Implementing Preemptive User Threads

Michael Xu
Information Networking Institute
Carnegie Mellon University

Rui Huang
Information Networking Institute
Carnegie Mellon University

Yueqi Liao
Information Networking Institute
Carnegie Mellon University

1 Background

User-level threads run on top of kernel-scheduled threads. The most common approach to implement user-level threading is one-to-one mapping, as seen in Pthread. This method delegates the tasks of scheduling and synchronization of user-level threads to the kernel scheduler.

Alternatively, many-to-one mapping involves a runtime system that creates a certain number of kernel-scheduled threads and periodically swaps user-level threads onto these kernel-scheduled threads. Since the kernel is unaware of the existence of the user-level threads, it is the runtime system that has to schedule and synchronize the user-level threads, and to achieve user-level scheduling, there are, again, two main schemes - cooperative versus preemptive.

The impetus for implementing user-level scheduling lies in its operational efficiency. This form of scheduling is executed within the user space, thereby minimizing kernel intervention and consequently diminishing the substantial overhead typically associated with kernel processes. This reduction in overhead is multifaceted. It encompasses the elimination of system call overheads, which are inherent when transitioning between user and kernel spaces — a transition that necessitates a system call, resulting in CPU mode switches, mandatory security verifications, and possible TLB (Translation Lookaside Buffer) flushes. Additionally, it mitigates kernel lock contention overheads. In a kernel-managed environment, lock contention can impose a significant toll due to its overarching system influence, the systemic overhead stemming from system calls during lock acquisition and release, and the inherent risk of engendering priority inversion scenarios. By circumventing these kernel space complexities, user-level scheduling enhances overall system performance. An additional compelling rationale for user-level scheduling is its ability to mitigate thread starvation, a condition where certain threads may receive insufficient processing time. Let's consider a practical illustration involving a multicore system with three cores, each initially running a single pthread dedicated to distinct applications: Spotify for music streaming, Chrome for web browsing, and a Linux system monitoring tool. Under normal circumstances, these threads would enjoy the full computational power of their respective cores. However, if we introduce a computationally intensive application that generates a large number of pthreads — let's say 100,000 — the dynamics change dramatically (shown in Figure 1). Assuming a simplistic round-robin scheduling approach is employed at the kernel level, where each pthread receives an approximately equal slice of CPU time, the original three pthreads would now face a significant reduction in available processing time. In such a round-robin scheme, the time allocated to each thread becomes a small fraction of what it was before because the CPU time is divided equally among a vastly increased number of threads. This means that

the pthreads for Spotify, Chrome, and the system monitor would have to compete with approximately 33,333 other pthreads (assuming an equal distribution across the three cores) for CPU cycles. As a result, these applications could become unresponsive or sluggish, leading to a suboptimal user experience. User-level threading offers a solution to the problem of thread starvation by introducing a layer of abstraction between the kernel and user space. In this model, multiple user-level threads, which are managed entirely in user space, are multiplexed onto a smaller number of POSIX threads (pthreads), which are managed by the kernel. The kernel schedules pthreads without awareness of the user-level threads. This means that each pthread receives a slice of the CPU's time as determined by the kernel's scheduler. Since the kernel is unaware of the user-level threads, it continues to allocate CPU time to each pthread as if it were a single thread of execution. The responsibility of scheduling the user-level threads mapped to each pthread is handled in user space. In user space, a library or runtime system takes charge of the user-level threads. This library can implement a variety of scheduling algorithms tailored to the application's needs, independent of the kernel's scheduling policies. For example, it can prioritize certain threads over others or allocate CPU time based on thread behavior or workload characteristics. Because user-level threads are lightweight, creating and managing them incurs less overhead than managing pthreads. When a user-level thread performs a blocking operation or otherwise yields the processor, the threading library can schedule another user-level thread onto the same pthread without involving the kernel. This means that the CPU quota assigned to the pthread is effectively shared among all the user-level threads it contains, allowing for more granular and efficient use of CPU time.

User-level scheduling is inherently cooperative due to the absence of direct access to hardware timing resources that can interrupt the execution flow asynchronously and hand control to some scheduler routine, preventing any user thread from taking up the CPU resource "indefinitely" until the underlying

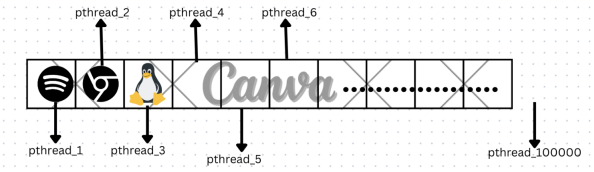

Figure 1: Thread-starving example

kernel-scheduled thread that backs it gets descheduled by the OS. Thus, kernel intervention is inevitable to achieving preemption on the user land. Signal-yield model has been widely adopted to implement user-level preemption since Unix signals have the favorable property of being asynchronous. More over, it gives a way to indirectly use the timing hardware from the user land via having the kernel relay the hardware event of timer expiring to the corresponding process by the delivery a signal. Since Linux checks pending signals that have been delivered to a process every time before returning from a trap to that process's user space and that each timer tick invokes a trap, the timer signals can be handled rather promptly (almost nearly as soon as the OS timer expires) and whereby provides us with a reasonably precise timing for implementing preemptive scheduling.

The concept of preemptive scheduling, implemented using POSIX signals, has been thoroughly studied and incorporated into programming languages like Go. Go is known for its lightweight threads - Goroutines. In newer releases, starting from version 1.14, Go has transitioned from pure cooperative scheduling to incorporating preemptive scheduling.

The reason for this transition is that the previous approach, where Goroutines voluntarily yielded at well-defined "safe points," had been particularly problematic when the control flow didn't touch any of these points, causing severe starvation. A simple example that causes Go scheduler prior to v1.14 to fail rather disgracefully is demonstrated in figure 2.

The code snippet has the main thread create as many Goroutines as there are cores. Each Goroutine simply increments x by 1 in an infinite loop, and then the main thread sleeps for 1 second. After this pause, the main

```
func main() {
    var x int
    n := runtime.GOMAXPROCS(0)
    for i := 0; i < n; i++ {
        go func() {
            for {
                x++
            }
        }()
    }
    time.Sleep(time.Second)
    fmt.Println("x =", x)
}
```

Figure 2: Scheduler Failure Demo

thread wakes up and prints the current value of x.

Unfortunately, running such a straightforward program on Go versions prior to 1.14 would result in it getting stuck indefinitely. The reason is simple: Go versions before 1.14 used purely cooperative scheduling. In cooperative scheduling, a Goroutine yields control to others upon function calls. However, in this program, all Goroutines have no function calls in their control paths. Consequently, when the main thread goes to sleep, all cores are taken by the spawned Goroutines, and they never yield, leading to the main thread being blocked indefinitely. This sounds rather catastrophic.

Fortunately, Go 1.14 came along and solved this issue by introducing a per-core system-level monitoring thread - sysmon and a signaling mechanism. Sysmon monitors the execution time of each Goroutine, and if one takes control of the CPU for an extended period, it sends a signal to the kernel-scheduled thread that that Goroutine runs on, forcing it to take another execution path that yields the CPU.

Inspired by how Go's preemptive scheduling on the user land is achieved, we aim to implement similar but fully preemptive user-level threading for the C programming language. With some research, there isn't much work done on the user-scheduled threading support for C, and even fewer such solutions have been adopted in real-world application scenarios. Therefore, we have decided to develop one in C, with the following goals in mind: to keep it as simple as possible to avoid any unnecessary overhead and to make it as portable as possible, avoiding the use of any OS-dependent context structures as well as any deprecated libraries often found in other implementations, such as `<ucontext.h>`.

2 Overview

We have chosen Pthreads as the kthreads[1] on which uthreads[2] run, for short. This selection allows us to utilize thread-local storage, and we will delve into its importance later. Each Pthread is referred to as a software core or **softcore**, for short. Their number should correspond to the available hardware cores.

We associate each softcore with additional information, including an ID, a work queue for user threads, and a pointer that references its scheduler context. Each softcore has its own scheduler context, resulting in a per-softcore scheduler, for short. The scheduler running on each softcore possesses its own stack space, which is independent of the stacks allocated for the user threads.

These scheduler stacks are, in fact, the very stacks that the Pthread library allocates during the creation of these softcores in the `pthread_create()` calls, for short. This separation allows us to distinguish the scheduler context from the user thread context and utilize the stacks allocated during the creation of the softcores (Pthreads).

We employ the signal-yield model to enable preemption. During initialization phase, our system registers an interval timer using `setitimer()`. Whenever the timer interval expires, the OS delivers a SIGALRM to a random softcore, which then propagates the signal out to the rest of the softcores in a chain. Each softcore upon receiving the SIGALRM will invoke a signal handler that calls `yield()` yielding control to the scheduler. The scheduler then picks up from where it left of and schedule the next runnable uthread. The

[1]We will refer to kernel-scheduled threads as kthread

[2]We will refer to user-level/scheduled threads as kthread

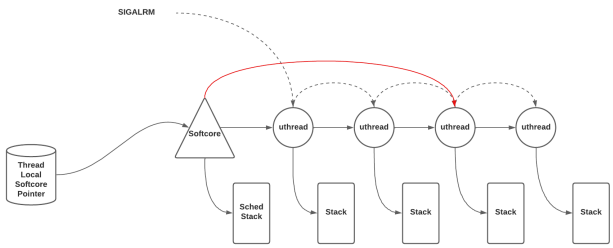

Figure 3: Overview

actually implementation of such a context switch is rather complicated since it involves writing assembly and requires a deep under standing of Linux or in general POSIX signal stacks and needs some careful synchronization through locking. However, in a nutshell, all the scheduler does is to switch to the stack of the selected runnable uthread and set the instruction pointer to where it was before it switched to the scheduler. And a thread can either be in the middle of acquiring a sleep lock which caused it to have descheduled itself or, in a more complex case, in the signal handler. For the former, when that uthread is scheduled it's guaranteed to have been assigned to be the lock owner and can carry on entering the critical section; and for the latter, the uthread simply returns from the signal handler, usually through returning to some OS created trampoline code that invokes a sigret syscall to return from the signal handler in which the kernel restores the uthread stack to what it was before the signal handler runs which usually involves popping the signal stack on which lives a copy of the CPU context of the uthread before being signaled.

Figure 3 provides an overview of our design where we use triangles for softcores, circles for uthreads, and rectangles for stacks. The cylinder in the diagram represents Pthread local storage. When a running uthread calls the `yield()` function to transfer control to the scheduler, it must determine the softcore it is currently running on. This knowledge is essential to identify the appropriate scheduler to yield the CPU to, or in other words, to determine which scheduler stack to switch to. The Pthread local softcore pointer simplifies this process, allowing the uthread to easily locate the softcore structure pointed to by the pointer. From there, it can identify the associated scheduler stack and perform the necessary stack switch. The Pthread local storage comes handy since whenever a uthread runs, it runs on top of the Pthread and can always use Pthread local variable to retrieve the information about which Pthread it runs on, or more specifically, which softcore it runs on. Another nice thing about using Pthread local storage is that any uthread can retrieve all information about itself or its TCB [3] struct within constant time by finding the struct of the softcore it runs on in which there is a pointer that points to the TCB of the uthread that's current running, which is the uthread who's trying to inspect its own tcb struct itself, and that's especially useful in calls such as `uthread_self()` that returns a pointer to the TCB of the caller uthread.

3 Implementation

3.1 Signal propagation

Figure 4 shows our code for signal propagation in the signal handler. In this snippet, *mycore* represents the current software core. It includes a thread-local variable named *term*, which is initialized to 0. The variable *g* holds a data structure containing global variables, including an attribute named *cores*. This attribute is an array of software cores; a specific core can be accessed by array indexing. Signal propagation is initiated by incrementing the *term* variable of *mycore*. This value, now referred to as *my-term*, is then compared with the *term* variable of the subsequent thread—denoted as *next-term*—which is obtained by incrementing *mycore*'s index by one and modulo by the number of cores and use it as the index to access the *cores* array. If *next-term* is less than *my-term*, it means the signal has not been propagated to the next thread thus we propagate this signal to the next thread by using pthread_kill. If *next-term* is equal to *my-term* it means that the next

[3] TCB stands for thread control block and contains all the information about an uthread including id, state, sp (stack pointer for saving and restoring the execution context), etc.

thread is the thread that first received the signal which means that we are finished with this round of signal propagation.

```
void sigalrm_handler(int signo) {
    ...
    mycore->term++;
    struct core *nextcore =
    ↪  &g.cores[(mycore->id + 1) %
    ↪  g.ncore];
    if (nextcore != mycore &&
    ↪  nextcore->term < mycore->term
    ↪  && nextcore->pthread != 0)
        pthread_kill(nextcore->pthread,
        ↪  SIGALRM);
    ...
}
```

Figure 4: Signal Propagation Code

3.2 Scheduling

Preemptive user-level scheduling, as mentioned before, is achieved using the signal yield model. A uthread's execution can be preempted by the delivery of a signal, after which the signal handler hands control over to the scheduler. The scheduler then finds the next runnable uthread and schedules it to run.

However, there is a catch: the delivery of a signal is a software event and has no ability to interrupt the execution of a program on a CPU asynchronously. It's commonly mentioned in textbooks that Linux signals are handled when a process or thread is scheduled, which is true in most cases. Imagine the following situation: we have two threads, A and B, and A sends a SIGTRAP to B. The earliest time for that SIGTRAP to be caught and handled by the kernel within B's time quota is when B is scheduled to run. Thus, there is a lack of real-time sense, making it unsuitable for our purpose. We aim to divide the time quota allocated by the kernel to one kthread into smaller chunks via signal deliveries. The delivery of each signal should be handled within the time quota of that backing kthread instead of when that kthread is scheduled again. Signals may not be suitable for this task, as there is always some kind of delay, sometimes up to one scheduling round.

However, there are certain situations in which signals can be handled almost in real-time with minimal delay that is almost negligible. Our user thread library registers for hardware timer events via 'setitimer()' that expires after a certain period. When it expires, the processor gets interrupted and enters kernel mode. From there, this hardware timer event gets translated into a signal and delivered to the kthread backing one of our softcores. The signal gets handled right before the processor switches back to user mode, which is more timely than waiting until the next time this kthread is scheduled.

Figure 5 shows how we switch from A to B without A cooperatively yielding to B. Let's say A is running, and the hardware timer expires. The processor running A takes a trap and enters kernel mode. The kernel's timer interrupt handler takes over, identifies the kthread that registered this hardware event of the timer expiring, generates a SIGALRM, and delivers it to the kthread's TCB (thread control block) residing in the kernel space. In each thread's TCB, there is a pending signal array, usually a bit set where each bit represents one type of signal, setting which indicates the receiving of the corresponding signal.

After handling the timer expire event, the kernel tries to return to user mode. At this time, something important happens, directly enabling the real-time handling of hardware events from the user mode - the Linux kernel checks for any pending signals in the TCB of the kthread it is going to transfer control back to. Doing so, it naturally sees the SIGALRM bit it set previously when handling the timer event and invokes the user signal handler. As you can see, the signal is generated right after the processor takes the trap and handled right after the processor exits from the trap, with no waiting until the next scheduling round. That is, I think, the most under-appreciated beauty of the Linux signal: it translates a hardware event into a software event and enables those events to be handled in a real-time manner, the same as how software events are handled.

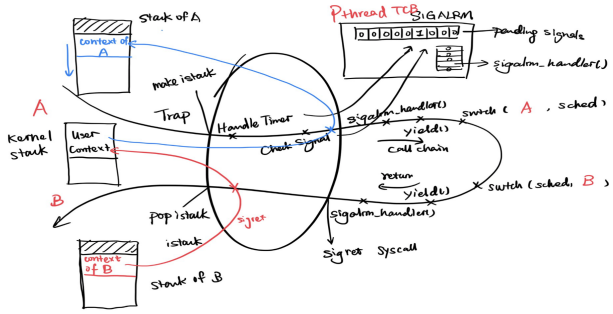

Figure 5: scheduling-1

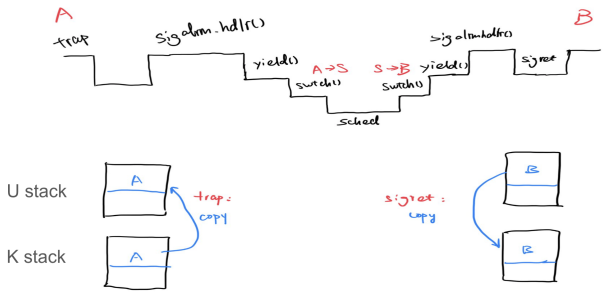

Figure 6: scheduling-2

The user signal handler is exactly the handler we registered, in which we implement 'yield()', which then calls 'swtch()', switching from uthread A to the scheduler. The scheduler runs and picks the next runnable thread, B, and 'swtch()' to B.

B then returns from 'swtch()', 'yield()', and then the signal handler in the reverse order of the process A has just gone through. You must notice this symmetry here. It exits because the reason B is marked as runnable in the run queue and gets picked by the scheduler is that it is the very same set of steps that A has just gone through that took B to where it is now. If you think of the situation of A right after calling 'swtch()', its context stops in the middle of 'swtch()' and it becomes RUNNABLE, waiting to be picked by the scheduler, as B is just picked up by the scheduler. It is the same situation as B up until P gets converted to RUNNING by the scheduler. Therefore, the return sequence of B is just the reverse of the calling sequence of A.

When B returns from the signal handler, it returns to a snippet of trampoline code that's part of the kernel executable but is marked as user-readable and -executable. There, B calls 'sigret', which is a syscall that restores B's context as it is at the moment B takes the trap.

Figure 6 possibly provides a clearer illustration of the same process, shedding light on the saving and restoration of contexts A and B. When A takes the trap, the kernel saves A's context on the kernel stack. Subsequently, when the user signal handler is invoked, the kernel copies A's context to the user stack—allocated to A by the thread library. After the scheduler switches from A to B, moving from A's stack to B's stack, the kernel loads the context of B into the kernel stack. This loading occurs from B's stack, where it was initially saved when B got interrupted and invoked the scheduler. Thus, upon returning to user space, the kernel restores the context of B rather than that of A. As you can observe, the actual context switch still occurs with assistance from the kernel.

3.3 Locking

In order to avoid using the heavy weight kernel locks, we implemented our own light weight locks which doesn't enter the kernel for locking and unlocking.Specifically we have two types of locks: Spin-Lock and Sleep-Lock.

3.3.1 Spin-Lock

Spin-locks are a fundamental synchronization mechanism used to protect data structures within a software core. Figure 7 illustrates the implementation of acquiring a spin-lock. The critical aspect of this process is that it begins by disabling signals on the core. This step is crucial to avoid potential deadlocks that might occur if a signal handler tries to acquire the same lock. The core of the spin-lock mechanism is the xchg instruction, a hardware-level atomic operation provided by x86 architectures. The xchg instruction is used to atomically swap values, which in the context of spin-locks, is employed to implement an atomic compare-and-swap operation. When attempting to ac-

quire the lock, xchg exchanges the current value of the locked variable with 1, indicating that the lock is being acquired. It simultaneously returns the previous value of locked. If the returned value is 0, it indicates that the lock was not previously held and has now been successfully acquired by the current thread. The surrounding while loop is a form of busy-waiting. It continually invokes the xchg instruction until the lock is acquired. This means that if the lock is already held (indicated by xchg returning a value other than 0), the thread will continuously loop, effectively 'spinning', until the lock becomes available.

```
void acquire(struct spinlock *lk) {
    // Disable signals on this core
    sig_disable();
    // Record signal status when no
    ↪  locks held
    if (!mycore->lkcnt)
        mycore->sigen_nolk =
        ↪  !(mycore->sig_disable_cnt >
        ↪  1);
    // Increment lock count
    mycore->lkcnt++;
    while (xchg(&lk->locked, 1));
    // Record ownership
    lk->owner = mycore;
}
```

Figure 7: Acquiring Spin Lock

3.3.2 Sleep-Lock

Building upon the concept of the Spin-Lock, our team has innovated a new type of locking mechanism known as the Sleep-Lock. This development was driven by the realization that in certain scenarios, busy-waiting for a lock to be released is not the most efficient approach, particularly when the lock is held by a thread that isn't actively running. This situation often occurs when the thread holding the lock and the thread seeking the lock are allocated to the same software core. In such instances, if the thread holding the lock is not executing, a thread that resorts to busy-waiting for the lock ends up consuming valuable CPU time and resources without making any real progress. This is where the Sleep-Lock offers a significant advantage. Instead of busy-waiting, the Sleep-Lock allows a thread to enter a sleep state when it encounters a lock held by a non-running thread. This strategy serves a dual purpose. Firstly, it conserves CPU resources by suspending the waiting thread, thus preventing it from using up CPU cycles unproductively. Secondly, it yields directly to the thread holding the lock, allowing it to complete its task and release the lock more quickly. Once the lock-holding thread has finished its execution and released the lock, the sleeping thread is awakened. It can then attempt to acquire the lock and proceed with its own execution. This method enhances overall system efficiency by reducing unnecessary CPU load and improving the management of thread execution.

Figure 8 presents the implementation of the sleep lock acquisition process. The procedure begins by obtaining a spin-lock. Following this, the code evaluates whether the lock is currently owned. In the event the lock lacks an owner, the code designates the executing thread as the new owner and subsequently releases the spin-lock. However, if the lock is already owned, the code adds the current running thread to the waiting queue of the lock and adds an additional check to determine if the owner thread is executing on the same software core as the current thread. Should this be the case, the current thread's state is set to 'sleep', after which it releases the spin-lock and yields control to the owner of the lock. Conversely, if the owner of the lock is operating on a different core, the lock's functionality reverts to that of a standard spin-lock. This mechanism ensures that the lock adapts its behavior based on the relative positioning of the threads on the software cores, thereby optimizing resource utilization and thread management.

Figure 9 illustrates the procedure for releasing the sleep lock. Initially, the spin lock is acquired. The next step involves checking the state of the waiting queue. If the queue is found to be empty, the lock's ownership is reset to zero (indicating no current owner), and the spin lock is then released. In contrast,

```
void sleeplock_aquire(struct sleeplock
↪  *lk) {
    struct tcb *thr;
    struct tcb *oldowner;
    struct waiter waiter;
    thr = mycore->thr;
    waiter.thr = thr;
    waiter.next = 0;
    acquire(&lk->lk);
    assert(mycore->thr != lk->owner);
    if (!lk->owner) {
        lk->owner = mycore->thr;
        lk->ownercore = mycore;
        release(&lk->lk);
    } else {
        waiter.next = lk->waiters.next;
        lk->waiters.next = &waiter;
        if (lk->ownercore == mycore) {
            thr->state = SLEEPING;
            oldowner = lk->owner;
            release(&lk->lk);
            yield_to(oldowner, lk);
        } else {
            release(&lk->lk);
            while (lk->owner !=
            ↪  mycore->thr);
        }
    }
}
```

Figure 8: Acquiring Sleep Lock

```
void sleeplock_release(struct sleeplock
↪  *lk) {
    acquire(&lk->lk);
    if (!lk->waiters.next) {
        // No one's in the waiting
        ↪  queue. Set owner to 0.
        lk->owner = 0;
        release(&lk->lk);
    } else {
        // Assign the first waiter in
        ↪  the queue as the owner.
        struct tcb *thr =
        ↪  lk->waiters.next->thr;
        // Remove the first waiter from
        ↪  the queue.
        lk->waiters.next =
        ↪  lk->waiters.next->next;
        // Assign owner.
        lk->owner = thr;
        // Mark the new owner as
        ↪  runnable.
        thr->state = RUNNABLE;
        release(&lk->lk);
    }
}
```

Figure 9: Releasing Sleep Lock

if the waiting queue contains one or more threads, the process involves removing the first thread in the queue and assigning it as the new owner of the lock. Subsequently, the state of this thread is changed to 'runnable'. Finally, the spin lock is released, completing the process of transferring lock ownership and resuming thread execution.

4 Performance Evaluations

Figure 10 illustrates the specific task executed by each thread in our performance evaluation. This task consists of a function that increments a variable repeatedly—a total of 100,000 increments. To assess performance, we conducted tests with thread counts set at 10,000, 20,000, and 30,000, executing these tests on configurations utilizing both single and dual-core processors. The outcomes of these tests are presented in Figure 11.

In both single and dual-core setups, our implementation of user-level threads (uthreads) demonstrates superior performance compared to POSIX threads (pthreads), as evidenced in Figure 11. Here, uthreads display a nearly twofold increase in speed relative to pthreads. It is also important to highlight that uthreads are configured to switch contexts every 1 millisecond, in contrast to the 10-millisecond switch interval of pthreads. This discrepancy suggests that should the context switch frequency be equalized, the performance disparity could be even more pronounced. The minimal overhead associated with context switching in uthreads, when compared to pthreads, is evident. Also, we expect to see more performance lag if we

```
void* thrfunc(void *arg)
{
    int id = (int)arg;
    for (int i = 0; i < NITER; i++)
    counters[id]++;
    return 0;
}
```

Figure 10: Task used for performance evaluations

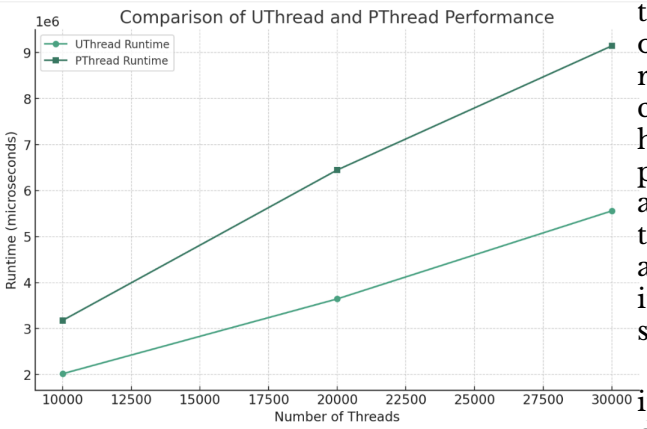

Figure 11: Runtime vs Num of threads for Pthread and Uthread on 2 cores

increase the number of cores as with more cores available, the benefit of efficient context switching is amplified as more threads can run in parallel, and context switches happen more frequently.

5 Future Work

For future work, our primary objective is to implement work stealing, with the aim of achieving a more balanced workload distribution. The idea is that when a softcore completes its assigned tasks and empties its work queue, it should be able to proactively "steal" tasks from the work queues of other softcores. This strategy prevents a softcore from idling, enhancing overall system efficiency. Notably, our current implementation already incorporates distributed work queues, which should facilitate the integration of work stealing. However, introducing work stealing also presents a challenge, as it might lead to increased locking overhead. Each work queue having its lock implies that we must refine our locking discipline to optimize performance in this context.

Regarding the sleep lock mechanism, our existing approach involves downgrading to a spin lock when two threads, competing for the lock, reside on different processor cores. An alternative approach we're considering is transferring the ownership of the lock from the run queue of one softcore to that of another, and subsequently yielding to the corresponding thread. However, this alternative comes with the trade-off of additional overhead. To make an informed decision, we plan to implement and rigorously test this approach against the spin lock implementation, considering various scenarios. Additionally, we might explore adopting a hybrid locking mechanism, tailoring our choice based on specific situations and system conditions.

In the realm of system calls, we are looking to implement a syscall wrapper, a crucial component for handling blocking syscalls like read. In our current version, these syscalls can consume time quotas even when no data is received. To address this issue, we are considering the strategy of yielding to other uthreads before initiating blocking syscalls. Achieving this efficiently requires a deeper understanding of the Golang source code, prompting us to conduct an in-depth study.

Similarly, a signal wrapper is on our radar. Given that the runtime relies significantly on signals, we recognize the need for caution to prevent user signals from disrupting our timing and to ensure that the runtime does not unintentionally interrupt user signals. Implementing this wrapper involves taking careful precautions to maintain the integrity of signal handling.

Finally, a substantial effort is directed towards developing a glibc alternative rather than simply adding a wrapper function layer. This approach is driven by the realization that using glibc wrapper functions in the current context could lead to inefficiencies, as these functions are optimized specifically for pthreads. The development of a dedicated alternative is a key step in ensuring optimal per-

formance and compatibility within our system architecture.

References

[1] "Go 1.14 Release Notes," The Go Programming Language, 2020. [Online]. Available: https://go.dev/doc/go1.14. [Accessed: Nov. 5, 2023].

[2] R. Cox, F. Kaashoek, and R. Morris, "xv6: a simple, Unix-like teaching operating system," 2020. [Online]. Available: https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf. [Accessed: Nov. 5, 2023].

[3] L. Torvalds, "linux/kernel at master," GitHub. [Online]. Available: https://github.com/torvalds/linux/tree/master/kernel. [Accessed: Nov. 5, 2023].

[4] S. Shiina, S. Iwasaki, K. Taura, and P. Balaji, "Lightweight preemptive user-level threads," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2021, pp. 374–388. doi: 10.1145/3437801.3441610. [Accessed: Nov. 5, 2023].

[5] D. P. Bovet and M. Cesati, "Understanding the Linux Kernel," 3rd ed., O'Reilly Media, Inc., 2005. [Online]. Available: https://www.oreilly.com/library/view/understanding-the-linux/0596005652/. [Accessed: Nov. 5, 2023].

[6] "Commentary on The Linux Kernel Version 0.01," OpenSource, 2019. [Online]. Available: https://opensource.rezaervani.com/wp-content/uploads/2019/05/The_Linux_Kernel_0.01_Commentary.pdf. [Accessed: Nov. 5, 2023].