```
CS214 / 2021-04-07
==================

Aside: managing memory

How do we make sure that every object we allocate gets deallocated exactly once?

Idea: every object has an "owner"
    - the owner of an object is responsible for freeing it
    - initially, the function that calls malloc() owns the object
    - ownership can change: functions can "give" objects to other functions
    - ways to transfer ownership
        - return pointer to transfer to caller
        - give pointer as argument to a function
        - incorporate object into some larger structure
    - functions can also "lend" objects to other functions
        - unlike transferring ownership, we expect to retain control
        - after the function returns, it should be safe to free the object
            - that is, there should not be any other references to that object

Example: a structure that holds a list of strings
    we can add and remove strings from it
    the structure itself "owns" the string objects it refers to

    void insert(stringlist_t *list, char *str);

        - we expect the list to persist after insert() returns, so we are
            lending the list object to the function

        - for str, insert() might borrow or take ownership of the string object

        - if insert() takes ownership of the object
             - it can just add the pointer directly to the list
             - callers cannot re-use the string object after calling insert()
             - we cannot pass pointers to global or local variables

        - if insert() borrows the object
            - it must make a copy of the string, because it cannot hold on to
                the string object that str points to
            - this would allow callers to pass in pointers to local variables
                and re-use the buffer

Again: this is a way of thinking about your program
    it is not a feature of C
    (it is a common strategy in C++, it is enforced by Rust)


// an insert that takes ownership of the string
void insert(stringlist_t *list, char *str)
{
    // create a new string node
    stringnode_t *node = malloc(sizeof(stringnode_t));
    node->string = str;
    node->next = list->head;

    // put new node at head of list
    list->head = node;

        // list now incorporates str
        // -> the owner of list is now the owner of str
}

// an insert that borrows the string
void insert(stringlist_t *list, char *str)
{
    // make a copy of str that we own
    unsigned len = strlen(str) + 1;
    char *s = malloc(len);   // s currently owned by this function
    memcpy(s, str, len);

    // create a new string node
    stringnode_t *node = malloc(sizeof(stringnode_t));  // node owned by this fun.
    node->string = s;
    node->next = list->head;

    list->head = node;
        // node and s now owned by owner of list
        // str still owned by its previous owner
}


// code that is only safe for the borrow version
... ( ... stringlist_t *list ...)
{

    char buf[BUFSIZE];

    while (...) {
        read(..., buf, ...)
        insert(list, buf);
    }

        // only okay if insert is borrowing buf
        // otherwise, we need to make a copy and malloc first
        //    (a) buf ceases to exist when the function returns, but list persists
        //    (b) we might re-use buf
}

Relevant to multithreading:

    we can pass a pointer to our thread function when we start a thraed
        pthread_create(&tid, NULL, thread_fun, thread_arg_ptr);

        -> notably, we are "lending" tid to pthread_create

    We must decide whether the thread function borrows or take ownership of its
        argument object

    -> if thread_fun takes ownership of the object, it will be responsible
        for deallocating it
        -> the caller should not assume that thread_arg_ptr points to anything
            after pthread_create returns
    -> if thread_fun borrows the object, then the caller is responsible for
        deallocating it
        -> this allows the caller to pass references to local variables/arrays
            to the thread function
        -> caller must not free or change the object until it knows the thread
            has completed (i.e., it has joined the thread)

---

Recap: multithreading so far

Creating threads
    pthread_create  -> creates a new thread that executes a given function
    pthread_join    -> ensures that the specified thread has completed
        -> creates a "rendezvous point"
            forces some synchronization between two threads
            e.g., we don't continue until the other thread has finished

        -> usually the creator of the thread will join it, but not always
            -> any thread can join any other thread
            -> we can think about a thread as having an owner, which will
                eventually join it

    pthread_exit  -> stops the current thread
        -> equivalent to returning from the thread function

        void
        pthread_exit(
            void *retval   // value to return from the thread
        )

    these exist, but we won't need them:
    pthread_cancel   -> stop the specified thread
    pthread_kill     -> send a signal to a specified thread
    pthread_detach   -> makes a thread unjoinable (it cleans up after itself)


Mutexes
    pthread_mutex_t   -> abstract struct definition
        -> we don't know what is in it
        -> must never make a copy

    pthread_mutex_init     -> sets up a mutex object (must call before using)
    pthread_mutex_destroy  -> tears down a mutex object (should call when done using)

    pthread_mutex_lock    -> begin critical section
        -> obtain lock, or wait until lock becomes available
    pthread_mutex_unlock  -> end critical section
        -> release lock

    Each mutex object represents a different lock
        only one thread can hold the lock at a time

Condition variables
    pthread_cond_t  -> abstract struct
        -> generally associated with a specific mutex (used with wait)

    pthread_cond_init     -> set up condition variable
    pthread_cond_destroy  -> tear down condition variable

    pthread_cond_wait(cond, mut)
        -> releases lock, waits until the condition is signaled, reacquires lock

    pthread_cond_signal
        -> wakes up one thread waiting for the condition
        -> if no threads are waiting, then nothing happens
            -> possible wait condition if we call signal outside of a critical section

    pthread_cond_broadcast
        -> wakes up every thread waiting for the condition



Queue example
    mutex lock        <- ensure exclusive access to queue
    cond read_ready   <- wait until queue is non-empty
    cond write_ready  <- wait until queue is non-full

insert()   - add item, possibly waiting until queue is non-full
remove()   - remove item, possibly waiting until queue is non-empty

Extending API

close()  - marks the queue as "finished"
    - nothing else can be inserted
    - remove should fail once the queue is empty

things to consider
    - what if we close the queue when threads are waiting to read?
    - what if we close the queue when threads are waiting to write?

    -> in both cases, we want to wake up the waiting threads and have them fail
        -> if a thread is waiting to dequeue when we close the queue, the queue must
            be empty, and the dequeue can never succeed
        -> if a thread is waiting to enqueue when we close the queue, the enqueue
            will never succeed

changes we would need to make:
    both insert and remove need to check whether the queue has been closed
        insert waits as long as the queue is open and full
        remove waits as long as the queue is open and empty

    queue_close() function needs to wake up all threads waiting to insert/remove
        -> pthread_cond_broadcast()

>> See queue2.c in Resources->Sample Code
```

```
CS 214 / 2021-04-05
===================

Recap:
- a thread is an execution context that belongs to some process
    - each thread has its own stack, current instruction & other attributes
        (such as signal mask)
    - every process has at least one thread
    - a process with more than one thread is "multithreaded"
- pthread_create allows us to execute a function in a new thread
    - the thread function takes one argument (void *) and returns void *
    - use of void * allows us to pass any value we want to a thread
        -> but the compiler won't catch type errors
        -> if you pass the wrong argument to a thread, you won't find out
            until the program runs
    - pthread_create returns immediately
        - gives us thread ID of new thread (written to pointer we provide)
        - returns 0 for success or an error number
    - pthread_join allows us to get the return value from a specified thread
        - if the thread is already finished, returns immediately
            - otherwise blocks until thread completes
            - similar to wait
        - we specify the thread to wait for
        - any thread can join any other thread
            - e.g., a thread can join its own parent
            - two threads should not call join with the same thread
    - every thread must either be joined or detached
        - joining cleans up a thread & obtains its return value
        - a detached thread cleans up after itself; return value is discarded
        - detached threads are useful if we don't need to know when a task is
            finished
        - for now, just use join
        - a joinable thread that has ended is a zombie
            - there is no reason to ever create a zombie thread

Why use threads?
    - take advantage of multiple processors
    - one thread can run when another is waiting (e.g., for IO)


int  // 0 for success, otherwise error
pthread_create(
    pthread_t *tid,  // location to write the new thread's ID
    pthread_attr_t *attr, // specify features for thread (or use NULL for default)
    void *(*function)(void *),  // function for thread to execute
    void *args   // argument to pass to the thread function
)

int err;
err = pthread_create(....);
if (err != 0) {
    errno = err;  // store error number so we can use perror
    perror("pthread_create");
    abort();  // or exit()
}

    // alternative to setting errno & using perror
    fprintf(stderr, "pthread_create: %s\n", strerror(err));

// error conditions are rare, but may indicate a program bug; always check!
// if you don't want to write the check every time, make a macro


int  // 0 for success, non-zero for error
pthread_join(
    pthread_t tid, // thread that we want to wait for/collect return value from
    void **ret  // location to write the return value to (or NULL to discard value)
)

Common pattern for divide-and-conquer breaking a large task into subtasks
    start a few threads
        use argument to have each thread do a separate part of the work
    then join all the threads
        doesn't matter which order we join in, as long as we eventually join all the
            worker threads


Note: we can only pass one argument to a thread
    but it can be anything, including a pointer to a struct

    struct thread_args {
        int *data;
        int length;
    };

    ... {

        struct thread_args arg = { my_array, my_array_len };

        pthread_create(&tid, NULL, array_handler, (void *) &arg);
            // passing the address of a local variable is only safe if this function
            // joins the thread before it exits & does not modify the variable

        // do other work

        pthread_join(tid, NULL);  // wait for thread to finish
    }


    ... {
        struct thread_args *arg = malloc(sizeof(struct thread_args));
        .. // initialize *arg


        pthread_create(&tid, NULL, array_handler, (void *) arg);
            // argument is on the heap, so we can return before joining
            // we should decide which thread is responsible for freeing the argument
            // (usually the child thread will deallocate its argument)

    }

coordination
------------

mutex gives us mutual exclusion for "critical sections"
    a critical section is a chunk of code that at most one thread can execute at a time

lock gives us exclusive access
    when one thread has the lock, no other thread can acquire it
        - threads wait (block) until the lock becomes available
    call at start of critical section
unlock releases exclusive access
    call at end of critical section
    if any threads are waiting, one will wake up and acquire the lock


e.g., global variable with lock

pthread_mutex_t balance_lock = PTHREAD_MUTEX_INITIALIZER;
int balance;


{

    pthread_mutex_lock(&balance_lock);     // start critical section
        // we are the only thread that has balance_lock
    balance += deposit;
    pthread_mutex_unlock(&balance_lock);   // end critical section
        // now other threads can acquire balance_lock
}

    // if we only read/write balance in critical sections, then we avoid
    // some concurrency bugs (things will behave in a predictable way)

A mutex must be initialized exactly once, before you lock

    int  // 0 for success, non-zero for failure
    pthread_mutex_init(
        pthread_mutex_t *mut,       // address of mutex object
        pthread_mutex_attr_t *attr  // set additional attributes (or use NULL)
    );

    int  // 0 for success, non-zero for failure
    pthread_mutex_destroy(
        pthread_mutex_t *mut        // address of mutex object
    );



    int // 0 for success, non-zero for failure
    pthread_mutex_lock(
        pthread_mutex_t *mut   // lock we want to acquire
    );

    int // 0 for success, non-zero for failure
    pthread_mutex_unlock(
        pthread_mutex_t *mut   // lock we want to release
    );


Only the thread that holds the lock may unlock!
    - by default, pthread_mutex_unlock does not check whether it is being called by
        the correct thread
    - think of lock/unlock like braces
        - should not hold the lock for too long
        - make it obvious that you only unlock after lock succeeds

Should check the return value of lock and unlock for errors
    - very rare, but could indicate a program error
    - another situation where a macro might be handy

    #define lock(X) \
        do { \
            int err = pthread_mutex_lock(X); \
            if (err) { \
                errno = err; \
                perror("lock"); \
                abort(); \
            } \
        } while (0) \



    lock(&balance_lock);   // less typing, and checks for errors

Locks may not be sufficient for our purposes

- Consider a synchronized queue (safe to use with multiple threads)

    enqueue  - add item to queue, or block if queue is full
    dequeue  - remove item from queue, or block until something becomes available


struct queue {
    int data[QUEUESIZE];
    unsigned head;  // index of first item in queue
    unsigned count;  // number of items in queue
    pthread_mutex_t lock;
};

int queue_init(struct queue *Q)
{
    Q->head = 0;
    Q->count = 0;
    return pthread_mutex_init(Q->lock, NULL);
}

int queue_add(struct queue *Q, int item)
{
    pthread_mutex_lock(&Q->lock); // make sure no one else touches Q until we're done

    while (Q->count == QUEUESIZE) {
        // wait for another thread to dequeue
        pthread_mutex_unlock(&Q->lock);
        sleep(1);  // but we don't know how long to wait
        pthread_mutex_lock(&Q->lock);
    }

    unsigned index = Q->head + Q->count;
    if (index >= QUEUESIZE) index -= QUEUESIZE;

    Q->data[index] = item;
    ++Q->count;

    pthread_mutex_unlock(&Q->lock); // now we're done

    return 0;
}



solution: condition variables
    gives us a way to wait for a condition to become true
        -> threads can wait for the condition
        -> other threads can indicate that the condition has become true
    (we actually wait for the go-ahead from other threads)


pthread_cont_t  -  type of a condition variable




struct queue {
    int data[QUEUESIZE];
    unsigned head;  // index of first item in queue
    unsigned count;  // number of items in queue
    pthread_mutex_t lock;
    pthread_cond_t read_ready;  // wait for count > 0
    pthread_cond_t write_ready; // wait for count < QUEUESIZE
};

int queue_init(struct queue *Q)
{
    Q->head = 0;
    Q->count = 0;
    pthread_mutex_init(&Q->lock, NULL);
    pthread_cond_init(&Q->read_ready, NULL);
    pthread_cond_init(&Q->write_ready, NULL);

    return err;  // obtained from the init functions (code omitted)
}




int queue_add(struct queue *Q, int item)
{
    pthread_mutex_lock(&Q->lock); // make sure no one else touches Q until we're done

    while (Q->count == QUEUESIZE) {
        // wait for another thread to dequeue
        pthread_cond_wait(&Q->write_ready, &Q->lock);
            // release lock & wait for a thread to signal write_ready
    }

        // at this point, we hold the lock & Q->count < QUEUESIZE

    unsigned index = Q->head + Q->count;
    if (index >= QUEUESIZE) index -= QUEUESIZE;

    Q->data[index] = item;
    ++Q->count;

    pthread_mutex_unlock(&Q->lock); // now we're done
    pthread_cond_signal(&Q->read_ready); // wake up a thread waiting to read (if any)

    return 0;
}

int queue_remove(struct queue *Q, int *item)
{
    pthread_mutex_lock(&Q->lock);

    while (Q->count == 0) {
        pthread_condition_wait(&Q->read_ready, &Q->lock);
    }

        // now we have exclusive access and queue is non-empty

    *item = Q->data[Q->head];  // write value at head to pointer
    --Q->count;
    ++Q->head;
    if (Q->head == QUEUESIZE) Q->head = 0;

    pthread_mutex_unlock(&Q->lock);

    pthread_cond_signal(&Q->write_ready);
}


conditions are always associated with a lock

pthread_condition_wait
    releases the lock
    suspends thread until the condition variable is signaled
    reacquires the lock

pthread_condition_signal
    wakes up one thread waiting for the condition variable
    - does nothing if no threads are waiting


    int  // 0 for success, non-zero for error
    pthread_cond_wait(
        pthread_cond_t *cond,  // condition variable to wait for
        pthread_mutex_t *mut   // lock to release/reacquire (must be held)
    );


    int
    pthread_cond_signal(
        pthread_cond_t *cond   // condition variable to signal
    );


>> See also queue.c in Resources->Sample Code
```