Mthreads
Like uthreads but...

- M-to-N model: M uthreads multiplexed on N posix/kernel threads

- Kernel threads represented by pthreads
  
  - We call them *lightweight processes* (share address space)
  - uthreads now run within an `lwpt_t` (pthread, really)

- Your TAs have given you a functional uthreads solution; you’ll make it multiprocessor safe!
What to fix

- uthread.c:
   uthread_exit, uthread_join, uthread_detach, make_reapable

- uthread_cond.c:
   uthread_cond_wait, uthread_cond_broadcast,
   uthread_cond_signal

- uthread_sched.c:
   uthread_yield, uthread_block, uthread_wake, uthread_setprio,
   uthread_startonrunq, **uthread_switch**, lwp_switch, lwp_park,
   uthread_runq_enqueue, uthread_runq_requeue

- uthread_mtx.c:
   uthread_mtx_lock, uthread_mtx_unlock,**uthread_mtx_trylock**
uthread.h

uthread_t structs now have:

    pthread_mutex_t pt_mut

- Since we have a pool of M pthreads with user threads on top of them, we need to synchronize access to each uthread (just like synchronizing any other data!)

- Any kernel thread writing to a uthread_t should hold its associated mutex.
  - pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_init, pthread_mutex_destroy, pthread_create, pthread_detach, and more!

- $ man pthreads # ← for a refresher
uthread.h (2)

uthread_t structs now have:

   int ut_no_preempt_count;

Since preemption management is a per-user-thread concern, these variables are no longer global data.

uthread_nopreempt_{on,off} is still there, though you don’t need to worry about it.
typedef struct lwp {
    pthread_t ptid;        // returned by pthread_create
    uthread_ctx_t lwp_ctx; // LWP context, NOT THE RUNNING UTHREAD!
    utqueue_t *queue;      // queue to put a calling uthread on
    int saveonrq;          // current thread is to be put on the run q?
    pthread_mutex_t *pmut; // mutex to unlock after uthread_switch
} lwp_t;

In mthreads different uthreads could be running truly in parallel! This means we have to take special care when implementing functions that modify queues. Bugs where the same thread is being run on multiple processors (modifying the same stack) abound!
extern __thread lwp_t *curlwp;

- Like ut_curthr in uthreads, but each LWP has its own!

- Saves the LWP context while it’s being used to run other uthreads.

- Isn’t re-assigned on a per-LWP basis.

- __thread - thread-local storage class
extern __thread uthread_t *ut_curthtr;

- Represents the thread currently running on an LWP
- Each LWP has its own
- It’s re-assigned every time an LWP switches to a new uthread (lwp_switch)
- Familiar invariants from uthreads:
  - State should be UT_ON_CPU
  - Never on multiple queues, etc.
uthread_switch(utqueue_t*, uthread_t*, pthread_mutex_t*)

- Called to switch away from a uthread’s context into an LWP context
- Returned from once an LWP picks a new thread to run and switches into it
- Must be multiprocessor safe

LWPs will run many uthreads during their life-time!
- each call to uthread_switch must
  - save the current uthread context in &ut_curthr->ut_ctx
  - switch over “LWP context”, &curlwp->lwp_ctx
  - Not modify the run queue! (unlike in uthreads)

Since a uthread will be run on many different kernel threads, a uthread might “switch out” (call uthread_switch) on one LWP, but then “switch in” in another! This means we can’t let the compiler optimize away stack variables used to differentiate the first time we “return” from get_context from the second time (use volatile)
uthread_switch(utqueue_t*, int, pthread_mutex_t*)

- Arguments correspond to the fields in lwp_t

- We offload a lot of work from uthread_switch to lwp_switch

- Set these fields in uthread_switch, that way they can be accessed in lwp_switch, where we’re no longer executing in a user thread context!

  - Use this technique to prevent all kinds of weirdness

  - uthread might yield, put itself on the run queue, but then while it’s still executing be pulled off the queue by another kernel thread. Now it’s stack is being written to by two LWPs!
**lwp_switch()**

- Runs only in LWP context (should be switched into from `uthread_switch`)
- Scheduling decisions: find another uthread to run, switch into its context (resume in `uthead_switch`)
- Sleep in `lwp_park` if no threads found
- Perform operations requested by a uthread that just switched out from its context (arguments to `uthread_switch`)
- Must be multiprocessor safe, it’s running in a pthread and modifying the run queue!

Think of `lwp_switch` as running in “kernel mode”, independent of any particular uthread.
lwp_park()

- Called from lwp_switch when no other runnable threads are found on the run queue. LWPs block until lwp_release

- pthread_cond_wait
  - static pthread_cond_t lwp_park_cond

- Remember to update lwp_parked_cnt. Why shouldn’t lwp_parked_cnt == lwp_cnt?
uthread_mtx_trylock(uthread_mtx_t*)

- Return 1 if the calling thread acquires the lock, return 0 otherwise.
- The bottom line: don’t block
uthread_mtx.h

uthread_mtx_t structs now have:

  pthread_mutex_t m_pmut

- pthread_mutex takes place of a spinlock
- Remember that uthreads can now run in parallel, so they can access the same uthread_mtx_t while running on different LWPs at the same time.

- Relevant for functions in uthread_mtx.c
uthread_cond.h

uthread_cond_t structs now have:

```c
pthread_mutex_t m_pmut
```

- Same deal as for mutexes
A note about pthread_mutex_t’s

Remember that pthread_mutex_t’s are owned by a specific pthread (lwp_t), not by uthreads.

This means that if a lwp_t holds a mutex, it must be that lwp_t that unlocks the mutex. If you attempt to unlock a mutex from a different lwp_t, your program will crash.

What does this mean? It means that you cannot uthread_switch while holding a pthread_mutex_t because you cannot guarantee that you will be run on the same lwp_t next time.
How to debug a pthread_mutex_t deadlock with GDB
A note about the Reaper

Reaper “lock tests” uthread_t structs it pulls off the reap queue before calling uthread_destroy.

This gives the thread that is being destroyed the opportunity to ensure that its mutex is only unlocked after switching into LWP context (so it’s stack isn’t cleaned up until it’s done executing)

```c
while (reap_queue not empty) {
    next_thread = dequeue(runq);
    pthread_mutex_lock(next_thread->ut_pmut);  // lock test
    pthread_mutex_unlock(next_thread->ut_pmut);
    uthread_destroy(next_thread);
}
```
Recap

Many LWPs (pthreads) multiplex uthreads, running one at a time by switching between `uthread_switch (user context)` and `lwp_switch` (“kernel” context). LWPs wait for runnable threads in `lwp_park`.

All thread-library structs that can be accessed by many pthreads (whether they’re executing in user context or “kernel” context) have a guard `pthread_mutex_t`. This includes queues, (uthread)mutexes, condition variables, and uthreads themselves.

Uthreads will often need to request that some operation happens after they get off their stack. These request are made by passing arguments to `uthread_switch`. 
Async-signal safety

- Since we’re doing preemption using signals, we have to worry about signal safety
- Shouldn’t call “unsafe” functions (like `malloc` or `printf`) while preemption is on
  - Why?
    - 1) Could deadlock if these functions take a lock
    - 2) Could cause a crash if these functions maintain some internal state and get interrupted and then called again from a different place
    - 3) Could lead to torn writes.
- Don’t call `printf` in your test code. Use `write` instead
Questions?