```
CS214 / 2021-04-12
==================

Project II

    -ssome_text

        suffix is "some_text"
        suffix may start with a dot (.), but is not required to
        suffix may be empty

        -s.txt  suffix is ".txt"
        -s      suffix is ""
        -sbak   suffix is "bak"

Math and log2
- use double for floating-point
    - there is never a reason to use float, unless you are conserving memory

- log2() is provided by math.h
    #include <math.h>

    Note that the math functions are not linked by default
        recall: linking occurs after compiling
            compiled libraries are "linked" into a single executable
    The math functions are in libm
        You need to tell GCC to link libm
            -l<library name>
            -lm   - link libm ("lib" is implicit)
        In the current version of GCC, -lm has to occur after your program

    one step:
        gcc -Wall...  compare.c -lm

    separate compilation
        gcc -Wall...  -c compare.c               # compiling
        gcc -Wall... compare.o -lm -o compare    # linking

        recall: -c means "compile, but do not link"

    -pthread has two roles
        - link program with pthread library
        - may also change how your program is compiled
            compiler can simplify some things if it knows your program is
                single-threaded



Sketch of program

main thread:
    read options (determine number of threads to create)
    create/initialize queues
    start requested number of file and directory threads
    add directories listed in arguments to directory queue
    add files listed in arguments to file queue
    join file and directory threads
    ... continue to phase 2


directory thread
    loop
        read directory name from queue
        open directory
            add entries to file or directory queues
    repeat until directory queue is empty and all directory threads are waiting

file thread
    loop
        read file name from file queue
        open file
        count all words in file
        compute WFD
        add to WFD repository
    repeat until the queue is empty and the directory threads have stopped


How can you tell when all the directory threads are finished?
-> easy method: keep track of how many dir. threads are "active"
    that is, not currently waiting to dequeue

    dequeue
        if queue is empty
            reduce number of active threads
            if no active threads:
                % we are the last thread to finish
                wake up all the threads (e.g., pthread_cond_broadcast)
                optional: close the file queue
                terminate
            wait until number of active threads is 0 or queue is not empty
            if no active threads
                terminate
        do actual dequeue (if we get here, queue is not empty)

When does a thread terminate?
    - if the thread function returns
    - if the thread calls pthread_exit()

Note:
    you are not writing a general-purpose queueing library
    it is perfectly fine to customize your design to work with this project


Bounded vs unbounded (synchronized) queues
- a synchronized queue is just a queue that can safely be used by multiple threads
    that is, a thread-safe queue
    - we probably use a mutex to enforce thread safety
        - only one thread can read/modify the data structure at a time
        - impossible to get inconsistent results due to nondeterminism

- typically, a synchronized queue will block if you try to dequeue from an empty queue
    - e.g., we wait until another thread adds something to the queue
    - condition variables are good for this

- a bounded queue has a maximum size
    - i.e., it can be full
    - we wait if we try to enqueue into a full queue until another thread dequeues

"Queue" is a an abstract type
    - it specifies an interface (enqueue, dequeue)
    - many possible implementations (linked list, array, tree, etc.)


In Project II, I recommend a bounded queue for the files and an unbounded queue
    for the directories

- bounded file queue prevents queue from growing too large
    - slows down the directory threads if they go too fast for the file threads
    - but: make sure the main thread starts the file threads before it adds the
        arguments to the file queue
        (if there are too many file arguments to fit in the file queue, the main
        thread will block; if the file threads aren't already running, we deadlock)

- directory thread easiest if unbounded
    if directory queue were bounded
        queue could fill up
        if all the threads are currently reading directories, then they all block
        -> no thread will be available to dequeue
        deadlock!
    -> possible to avoid this deadlock, but doing so is more complicated


Deadlock
--------

Deadlock occurs when threads are blocked and cannot get unblocked
    -> program is stuck and cannot make progress

Simple example: two resources X and Y and two threads A and B

thread A:
    lock X
    lock Y
    do something
    unlock Y
    unlock X

thread B:
    lock Y
    lock X
    do something
    unlock X
    unlock Y

Danger scenario (not guaranteed to happen, but not guaranteed not to happen)

    A           B
    lock X
                lock Y
    lock Y
    (blocks)
                lock X
                (blocks)

Both threads are blocked
    A is waiting for B to release Y
    B is waiting for A to release X

    -> neither thread can advance until the other finishes
    -> thus, neither thread can release the resource

What is required to have deadlock?

1. mutual exclusion
    - it must be possible for a single thread to hold a resource and prevent
      other threads from obtaining it until they are finished
2. hold and wait
    - must be possible to block while holding exclusive access to something
    - e.g., any time we call printf while we hold a lock
3. no preemption
    - other threads cannot force a thread to give up exclusive access
4. circular wait
    - two or more threads are waiting for each other to give up a resource
        A waiting for B, B waiting for A
        A waiting for B, B waiting for C, C waiting for A
        etc.

Eliminating any of these is sufficient to prevent deadlock
- we can't really give up 1 or 2
- eliminating 3 is hard (what happens to the thread that got preempted?)
- most solutions focus on avoiding circular wait


A partial strategy
    always acquire resources in a fixed order
        e.g., if we need both X and Y, always get them in the same order
        -> prevents the simple scenario described above
    avoid holding multiple locks whenever possible
    -> have a priority ordering of locks
        e.g., it's okay to get exclusive access to stdout if you hold lock X,
            but not vice versa

In general, detecting deadlock in code is hard
- protect yourself by avoiding complicated interactions between threads as much
    as you can
- if deadlock is possible, it is a bug and should be fixed

-> no scheme can find all deadlocks
-> write carefully and think about possible scenarios

How can we tell if a program is deadlocked?
- no general method
    - how can we tell whether the program is blocked forever or for a long time?
- if you understand how your program works, you can get insight into what is happening

When working on project II, ask yourself what possible ways can your threads run?
- a context switch may happen at any time
- your program should be okay even if the switch happens at the worst possible time


"Monitors" are data structures/interfaces that can avoid or detect deadlock (relating
to their own use)
    - we will talk more about these later


Barriers
--------

A way of enforcing a different kind of coordination between threads

A barrier has a specific number
    threads can wait at a barrier
    once the specified number of threads are waiting, all the threads resume

https://man7.org/linux/man-pages/man3/pthread_barrier_init.3p.html
https://man7.org/linux/man-pages/man3/pthread_barrier_wait.3p.html

pthread_barrier_t

int   // 0 for success, non-zero for error
pthread_barrier_init(
    pthread_barrier_t *barrier,    // barrier to initialize
    pthread_barrierattr_t *attr,  // configuration options, or NULL for defaults
    unsigned count
)

int  // 0 for success, non-zero for error
pthread_barrier_destroy(
    pthread_barrier_t *barrier
)

int   // 0 or PTHREAD_BARRIER_SERIAL_THREAD for success, anything else for error
pthread_barrier_wait(
    pthread_barrier_t *barrier   // barrier to wait at
)

    - exactly one thread gets PTHREAD_BARRIER_SERIAL_THREAD
        - unspecified which thread gets it
    - every other thread gets 0



Another way to create a "rendezvous"
    i.e., coordinate threads to make sure they reach a point before they continue

Example
    - we have N worker threads
    - each worker thread has to set up some data before it can start working
    - all the threads must be ready before any of them can start

We could do this with a mutex and condition variable
    - keep track of threads that have finished setting up
    - last thread broadcasts to all the other threads

But this is what barriers are for
    - main thread creates barrier for N threads
        pthread_barrier_init(&worker_bar, NULL, N);
        for (i = 0; i < N; ++i)
            pthread_create(&tid[i], NULL, worker, &arg[i]);



    void *worker(void *arg)
    {
        //  do set up stuff

        err = pthread_barrier_wait(&worker_bar);
        if (err) ...

        // do coordinated work
        // we won't get past the barrier until all the worker threads have finished
        // setting up
    }


See "barrier.c" in Resources->Sample Code
```