The George Washington University

Advanced Operating Systems

M:N Threading with Preemption

Specification by G. Parmer

1 Part 1: M:N Threading

In this part, we will add real concurrency to your `lwt` implementation! This assignment will force you to think about preemptive scheduling, along with your cooperative scheduling implementation, and trade-off concurrency for performance. Specifically, preemptive environments allow you to handle non-deterministic events, along with prioritizing specific execution over other. This is generally necessary in computer systems to effectively interact with I/O, and to control the latency and interactivity of different operations.

M:N threading designates having M user-level threads managed by your `lwt` library that execute on top of N kernel threads. As kernel threads are multiplexed by the kernel, their context (stack and registers) are saved and restored when interrupts occur, which enables scheduling to be non-cooperative (i.e. preemptive). However, switching between these kernel threads (using `cos_defswitch`, for example) requires a system call via `sysenter` + `sysexit`, thus has significant overhead over `lwt` switches.

1.1 New APIs

First, we have to have the ability to create actual 1:1 threads in the system. We will generally call these threads `kthd`s, or threads that are scheduled by the kernel. In our case, we will use the *Composite* support for creating sl_thds which we will refer to as `kthd`s in this document. We will implement a system where specific `lwt` threads are bound to one of the `kthd`s. In this case, `lwt` threads do not (yet) migrate between `kthd`s. Channels are referenced between different `kthd`s and are the only way in the API to communicate between `kthd`s, but still only have one *receiver* (thus are, in a sense, bound to the `kthd` of that receiver).

Note that for this part of your assignment, you will have to spend a significant amount of time reading the

src/components/implementation/no_interface/scheddev/* files, and asking questions on Piazza. The main functions that you will likely use are sl_thd_alloc, sl_thd_yield, sl_thd_block[1], sl_thd_wakeup, sl_thd_curr, and sl_thd_param_set. Don't forget that you have to use sl_init and sl_sched_loop correctly. If you need to store some kthd-local data (called TLS, Thread Local Storage) to store some data per kthd, you can use cos_thd_mod in conjunction with the __thread storage modifier in C (examples in the micro_booter), or you can use other techniques.

For information about the rest of the Composite kernel API, most of it is used in
src/component/implementation/tests/micro_booter/. Allocation of pages of memory can be done using cos_page_bump_alloc.

We add the following API:

- int lwt_kthd_create(lwt_fn_t fn, lwt_chan_t c) – This function creates *both* a kernel-scheduled thread – a sl_thd in our case – and a lwt that is created to run on that kthd. That lwt thread on the kthd calls the fn as with the normal lwt_create API sending the channel as normal. The difference here is that the lwt thread is executing on a separate kthd, rather than the current one. Note that we don't return a lwt_t as we want to partition the lwt threads across the kthds (i.e. that lwt_t should only be accessed within the new kthd). Instead, return 0 if we successfully create the thread, and -1 otherwise.
- int lwt_snd_thd(lwt_chan_t c, lwt_t sending) – This is equivalent to lwt_snd except that a thread is sent over the channel (sending is sent over c). This is used to migrate lwt_t threads between different kthds, thus to change the allocation of lightweight threads to actual kernel threads, and might be used in load-balancing work between kernel threads. A thread *cannot* send itself. Return values are the same as for lwt_snd.
- lwt_t lwt_rcv_thd(lwt_chan_t c) – Same as for lwt_rcv except an lwt_t thread is sent over the channel. See lwt_snd_thd for an explanation.

It is tempting to add additional library functions to create a lwt thread on a given kthd, but such functions can be implemented using the normal channel communication between lwt threads (where the two communicating lwt threads are on separate kthds, and one asks the other to create a "local"

[1]The sl library does not currently support dependencies on block, so you should always pass in 0, and effectively ignore the parameter. It is there for future expansion.

thread).

You must still support communication via channels between `lwt`s on different `kthd`s. This is the main means of communication between `kthd`s. To maintain high performance, we wish to avoid sharing data-structures between different `kthd`s. Doing so would require locks, and heavy-weight concurrency management and synchronization primitives. The API we've created has very simple *ownership* rules: the `kthd` a `lwt` thread is executed on owns that `lwt` thread; a channel is owned by the `lwt` thread that is the *receiver* of that channel; a channel group is owned by the `lwt` thread that created the group. The ownership rule is transitive, so a `kthd` owns all objects owned by one of the `lwt` threads it owns. We can use the simple rule for maintaining high performance, and simplicity: *a* `lwt` *thread owned by a* `kthd` *cannot modify an object that is owned by a different* `kthd`. The only exception to this rule is in the channel implementation. This (along with the reference counting we already do) means that we do not have to use any locks!

The `kthd` owner of a thread/channel/group can be changed by `lwt_snd_thd`ing a thread over a channel that belongs to a `lwt_t` owned by a different `kthd`. This will have the side-effect of removing the thread from all of the scheduling structures on the sending `kthd`, and adding the thread on the receiving `kthd`. The thread should *not* be scheduled while being transferred (i.e. before it is `rcv`ed).

Note that we will use the terminology "remote" to mean an object owned by a different `kthd` from the current one, and "local" to mean an object owned by the same owner of the current `lwt` thread.

1.2 Channels for Inter-`kthd` Communication

So that different `kthd`s of the system can interact, we do have to make modifications to channels that are owned by other `kthd`s, so how can we do so? We keep the same channel API as in previous assignments, but some channel `snd`s will result in communication between separate Here we will use simple message passing between `kthd`s. We will use ring-buffers for communication between different threads. You have a number of different choices in how to implement this.

1. Each `kthd` can have a ring buffer that is written into when "modifications" to the channels and groups are made. We will call this ring-buffer, the inter-`kthd` buffer. In this implementation the inter-`kthd` ring buffers are

not associated with any specific channel, and are instead simply mechanisms for inter-thread communication (a valid, alternate implementation would use such buffers for each channel). These buffers should be synchronized between different `kthd`s. A `lwt` thread might be responsible for communicating over the `kthd` ring buffers to the other `kthd`s, and it might act as a *proxy* for each channel shared between different `kthd`s. One of the downsides of this approach is that channel communication between `kthd`s involves multiple channel operations and thread switches.

2. Each channel can be implemented in a way that is safe for both local and remote communication. They must be synchronized, but still efficient. This has the down-side that it might make common-case operations somewhat more inefficient, but likely makes inter-`kthd` communication more efficient. This implementation might also make monitoring `kthd` status (blocked, or active) more difficult (see the notes on blocking in the next section).

You can assess the challenges and trade-offs of both implementations, and decide which you choose to implement.

1.3 Synchronization Between `kthd`s

Synchronization between different `kthd`s can be done in a number of ways.

1. The simplest is to use the critical section facilities of the `sl` code. You can use `sl_cs_enter` and `sl_cs_exit` which create a critical section that provides mutual exclusion between different `kthd`s. You will get partial credit based on the completeness, correctness, performance, and simplicity of any solution based on this synchronization mechanism.
2. You can use atomic instructions to synchronize modifications to shared channel data-structures between threads. The ring buffers used for channel-based communication can be implemented using atomic instructions for modifications. There might be instances where one `kthd` is observing partial modifications of another `kthd` (because a preemption stopped it from completing the modifications), and in that case, a directed yield can be used to "help" the thread doing modifications so that it can complete them. When using atomic instructions (compare-and-swap via `ps_cas` and fetch-and-add via `ps_faa`), you have to be care that all modifications are somehow visible with the successful completing of a single atomic instruction. To learn more about non-blocking data-structures that don't use locks, read `https://www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf`. To

get maximum credit for this assignment, you must have a correct, fast implementation that uses atomic instructions for synchronization instead of critical sections.

Additionally, you must explicitly support blocking and waking up `kthd`s. You *cannot* use "spinning" for any of your synchronization code. Since we're using a single CPU, spinning only wastes power, performance, and generates heat. Each `kthd` should block when it has no processing to accomplish (all its `lwt` threads are blocked). A `kthd` should be woken up when it is blocked, and a `snd` is made to any of its local channels. Avoid race conditions with multiple `kthd`s waking up a single other `kthd`.

1.4 Modifying Your Implementation

First, it should be clear that any operations to objects on a local `kthd` are unchanged. Whereas on send or receive to local objects where direct modifications are currently made, when the object (e.g. channel) is on a remote `kthd`, notifications of these changes must be sent as messages via the inter-`kthd` buffer to the `kthd`. These messages will include any pointers and information necessary to modify the channel and the `lwt` thread that is currently blocked on its owning `kthd`.

The following subsections of your current implementation will need to be modified:

- *Channels.* To pass notifications that a channel has been sent to or received from between two `kthd`s, corresponding events need to be produced, and consumed on the remote `kthd`. This means that instead of adding a thread that must block on a synchronous send or receive, we would instead send a message via the inter-`kthd` buffer to the owner `kthd` asking it to block the `lwt` thread. Note that though this (now remote) `kthd` would be blocking the remote `lwt` thread, it is "blocked" on its owning `kthd`, thus avoiding any race conditions. Note that if the remote thread must block (due to synchronous behaviors), then a message will need to be sent back to the original `kthd` to wake the thread back up when it again becomes runnable. There are other ways to implement this, and you're free to use them, but ownership constraints on the different objects must be adhered to.
- *Group/multi-wait.* When a channel is owned by remote `kthd`, and a send or receive on the local `kthd` triggers an event on that group, this event should be sent as a notification via the inter-`kthd` buffer. You might not need to

do this separately from the event sent for channels.

- *Threads.* Passing threads between different schedulers modifies the owner of the thread, and all of the channels and groups that the thread owns. Transferring the ownership must be done carefully as you likely want to avoid *all* synchronization complications where two threads are modifying it, or the scheduling structures for a given `kthd` are modified by another.

All of these interactions should be implemented carefully. You need to examine your code and determine, for a remote object modification, when the thread in question is writable by its owner, and when it is modified by the remote `kthd` (i.e. when it is placed into the inter-`kthd` ring buffer). Note that at all points in time, you want to make sure that the object is only modifiable by a single `lwt` on a `kthd`.

1.5 A Possible Implementation Strategy

My suggestion is to break the work for this into a few phases:

1. *Data-structure design.* You must plan how your `kthd` control block is related to other data-structures. I suggest a very simple structure in which each data-structure points to the `kthd` that owns it. This makes it very simple to see if an operation is local or remote by comparing the owner of the current `lwt` thread with the object's owner. Answer the simple questions: How do I determine if an operation is on a remote object? How do I find the run-queue? How does an operation that must be performed on a remote object, know where the `kthd` of that object is, and how does it locate that `kthd`'s ring-buffer?
2. *Break operations into local and remote logic.* This is by far the most important aspect of the implementation. Look at your channel operations, and determine what parts of the implementation can be performed locally, and what must be performed remotely (on the local `kthd` for an object)? Most of the logic for this should be in the `lwt_snd` and `lwt_rcv` functions, but there will be some in `lwt_cgrp` functions as well. Your entire implementation should not require lock-based synchronization at all!
3. *Idle threads.* A side-effect of this change is that it is possible to have *no threads* active on a specific `kthd`. You likely want to add idle threads that should only be executed when there are no other threads to execute on a specific `kthd`. Additionally, once you add idle threads, you'll have to add the interface between them and the ring-buffers. After this is working, then

you should add the condition variables to block the `kthd`s.

4. *Thread blocking and waking up.* You have carefully consider when a `kthd` (thus all its corresponding `lwt` threads) should be blocked and woken up. How do you detect the blocking of one `kthd` in another so that it can be woken up?
5. *Communication ring buffers.* After consider all of the rest of the issues, and if you are going to use shared channel ring buffers, or per-`kthd` ring buffers, then you should design the data-structure for the buffer itself. The most important issue is to understand the concurrency issues, and make your design around that.

2 Part 2: Performance

Everything is now a trade-off. If you maintain fast local operations, then you likely increase the cost of remote operations, and vice-versa. Please document all of these trade-offs in a `TRADEOFFS.md` file in your repo.

System	context switch	pipe/channel	proc/thd create	event notification
Linux/`lmbench`	lat_ctx	lat_pipe	lat_proc	lat_select
lwt local hw3	yield	snd/rcv	lwt_create	grp
lwt local	yield	snd/rcv	lwt_create	grp
lwt remote	yield	snd/rcv	lwt_kthd_create	grp

Table 1: Operations that you should use for a performance comparison. Please emulate the structure of what `lmbench` does for their evaluation.

Please use `lmbench` to evaluate the costs for various Linux operations, and compare them to those of your library, both for local and remote operations. Table 1 shows which operations should be compared. For all of the `lmbench` versions, you'll need to look in the `lmbench` source to see how they are evaluated, and attempt to mimic that. For example, if measuring pipe latency is done by setting up N threads that form a "ring" where a message is passed around them, then you'd want to set up N `lwt` threads that pass a message around a set of channels. All of the "snd/rcv" operations should be performed using synchronous channels (size 1), channels with size 8, channels with size 64, and channels with size 512. Note that the "lwt local" evaluation should measure local operations, while "lwt remote" measures all remote operations. The "lwt local hw3" provides your implementation from homework 3 that focuses only on local operations. This serves as a "baseline" for how expensive some of the trade-offs you made in this homework impacted performance. You should write

a separate test file that runs these tests for the libraries.

3 Part 3: Qualitative Comparison

I'd also like you to write a document comparing your `lwt` library and its APIs against `go`'s APIs for creating `goroutines`, using channels for communication, its facilities for multi-wait, and how `go` handles concurrency (i.e. preemptions, and threads blocking on I/O).

For *extra credit* (not too much, but not an insignificant amount either), you can evaluate the performance and directly compare it against your lwt libraries, just as you did in Section 2.

4 Testing and Evaluation

There will be **no test file for this homework**. The success of your implementation (and your grade) will be determined partially by how complete your test file is. I'd start your test file from the one for homework three, and expand from there. If you do *not* test an important case, then it will be assumed that feature does not work. If your application tests the functions of your library, then it is sufficient to provide your application. However, I suggest that you write tests for your library to get it working before you write the application on top.