The George Washington University

Advanced Operating Systems

Programming Exercise - User-Level Threading – Communication V2

You *may* continue to work in the teams you've already established.

1 LWT channels: async and multi-wait

Summary. For part two of the community lwt story, you will add two key capabilities, and one small feature addition. First, bounded communication buffers for limited asynchronous communication. A finite buffer is filled with sent data, and while it is not full, senders do not block. Comparably, receives do not block if there are data items in the buffer, instead simply returning one of them. Second, the ability to synchronously wait for a message on any one of a set of channels, instead of simply synchronously block only on a single channel. This *multi-wait* enables a single server to handle multiple clients without starvation. The small feature addition is to provide non-joinable threads.

All operations must be O(1) for this assignment.

Bounded asynchronous send/recv. The bounded buffers do not change the current API. Instead, note the `int sz` argument to `lwt_chan`. We slightly change the meaning of this value from the previous assignment. Synchronous communication results from passing `0`. Any value greater than `0` specifies the number of data items that can be asynchronously buffered. For the change, you will be required to:

- Implement a ring buffer that is written into on a send, and read out of on a receive to hold the buffered data. The ring buffer should be of the length passed into `lwt_chan` as the size. This enables the channel creator to control the degree of asynchrony. This buffer should be allocated at channel creation time, and have the same life-time as the channel. If it is freed (all references go away) while there is data in the channel, that data is lost.

Multi-wait. The new data-types for this support is the `lwt_cgrp_t`, which is the "set", or group of channels that we will wait on. The API includes functions to create and free a group, for adding channels to a group, removing them from the group, and for waiting for an event on any one of the channels associated with a group. When we discuss an *event*, we mean that a channel that does not have data that has been sent on it (thus a receive on it would result in the receiver blocking), now has data available. This change in state designates the event. Event notification APIs (which is what multi-wait is) provide means for a thread learning about which events happen, when.

For background on event notification APIs, read the documentation on `select`, `poll`, (Linux's) `epoll`, and (FreeBSD's) `kqueues`.

The API you must implement follows:

- `lwt_cgrp_t lwt_cgrp(void)` – Create a new channel group, or return `LWT_NULL` if one cannot be created.
- `int lwt_cgrp_free(lwt_cgrp_t)` – Free a channel group and return `0` only if there are no pending events. If there are pending events, return `-1`, and do not delete the channel. Do not free the associated channels.
- `int lwt_cgrp_add(lwt_cgrp_t, lwt_chan_t)` – Add a channel to a channel group. Any event that subsequently happens on the channel will be notified through the group. Additionally, any even that *already* happened and is still pending (i.e. a blocked receiver, or data in the asynchronous buffer) will also be notified through the group. A channel can be added into only one group (for simplicity). If you try and add a channel into two separate groups, this function returns `-1`, otherwise `0`.
- `int lwt_cgrp_rem(lwt_cgrp_t, lwt_chan_t)` – Remove a channel from a channel group. If the group has a pending event from the specific channel, do not remove it and return `1`. If it does not have a pending event, remove it from the group and return `0`. If the channel does not belong to this group, return `-1`.
- `lwt_chan_t lwt_cgrp_wait(lwt_cgrp_t)` – This is the function for which the others exist. This is a *blocking* function. The calling thread blocks unless there is a pending event on one of the channels. A blocking thread will block until one of the channels has an event.
- `void lwt_chan_mark_set(lwt_chan_t, void *)` – A thread that communicates with many other threads through separate channels might want to maintain different data-structures for each channel and client thread. However, tracking which of these data-structures corresponds to each channel is a little annoying (requiring a separate data-structure). *Mark*s solve this problem. A thread that receives on a channel can mark that channel by associating a `void *` with a channel that can later be accessed by the next function.
- `void *lwt_chan_mark_get(lwt_chan_t)` – This enables a thread to retrieve a previously set mark (i.e. a pointer to a data-structure).

There is one small final addition you must make:

- Add a new argument to `lwt_create`: `lwt_t lwt_create(lwt_fn_t fn, void *data, lwt_flags_t flags)`. The `flags` parameter is either `0` in which the default, previous behavior is used. If `flags & LWT_NOJOIN`, then the new child will *not* be `lwt_join`ed. If that child calls `lwt_die`, it will be deallocated immediately. You can assume that the parent will *not* call `lwt_join`.