```
CS 214 / 2021-03-31
===================

Asst 3: wcat

- Do you need to worry about error messages messing up paragraph spacing?
    No. Text output goes to stdout, error messages go to stderr
    Only stdout needs to respect the paragraph spacing rules
    If the user cares about keeping error messages separate from output, they
        can use file redirection

        ./ww 80 *.txt > output.txt
        ./ww 80 *.txt 2> errors.txt

- How can you test wcat without a working ww?
    All we need is a "mock" or a "shim"
        -> something simple that replaces a needed component for testing purposes
    What are the responsibilities of wcat?
        - call ww with the appropriate arguments
        - insert newlines where appropriate
        - for enhancement 3: set up the pipe, detect empty output from ww

    For testing purposes, all we need is a program that prints a single line of text
        - maybe have some arguments that force an error (return EXIT_FAILURE)
        - for Enh. 3, have a way to force empty output


    int main(int argc, char **argv)
    {
        if (strcmp(argv[2], "err") == 0) {
            return EXIT_FAILURE;
        } else if (strcmp(argv[2], "empty") == 0) {
            return EXIT_SUCCESS;
        }

        printf("Fake ww called with %s %s\n", argv[1], argv[2]);

        return EXIT_SUCCESS;
    }


- in general, only print a newline after a non-empty output from ww

    Problem scenario: ./wcat 20 non_empty empty

        wcat will print extra newline after calling ww 20 non_empty
        but then output will end (because ww 20 empty has no output)
        -> output will end with a blank line
        -> this is okay, but still possible to avoid
            -> wait until the first call to read returns before printing the newline

-------

Clarification for wait and wstatus

    int wstatus;
    pid_t finished_child = wait(&wstatus);
        // finished_child is pid of child that terminated (or -1)
        // wstatus contains exit status information

    // we can use macros to find out how the child process exited
    // - did it exit normally, and/or did it abort after a signal?
    // - is it just stopped?
    // - if it exited normally, what was its exit status
    // - if it aborted, what signal caused it?

    if (WIFEXITED(wstatus) && WEXITSTATUS(wstatus) == EXIT_SUCCESS) {
        // child process is finished with success
    }

// Technically, WEXITSTATUS() is only useful when WIFEXITED() is true
// Also, you can use WIFSIGNALED() to test whether program exited due to a signal
// (But we can ignore that for Asst. 3)


-------

Returning to multithreading:


What do we need to do to use Pthreads?

    In our source code
        #include <pthread.h>

    For GCC, add -pthread to command line when compiling
        - this tells GCC to link against the PThread library
        - this may tell the compiler to use threaded variants of some syntax/functions

How can we coordinate multiple threads?

- For simple programs, we may not need to coordinate
    - each thread works with its own data
    - arguments to the thread / responses from the thread handled with pthread_create
        and pthread_join

- We can use some facilities for message passing, such as pipes
    - e.g., one thread can use a pipe to send a stream of bytes to another thread

write and read are thread-safe and atomic
    - when writing to/reading from a file, no other thread will be able to use the
        file until my call is complete
    - if two threads try to write at the same time, one will wait until the other is
        finished
    - note that only single calls are atomic
        - if I call write twice, another thread might get a write in between

All the FILE* functions are also atomic
    I can call printf from multiple threads without problems
        E.g., each call to printf will complete before the next one can start


But these may not be sufficient for all purposes
    We may want more coordination than simply message passing
    Also, someone had to write printf and write to be thread-safe, but how?


Basic idea: mutual exclusion
    - mutual exclusion is a way to ensure that at most one thread has access to a
        resource at a time
    - this is the basis for all coordination between threads


int bank_balance = 1000;  // global variable

thread 1:
    bank_balance += 100;

thread 2:
    bank_balance -= 50;


What is value of bank_balance after both threads run?

    It should be 1050
    It might be 1100
    It might be 950

Problem: the threads used non-atomic operations
    atomic operations are either finished or have not started
    non-atomic operations have multiple steps
        -> so other threads may run in between steps


    bank_balance += 100 is actually three steps
        1. read value of bank_balance
        2. add 100 to value
        3. write new value to bank_balance

Thread 1                Thread 2
Read bank_balance (1000)
add 100 (1100)
                        Read bank_balance (1000)
Write bank_balance (1100)
                        subtract 50 (950)
                        Write bank_balance (950)


Problem arises because
    1. increment/decrement are non-atomic operations
    2. both threads could access bank_balance simultaneously

This is an example of a data race
    -> the output that we get depends on which thread finishes first


To avoid problems, we must make the operation atomic or enforce mutual exclusion

Problem:
    - we can't create atomic operations in software
        -> require hardware support to enforce atomic nature
    - we can't enforce mutual exclusion without some atomic instructions


Solution:
    CPU designs include one or more atomic operations that can be used to build
        mutual exclusion tools (e.g., locks)

Test-and-set
    set a memory location and return the previous value

    We can use test-and-set to build a lock

        int lock; // global variable

        void lock() {
            int prev = test_and_set(lock, 1);  // example; not a real function
                // sets lock to 1
                // prev has previous value of lock

            // if it was already locked, we need to wait until someone else unlocks
            while (prev == 1) {
                prev = test_and_set(lock, 1);
            }

            // once we get here, we know that we closed the lock

        }

            // only safe to call this if we hold the lock
        void unlock() {
            lock = 0;
        }

    This is what is called a "spin lock"
        - we have a loop that does not end until we are the thread that has locked
            the lock

    Safe version of previous example

    Thread 1:
        lock();
        bank_balance += 100;
        unlock();

    Thread 2:
        lock();
        bank_balance -= 50;
        unlock();

    This enforces mutual exclusion of bank_balance
        thread 2 cannot interfere with thread 1, because lock() will not return
            until after thread 1 calls unlock

    This lock/unlock pattern is called a "mutex" (short for "mutual exclusion")

Fetch-and-add
    Similar idea, except we add the argument to the value
    Still atomic

Compare-and-swap
    The most generally useful primitive
        we tell it the value we expect to see, and the new value
        if the value is correct, it is changed; otherwise it is left alone

    We can implement test-and-set and fetch-and-add using a finite number of
        compare-and-swaps (the reverse is not true)


These are part of the CPU's instruction set
    -> the people who write the C standard libary and Pthread library use these

We use the functions provided by pthread.h


Use mutex to create a lock


    pthread_mutex_t lock;   // a struct or something (abstract)

    pthread_mutex_init(&lock, NULL);  // initialize lock
        // must be called exactly once before the lock can be used

    pthread_mutex_lock(&lock);  // acquire lock; block until lock becomes available

    pthread_mutex_unlock(&lock);  // release lock


    The example from before, using pthread functions

    pthread_mutex_t balance_lock = PTHREAD_MUTEX_INITIALIZER;
        // global variable; already initialized


    Thread 1:
        pthread_mutex_lock(&balance_lock);
        bank_balance += 100;
        pthread_mutex_unlock(&balance_lock);

    Thread 2:
        pthread_mutex_lock(&balance_lock);
        bank_balance -= 50;
        pthread_mutex_unlock(&balance_lock);

The rule with a mutex, is that at most one thread can acquire the lock at a time
If a thread tries to lock the mutex, they have to wait

    Note that mutex only guarantees mutual exclusion of the lock itself
    Nothing stops me from writing code that accesses the resource without locking first

The second rule with a mutex is that only the thread that locked the lock can
    unlock it


int pthread_mutex_init(pthread_mutex_t *mut, pthread_mutex_attr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mut);
int pthread_mutex_unlock(pthread_mutex_t *mut);
int pthread_mutex_destroy(pthread_mutex_t *mut);

    We always pass a pointer to the mutex object
        -> duplicating the object has undefined result
        -> we don't ever assign to a pthread_mutex_t
    These all return 0 on success, non-0 on failure


You can put your mutex object in global space, the heap, or even the stack (if you
    are careful)

    -> pthread_mutex_init does not allocate space

    pthread_mutex_t *lock;
    pthread_mutex_init(lock, NULL);   // undefined behavior! bad pointer! AAAAA!

We can use mutex to enforce mutual exclusion, but it only works if we use it correctly
    -> mutex doesn't protect you from badly written or malicious code



Initialize once
lock before entering the mutually exclusive part of your code
unlock after exiting the mutually exclusive part of your code
Destroy when you no longer need the lock

Next time:
    - using mutex to create synchronized data structures
    - additional synchronization tools
```