```
CS214 / 2020-04-14
==================

struct comp_result {
    char *file1, *file2;
    unsigned tokens;       // word count of file 1 + file 2
    double distance;       // JSD between file 1 and file 2
};

{
    ...
    unsigned num_files = length_of_wfd_repository(...);

    unsigned comparisons = num_files * (num_files - 1) / 2;

    struct comp_result *results = malloc(comparisons * sizeof(struct comp_result));

    i = 0;
    for (f1 = 0; f1 < num_files; f1++) {
        for (f2 = f1 + 1; f2 < num_files; f2++) {
            results[i].file1 = file_name(f1);
            results[i].file2 = file_name(f2);
            results[i].tokens = file_tokens(f1) + file_tokens(f2);
            results[i].distance = compute_jsd(f1, f2);
            ++i;
        }
    }

    qsort(results, comparisons, sizeof(struct comp_result), sort_comps);
        // elsewhere: int sort_comps(void *r1, void *r2);

    for (i = 0; i < comparisons; ++i) {
        print_result(results[i]);
    }
}

Simple scheme for dividing work among N analysis threads

thread 0 does comparisons 0, N, 2N, 3N, ...
thread 1 does comparisons 1, N+1, 2N+1, 3N+1, ...
thread 2 does comparisons 2, N+2, 2N+2, ...
thread N-1 does comparisons N-1, 3N-1, 4N-1, ...




recap: synchronization tools
    create/join
        - start threads
        - wait for a specific thread to finish
    mutex (MUTual EXclusion)
        - exclusive access (e.g., to a resource)
        - critical section: the protected code between the lock and unlock
    condition variables
        - suspend thread until some condition is satisfied
        - analogous to a loop that repeatedly checks the condition
            - pro: doesn't waste time checking constantly
            - pro: don't need to guess how long until condition will be true
            - con: thread doesn't wake up automatically; some other thread
                must signal
    barrier
        - make some number of threads wait until all threads have reached some
            point in their work
            - the number of threads is specified the barrier is created/initialized
        - prevent some threads from getting ahead of other threads
        - e.g., all threads must finish one phase before any start the next




Alternative design for Project II
* directory and file threads become analysis threads in phase 2
* use barrier to ensure phase 2 follows phase 1
* do not actually implement this for the assignment

Main thread          Directory threads          File threads
read options
start threads        repeat                     repeat
enqueue arguments        dequeue dir                dequeue file
                         enqueue dir entries        add WFC to repository
                     until all dirs read        until all files read

barrier              barrier                    barrier

allocate JSD table
set up work queue

barrier              barrier                    barrier

                     repeat                     repeat
                         get next comparison        get next comparison
                         add JSD to table           add JSD to table
                     until work done            until work done

join all threads     <-                         <-
sort table
print table



Project II notes
* Why organize our code this way?
- there is a maximum number of files (including directories) you can have open at once
- the model in the assignment guarantees a fixed number of open files at once
    -> we never have to hit the limit
- allows user to tune performance (e.g., choose degree of concurrency)



Semaphores
----------

optional reading:

        The Little Book of Semaphores
        https://greenteapress.com/wp/semaphores/

The "original" synchronization mechanism
    - all other mechanisms can be made using one or more semaphores
    - very general, so compiler/library/runtime cannot optimize as much

Two basic operations
- several names, none of which are completely intuitive
- original names: P and V
        (may be derived from Dutch railway terminology)

Idea: we have an integer
- threads can (safely) increase it or decrease it
- a thread that tries to decrease it below zero blocks until some other thread increases it

Operations
    - create / initialize
        set initial number
    - P / wait / decrease
        reduce integer, or wait until it becomes non-negative
    - V / post / increase
        increase integer

We can use these as a mutex


    pthread_mutex_init    -  sem_init 1
    pthread_mutex_lock    -  sem_wait   (reduce by 1)
    pthread_mutex_unlock  -  sem_post   (increase by 1)



sem X initially 1

    thread A                thread B
    wait X
        X <- 0
        does not block

                            wait X
                            |   X already 0
                            |   blocks
                                                     |
    ...                                              |
                                                     |
    post X                                           |
        X <- 1              -> now wait can finish
                                X <- 0


                            ...

                            post X
                                X <- 1

"Binary" semaphore: value is always 0 or 1
General semaphore: value can be any non-negative integer


We can use a semaphore to simulate a mutex
    -> mutex is more restricted
        -> only the thread that locked can unlock
    -> semaphore is more general
        -> any thread can post at any time



Turnstile pattern  -> wait followed by post

    wait X
    post X
        <- we pass through this immediately if X > 0


    If some thread waits (reducing X to 0), then no other thread can pass through
        the semaphore



What about general semaphores?

- we can think of these as representing the amount of resources available
    - wait claims a resource
    - post releases a resource



bounded queue can be done with 3 semaphores, or 2 semaphores + 1 mutex

Queue:
    int array[QSIZE];
    sem_t open;             // spaces available to write
    sem_t used;             // items available to read
        // open + used == QSIZE
    pthread_mutex_t mut;   // gives us mutual exclusion (could use another semaphore)
    unsigned head;
    unsigned count;

enqueue

    sem_wait(open);    // claim one open space, or wait until one is available

    lock(mut);
        i = (head + count) % QSIZE;
        array[i] = item;
    unlock(mut);

    sem_post(used);   // indicate one more item is ready to be dequeued



dequeue

    sem_wait(used);  // claim one item in queue, or wait until one is available

    lock(mut);
        ret = array[head];
        head = (head + 1) % QSIZE;
    unlock(mut);

    sem_post(open);  // indicate one more open space is available



unbounded stack

    struct node *head;  // initially NULL
    sem_t available;   // number of items available (initially 0)
    sem_t lock;         // mutex (initially 1) (could use pthread_mutex_t instead)

push
    struct node *new = malloc(sizeof(struct node));

    sem_wait(lock);
        new->next = head;
        head = new;
    sem_post(lock);

    sem_post(available);   // indicate that another item is available

pop
    sem_wait(available);  // claim an item or wait until one is available

    sem_wait(lock);
        head = head->next;
            // NOTE: we don't need to check whether head is NULL; why?
    sem_post(lock);




sem_t   // semaphore type (a struct; do not duplicate)

int  // 0 for success, -1 (and set errno) for failure
sem_init(
    sem_t *sem,
    int pshared,           // whether or not it is shared between processes
                           // always 0 unless you are using shared memory regions
    unsigned int value     // initial value of semaphore
);

int  // 0 for success, -1 (and set errno) for failure
sem_destroy(sem_t *sem);

int  //   0 for success, -1 (and set errno) for failure
sem_wait(sem_t *sem);

int  // 0 for success, -1 (and set errno) for failure
sem_post(sem_t *sem);



// named semaphores exist outside an individual process
//    essentially files that contain an integer, with atomic increment/decrement
// not much call for these, but useful to coordinate multiple processes
// man 7 sem_overview explains what sorts of names you may use

// open an existing named (persistent, multi-process) semaphore
sem_t *
sem_open(
    char *path,  // essentially a file name
    int oflag    // same as flags to open(), excluding O_CREAT
);


// open or create named semaphore
sem_t *
sem_open(
    char *path,         // name of semaphore
    int oflag,          // should include O_CREAT, optionally O_EXCL
    mode_t mode,        // permissions (if creating new)
    unsigned int value  // initial value (if creating new)
);

int  // 0 for success, -1 (and set errno) for failure
sem_close(sem_t *sem);
```