switching context to a user-level thread, how could we ensure we will be in exact the same kernel-level thread after the user-level thread is running out of time slice? In next section, we'll introduce our global context hash table.

In Figure 3, *1.1* means the first step of kernel-level thread "1". For kernel-level thread "2" in the figure, (2.1) it first tries to grab user-level thread "1", however it fails because kernel-level thread "1" has already grubs user-level thread "1" even they're almost arrive at the same time (This is why we need hardware atomic operation support here). (2.2) Then it grabs user-level thread "2" successfully; (2,3) And bookkeeping the context of current user-level thread "2" inside its global hash table entry. (2.4) Next it should swap its current context to user-level thread "2". (2.5) When the time quantum of user-level thread "2" is up, the scheduler should swap back to the context of kernel-level thread "2".

#### 3.2.2 Global Context Hash Table

Each kernel-level thread runs the same logic, but not the same user-level threads. When switching context between kernel-level threads and user-level threads (the context switching here is still in user space), we must bookkeep both the user-level thread context and kernel-level thread context.

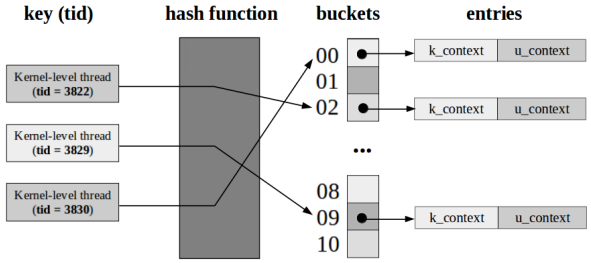


Figure 4: Global Context Hash Table

As it shown in Figure 4 above, we use a global hash table bookkeeping the key-value pair of kernel-level thread context and its current user-level thread context. The unique identifier of a kernel-level thread is its thread id given by the kernel, which could be retrieved by using *syscall(SYS_gettid)*. Thus, by using *tid* of a kernel-level thread as the key, we could build a global hash table on context mapping between kernel-level threads and their current executing user-level threads. In our current implementation, the hash function here is pretty simple by using (*tid* mod *M*). However, when the number of kernel-level threads becomes very large, it's preferrable to choose a more effective hash function to avoid entry collision.

## 4. Performance Study

In this section, we'll do both vertical and horizontal performance comparison and study on rThread. Different hardware configurations, different kernel-level thread configurations and comparison with `pthread`.

### 4.1 Performance on different hardware configuration

rThread takes advantage of both kernel-level thread and user-level thread. The maximum number of kernel-level

threads parallelism support in the hardware specifies the upper bound of the thread-level parallelism. We use different hardware configuration to study the performance of rThread library. We use three hardware configurations: 1) Intel *Core i7-6700* Quad-Core with Hyper-Threading CPU system with 16GB RAM. 2) Intel *Xeon E5-1650 v4* Hex-Core with Hyper-Threading CPU system with 32GB reg-ECC RAM. 3) Intel *Core i5-6200U* Dual-Core with Hyper-Threading CPU system with 8GB RAM.

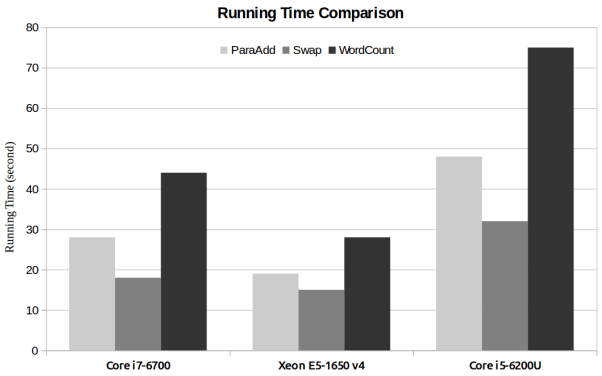


Figure 5: Running time on three hardware configurations

Figure.5 above shows the running time comparison in the three hardware comparison listed above. We also use three different benchmark programs: *ParaAdd*, *Swap* and *WordCount*. All of them uses 1:1 mapping model. The evaluation result indicates that for the same task, more cores typically yields to shorter executing time; More kernel-level threads could run in parallel. However, it's not "the more kernel-level threads the better".

It's also interesting to see that *Xeon E5-1640 v4* achieved better performance on *Swap* than *Core i7-6700* though the single core frequency of the two processors is nearly identical and *Swap* support only 4 way thread-level parallelism. Because *E5-1640 v4* has 4 memory channels while *i7-6700* only have 2 memory channels. Thus *Xeon E5* is likey to have less memory bus competition.

### 4.2 Performance on different thread number configuration

It's not always true that "the more kernel-level threads the better". As I stated in the previous subsection, the maximum number of kernel-level threads parallelism support in the hardware specifies the upper bound of the thread-level parallelism. For the first hardware configuration: Intel *Core i7-6700* Quad-Core with Hyper-Threading CPU system with 16GB RAM; The maximum parallel number of kernel-threads is 8; For the second configuration - Intel *Xeon E5-1640 v4*, the maximum parallel number of kernel-level thread is 12; And for the last configuration, the maximum parallel number of kernel-level thread is only 4.

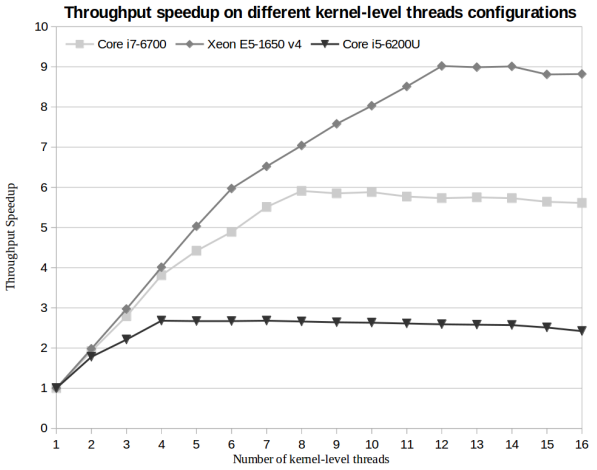


Figure 6: Throughput speedup on different kernel-level threads configurations

Figure.6 shows the throughput speedup with different kernel threads number configuration with the same task. For *Core i7-6700*: When kernel-thread number increase from 1 to 4, the throughput increasing is nearly linear; Each core will be assigned to a kernel-level thread and running in parallel. When kernel-thread number increase from 5 to 8, the throughput increasing is also linear but the increasing speed decreases. As we know, Hyper-Threading is an implementation of SMT by Intel; Each physical core has two seperate register files and decoders but they share the same per-core cache and memory bus. Thus, even if two hyper-threads could run in parallel, they still have problems like cache invalidation and memory bus competition. When kernel-thread number is greater than 8, the throughput turns to decrease slightly; Because this configuration exceeds the maximum parallel number of kernel-thread, and Linux scheduler need to schedule them. Thus, context switching overhead between kernel-level threads slightly harms throughput and in addition, cache invalidation and memory bus competition problems also exists. The above analyses also apply in *Xeon E5-1640 v4* and *Core i5-6200U*. *Xeon E5-1640 v4* has 6 cores and each core supports 2 hyper-threads; *Core i5-6200U* has 2 cores and each core supports 2 hyper-threads.

### 4.3 Compare with *pthread*

As we know, `pthread` uses 1:1 thread mapping model. It's interesting to compare the performance of `pthread`

with rThread in different workloads. Kernel-level thread has advantage over user-level thread when the tasks need high requirement of concurrency and the tasks also have blocked I/O operations. While user-level thread has advantage over kernel-level thread when CPU-bound tasks are interdependent and context switch happens very often with little I/O operations.

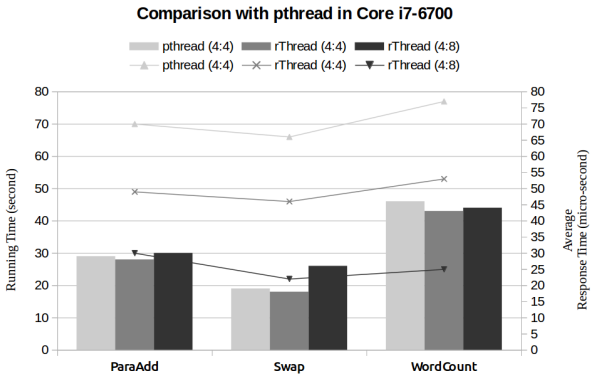


Figure 7: Throughput speedup on different kernel-level threads configurations

Figure.7 show the throughput comparison with `pthread` library on three different benchmarks. *ParaAdd* and *WordCount* are CPU-bound tasks while *Swap* is I/O-bound jobs. For CPU-bound jobs, rThread achieved equivalent performance as `pthread` having the same kernel-level thread numbers. In addition, rThread could achieved significantly lower response time when having more user-level threads without causing performance loss because context switch overhead between different user-level threads is mush lower than that of kernel-level threads.

## 5. Discussion

rThread could be fit into workloads prefer either user-level thread or kernel-level thread. In section 4, we've already analyzed its performance under different benchmarks. In addition, lightweight attribute helps rThread achieve good performance. However, there're still some tradeoff in designing a thread library. In this section, we'll discuss on limitations and what future work could be improved in rThread.

### 5.1 Limitation

We sorted up three existing limitations in rThread Library current implementation.

**Lack of Other OS support:** Currently, rThread is implemented based on Linux; It's lack of other OS support such as BSD family and Windows. As we know, in *Win32* API, kernel-thread implementation is very different than that in UNIX-like OS. rThread successfully provided a potable thread library but it may be difficult to implement M:N mapping in Windows.

**Non-intact Functionality:** Lightweight means a good thing: short binary files and cost less in runtime. However, in other aspect, lightweight might also means have less functionality than a bulk full-functioned implementation. We could take a grasp from two well-known desktop environment *GNOME* and *Xfce*.

**Mismatch between number of user-level threads and kernel-level threads:** When there are mismatches between the number of kernel-level threads and user-level threads, which means M<N. The current handling is very simple, just let idle kernel-level threads spinning without doing tasks.

### 5.2 Future Works

We've already built the basic architecture of rThread and done some performance evaluation. However, there're still some work need to be done based on the limitations of rThread as we listed in the previous subsection.

**Improvement on functionality intactness:** We only implemented a basic, important and small subset of `pthread` functionality. There're still some system-level issues need to be handled and implemented. In addition, portable issue also exists; To make rThread portable, we also need to add an abstract level handling different system call and native kernel-level thread implementation on different OSes.

**Performance study on different scheduling algorithm:** We currently only implemented two scheduling algorithms: *round-robin* and *multi-level feedback queue*. As we illustrated, user-specified scheduling algorithm for user-level thread is a key feature for rThread library; It's quite meaningful and important to learn the performance on different scheduling algorithms on different workloads.

**Dynamic-tuning on dual-level thread number mismatch:** Currently, the number of user-level threads and kernel-level threads are specified before compilation. During some time of some certain workloads, there should be some idle kernel-level threads in our library just spinning and doing nothing, which means a great waste for CPU resource and energy-inefficieny. It's also meaningful to improve our framework with a dynamic-tuning mechanism that could block some idle kernel-level threads.

## 6. Conclusion

rThread is a lightweight thread library implementation providing flexible mapping relation between user-

level threads and kernel-level threads. rThread achieved at least equivalent performance within its functionality scope as `pthread` and `NPTL`. Flexible mapping relation is the key point in rThread design philosophy, for users could adjust their own scheduling algorithms on different kinds of workloads.

The performance study shows that rThread could achieve equivalent performance compared with `pthread` and even better performance than `pthread` in some scenario for context switching between user-level threads is faster. Although rThread currently has several limitations, it will still serves in many scenario in practice. Kernel-level threads dynamic-tuning will be a good topic for our future research on improving the performance and energy efficiency.

## 7. Acknowledgements

rThread library is based on class assignment in Operating System Theory class in Computer Science Department of Rutgers University. The author really appreciates the help and guide on both the assignment and lecturing from professor Thu.D Ngyuen. After completing the class, the author finished and extended the class project to rThread library with this paper. Thanks to two amazing books: *Understanding the Linux Kernel - third edition* and *Advanced Programming in the UNIX Environment* which helps the author get better understanding on Linux thread details on synchronization, scheduling and implementation. Last but not least, the author appreciates Professor Uli Kremer and Professor Eddy Zheng Zhang for reviewing this paper.

## References

[1] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy *Simultaneous multithreading: maximizing on-chip parallelism*. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, Pages 392-403, ACM New York, NY, USA, 1995

[2] Felix Garcia and Javier Fernandez *POSIX thread libraries*. In *Linux Journal*, Volume 2000 Issue 70es, Article No. 36, Belltown Media Houston, TX, USA, Feb. 2000

[3] Ulrich Drepper and Ingo Molnar *The Native POSIX Thread Library for Linux*. In *Red Hat White Papers*, Red Hat Inc, USA, 2003

[4] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc and Evangelos P. Markatos *First-class user-level threads*. In *SOSP '91 Proceedings of the thirteenth ACM symposium on Operating systems principles*, Pages 110-121, Pacific Grove, CA, USA, 1991

[5] M. Shreedhar and G. Varghese *Efficient fair queuing using deficit round-robin*. In *IEEE/ACM Transactions on Networking Volume: 4, Issue: 3*, Pages 375 - 385, Jun 1996

[6] Andrew D. Birrell *Implementing Condition Variables with Semaphores*. In *Computer Systems - Theory, Technology, and Applications ISBN: 978-0-387-20170-2*, Pages 29-37, 2004

[7] Rami J. Matarneh *Self-Adjustment Time Quantum in Round Robin Algorithm Depending on Burst Time of the Now Running Processes*. In *American Journal of Applied Sciences 6 (10): 1831-1837, 2009 ISSN 1546-9239*

[8] Abbas Noon, Ali Kalakech, and Seifedine Kadry *A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average*. In *IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 3, No. 1*, Pages 224-229, 2011

[9] Kenneth J. Duda, and David R. Cheriton *Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*. In *SOSP '99 Proceedings of the seventeenth ACM symposium on Operating systems principles*, Pages 261-276, Charleston, CA, USA, 1999

[10] Carl A. Waldspurger, and William E. Weihl *Lottery scheduling: flexible proportional-share resource management*. In *OSDI '94 Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation Article No. 1*, Monterey, CA, USA, 1994

[11] Risat Mahmud Pathan *Report for the Seminar Series on Software Failures - Mars Pathfinder: Priority Inversion Problem.*

[12] Bjorn B. Brandenburg, and Andrea Bastoni *The case for migratory priority inheritance in linux: Bounded priority inversions on multiprocessors*. In *RTLWS 12*, 2012

[13] John Turek, Dennis Shasha, and Sundeep Prakash *Locking without blocking: making lock based concurrent data structure algorithms nonblocking*. In *PODS '92 Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, Pages 212-222, San Diego, CA, USA, 1992