The George Washington University

Advanced Operating Systems

Programming Exercise - User-Level Threading – Synchronous Communication

The deadline is at midnight on the day specified on Piazza. The next assignment will likely be assigned before this deadline, so please shoot to finish early. As before, you *must* thoroughly test your implementation as I'll provide a test file rather close to the deadline.
Use the same teams you previously used, and if you would rather not, you *must* talk to me.

1 Synchronous LWT Channels

You will add synchronous communication to your user-level threading library. The API follows:

- `lwt_chan_t lwt_chan(int sz)` – Currently assume that `sz` is always `0`. In the future, this will be the size of the buffer for communication over the channel. This function creates a channel referenced by the return value. The thread that creates the channels is the *receiver* of the channel. This function will use `malloc` to allocate the memory for the channel.
- `void lwt_chan_deref(lwt_chan_t c)` – Deallocate the channel only if no threads still have references to the channel. Threads have references to the channel if they are either the *receiver* or one of the multiple *sender*s.
- `int lwt_snd(lwt_chan_t c, void *data)` – This function sends data `data` over a channel to the receiver. Channels are synchronous, so if the *receiver* is currently waiting for a send, then it will unblock. If a *receiver* is not waiting, then this sender is blocked. If other senders are currently blocked on the channel, then the current sender must be queued. Channels are serviced FIFO. The `data` *cannot* be `NULL`. If that is passed in, an assertion is triggered. If no receiver exists for the channel (because it called `lwt_chan_deref`), then return `-1`. Otherwise, it returns a `0`.
- `void *lwt_rcv(lwt_chan_t c)` – The *receiver* can call this function for its channel. This is the mirror function of `lwt_snd`. If a sender is blocked on the channel, wake it up, and return the data it was sending from this function. Otherwise, block the receiver until a sender calls `lwt_snd`. A return value of `NULL` means that no *senders* exist on the channel (either because they haven't been added to the channel, or because they have called `lwt_chan_deref`).
- `int lwt_snd_chan(lwt_chan_t c, lwt_chan_t sending)` – This is equivalent to `lwt_snd` except that a channel is sent over the channel (`sending` is sent over `c`). This is how *senders* are added to a channel: The thread that receives channel `c` is added to the `senders` of the channel. This is used for reference counting to determine when to deallocate the channel. Return values are the same as for `lwt_snd`.

- `lwt_chan_t lwt_rcv_chan(lwt_chan_t c)` – Same as for `lwt_rcv` except a channel is sent over the channel. See `lwt_snd_chan` for an explanation.
- `lwt_t lwt_create_chan(lwt_chan_fn_t fn, lwt_chan_t c)` where `typedef void *(*lwt_chan_fn_t)(lwt_chan_t)`. This function, like `lwt_create`, creates a new thread. The difference is that instead of passing a `void *` to that new thread, it passes a channel. The new thread is now a *sender* for that channel. This function is equivalent to the one passed to `lwt_create`, but that it takes a channel as its argument. Note that the channel will only be sendable from the created thread, so a protocol will be necessary for communicating from the parent to child. as it was created by another thread. The return value for the function is the same as in assignment 1, and it interacts with `lwt_join` identically.

Additionally, you must add to the utility function from before:

- `int lwt_info(lwt_info_t type)` – This function returns a simple integer representing a specific type of state of the `lwt` system. The type is selected with the argument. The `lwt_info_t` (which should probably be an `enum` can be the following values: `LWT_INFO_NTHD_ZOMBIES` (number of zombie threads that have `lwt_died`, but haven't been `lwt_joined`), `LWT_INFO_NTHD_BLOCKED` (number of threads that are blocked on an `lwt_join`), `LWT_INFO_NTHD_RUNNABLE` (number of threads in the run-queue), `LWT_INFO_NCHAN` (number of channels that are active, i.e. that have not been freed), `LWT_INFO_NSNDING` (number of threads blocked sending on channels), and `LWT_INFO_NRCVING` (number of threads blocked receiving on channels).

1.1 Example Channel Structure

Though you don't need to use this structure, this is an example of a subset of my channel structure:

```
struct lwt_channel {
    int snd_cnt; /* number of sending threads */
    struct clist_head snd_thds; /* list of those threads */

    /* receiver's data */
    struct lwt_tcb *rcv_thd;  /* the receiver */
};
typedef struct lwt_channel *lwt_chan_t;
```

1.2 Contemplate API Usage

To be able to properly understand how this API could be used for different purposes, you'll want to consider how to use this library to set up different communication patterns. For example,

- How can you create an abstraction using the library's mechanisms for bi-directional communication?

- What if a parent thread wants to create two children threads that can communicate through the parent?
- What if a parent thread wants to create two children threads that can communicate directly to each other?
- A "nameserver" will allow threads to handle requests of a specific type, enable other threads to ask the nameserver how to talk to the thread of a specified type, and enable them to communicate directly. For example, DNS is a nameserver for domain names on the internet. How would you implement this scheme?