```
ww
--

We can distinguish user errors from program errors (in wcat)
    - user errors
        - file does not exist or has wrong permissions
        - page width non-positive
        - file has word longer than page width
    - program errors / error conditions
        - out of memory
        - fork fails
        - exec fails
        - close fails

wcat should check the exit status of ww
if ww returns non-zero, then wcat will continue and return exit failure

    ww is responsible for detecting most user errors
    wcat should check if it is given a directory name

For bad arguments (directories, missing files), wcat prints a message and continues
If ww exit status EXIT_FAILURE, wcat continues (assume ww already printed a message)
If fork or wait fail, wcat should print a message and exit (don't continue)

wcat returns EXIT_FAILURE if any error condition occurs

Output to standard error:
    write(2, buf, bytecount)
    fprintf(stderr, ...)
    fputs(message, stderr)


General note:
    wcat is concurrent, in that two programs run at once (wcat and ww)
    wcat does not try to run ww itself concurrently
        we do not want to fork multiple simultaneous child processes
            -> this could result in interleaving of output (unless we do enhancement 3)


-------

We have discussed one method of multitasking:
    multiprocessing - running several processes simultaneously
        (either on separate processors or sharing time on a single processor)

    one limitation of this model is communication
        if two processes want to communicate, we can only do so by sending messages
            -> through the file system
            -> through sockets
            -> through shared pipes
            -> through signals (very limited!)
            -> parent can pass some data via arguments
            -> child can return some (limited!) data via exit status

    one advantage is safety
        separate processes have separate memory spaces
        can start and stop independently
        -> a bug in one process won't take other processes down

e.g., a server environment might start multiple processes
    child process does all the work
    parent process waits for the child process to crash, and restarts it
        -> "nanny" process

But sometimes we want more than this
    -> we might want to open/read from multiple files concurrently
        -> don't want to block the whole program while we wait
    -> we might have a complex computation that we need to perform (e.g., parse XML)
        -> we want our program to remain responsive
        -> we need a way to perform large tasks asynchronously


We want to be able to perform multiple tasks within a single process
    -> so we introduce threads and multithreading

a "thread" is an execution context with in process
    a single process with multiple threads is multithreaded

There are a few ways to do threads
    - user threads / "green" threads
        -> language run-time or library allows us to switch between different tasks
        -> no involving the OS
    - OS threads / kernel threads
        -> essentially multiple processes running in the same memory area
        -> each "thread" has its own process control block, processor context, stack
            -> all threads share process id

Threads behave like processes in some respects, but not all
    exec changes the program running for the entire process (all threads are halted)
    fork only duplicates the current thread
    signals can be received by any thread
        each thread has its own signal mask (which signals are blocked)
            -> signals sent to a process will be delivered to any thread that is not
                blocking that signal
            -> at most one thread will be delivered any particular signal
        signal handlers are shared among threads

    exit(), _exit(), and abort() terminate all threads in the process
    returning from main(), all threads are terminated (because it calls exit())

man 7 pthreads lists some attributes that are thread-specific and some that are
    shared by all threads


So: how can we use threads?
In this course, we will discuss the Posix threading interface ("pthread")

type
    pthread_t - thread ID (essentially an int)

We use pthread_create to start a new thread
    -> but we need to indicate what that thread should do
    -> instead of a fork-like model, we give pthread_create a specific function
        to execute

A "thread function" is a regular function

    void *my_thread_function(void *arg);

    - takes one argument, a pointer
    - returns a pointer

err = pthread_create(&tid, NULL, my_thread_function, argptr);
    <- calls my_thread_function "in the background" (asynchronously)
    <- pthread_create returns immediately

compare with
    p = my_thread_function(argptr)
        <- calls my_thread_function synchronously
        <- does not return until my_thread_function is finished


    int   // returns 0 if successful, or an error code (does not set errno)
    pthread_create(
        pthread_t *tid, // id of new thread will be written here
        pthread_attr_t *attrs,  // ptr to struct requesting features
            // or NULL to get the default attributes
        void *(*thread_function)(void *),
            // function that the thread will execute
        void *args  // argument to pass to thread_function
    );


Why void*?
    This is how we write generic functions in C
    Using void * means we can pass (a pointer to) any value as an argument
        and we can return (a pointer to) any value
    Downside: no type checking; compiler will trust that what we do makes sense

Why have an argument at all?
    We want to start multiple threads with the same function
        -> passing an argument lets us send specific data to each thread

    We can just use NULL if we have nothing useful to pass to the thread

pthread_t tid;
err = pthread_create(&tid, NULL, thread_fun, NULL);
if (err != 0) { errno = err; perror("pthread_create"); exit(EXIT_FAILURE); }

// at this point, the thread has started and tid contains its thread id


But what about the return value?
    the thread function returns a void pointer, but who gets it?

How do we know when the thread has finished doing its task?

    pthread_join collects the return value of a specified thread

    int   // returns 0 for success, error number on error
    pthread_join(
        pthread_t tid,   // id of the thread we want to join (wait for)
        void **retval   // where to write the void * that the thread returned
                        // or NULL to ignore the return value
    )


    pthread_t tid;
    struct arg_t *args = ....;
    struct retval_t *ret;

    pthread_create(&tid, NULL, background_task, args);

    // at this point, the thread has started

    // do other things

    pthread_join(tid, &ret);

    // at this point, the thread has stopped and ret points to its return value
    // note: we passed &ret so that join could change what ret points to


create and join are analogous to fork and wait

when we join a thread
    - join will block until the specified thread has ended
    - join returns immediately if the thread has already ended

If we do not join a thread, no one will collect its return value
    -> potential memory leak

-> As a rule, any thread we create must be joined
-> any thread can join a thread, not just its creator
    -> but only one thread should join

----
// very simple example of splitting a task into multiple threads
// note: code to check errors not included

struct arg {
    int length;
    double *data;
};


// thread function
void *compute_sum(void *argptr)
{
    struct arg *args = (struct arg *) argptr;  // assume argptr has correct type

    int i;
    double sum = 0.0;
    for (i = 0; i < args->length; i++) {
        sum += args->data[i];
    }

    // must return a pointer
    double *retval = malloc(sizeof(double));

    *retval = sum;

    return retval;
}

// driver function

#define THREADS 5

void compute_sums(double *array, int length)
{
    pthread_t tids[THREADS];   // hold thread ids
    struct arg args[THREADS];  // hold arguments
    double *retval, sum = 0.0;
    int i, start, end;

    // initialize all arguments
    // e.g., we could break an array into equal-sized chunks
    end = 0;
    for (i = 0; i < THREADS; i++) {
        start = end;
        end = length * (i + 1) / THREADS;
        args[i].length = end - start;
        args[i].data = &array[start];
    }

    // start all threads
    for (i = 0; i < THREADS; i++) {
        pthread_create(&tids[i], NULL, compute_sum, &args[i]);
    }

    // we now have THREADS+1 threads running

    // wait for all threads to finish
    for (i = 0; i < THREADS; i++) {
        pthread_join(tids[i], &retval);
            // (only) current thread waits until tid[i] has finished
            // retval will point to the sum
        sum += *retval;
        free(retval);
    }
}

Next time:
    -pthread option to compiler
    details about argument and return value safety
    coordination between threads
```