IMPORTANT NOTES for all questions on the exam: For coding questions, the majority of the points are typically focused on the correctness of the code. However, there may be deductions for code that is roundabout/awkward/inefficient when more appropriate alternatives exist. For any coding questions, your answers should compile cleanly and not have any memory leaks or errors. You may need to scroll vertically or horizontally to fully view blocks of code. Solutions that violate any specified restrictions may get partial credit. Style is secondary to correctness (e.g. there are no style deductions for using magic numbers). **There is 1 point per minute of the exam**.

1) Short Answer **39 Points/120 Total**

A) [4 points] Your friend suggests that it would be better for a file system to eliminate inodes as a separate structure and simply store all of the inode information for a file directly in the directory entry for the file. Give one advantage and one disadvantage of this new approach.

B) [4 points] Your friend has built a file system that uses write-ahead logging for crash recovery; it logs both metadata and file data. However, you notice that the system does not seem to recover properly after some crashes. In looking over the file system code, you discover that in some situations the system creates a log record of the form “append data D to the file with i-number I” (the log record contains an opcode indicating an append operation, plus the new data and the file’s i-number). How can this log record cause incorrect behavior after a crash?

C) [4 points] Your assign1 file system relied on direct indexing for small files and singly and doubly indirect indexing for large files. In the name of code uniformity, you could have just represented all files, large and small, using entirely doubly indirect indexing. Briefly describe the primary advantage (other than uniformity of implementation) and primary disadvantage of relying on just doubly indirect indexing for all file sizes.

D) [6 points] The block layer of a file system needs to keep track of which blocks are allocated and which ones aren't. One scheme sets aside a portion of the underlying device to store a bitmap, where a single bit notes that some block is free if it's 0, or allocated if it's 1. The 0th bit tracks the allocation status of the 0th block beyond the bitmap, the 1th bit tracks the allocation status of the 1th block, and so forth. Another scheme might thread all of the unallocated blocks into a linked list, where the first four bytes (i.e. the improvised next pointer) of each unallocated block store the block number of the next unallocated block in the free list, and so forth. The super block could store the block number of the first unallocated block in this list, and the last unallocated block could store 0 in its next pointer to signal the end of the free list. Briefly present one advantage of each approach.

E) [6 points] If block sizes are 1024 (or 2^10) bytes and inodes are 32 (or 2^5) bytes, what percentage of the storage device should be allocated for the inode table if we never want to run out of inodes? Your answer can be approximate, and the 50-words-or-less defense of your answer should include the necessary math. Assume that all files require at least one block of payload, and assume a minimum storage device size of 1 terabyte (or 2^40 bytes).

F) [6 points] The `rename` system call renames a file, moving it from one directory to another if necessary. It comes with the following prototype:

```
int rename(const char *ep, const char *np);
```

`ep` is short for "existing path", and we'll assume it's an absolute path to a valid file you have permission to rename. `np` (short for "new path", and also absolute) identifies where the file should be moved to and what new name it should assume. Any intermediate directories needed for the move are created. So, a call to

```
rename("/WWW/index.html","/archive/winter-2019/index-w19.html");
```

would remove `index.html` from `WWW` and move it to `archive/winter-2019`, creating `archive` and `winter-2019` if necessary, with the name of `index-w19.html`. The renaming works even if the file being moved is a directory.

Without worrying about error checking, describe how rename could be efficiently implemented in terms of your Unix v6 file system implementation.

G) [4 points] In a few sentences, explain why increased crash recovery capability means tradeoffs with performance, and give one specific example of such a tradeoff.

H) [5 points] Consider the following code:

```
void func(int& x, int val) {
    x += val;
    x *= 2;
}

int main(int argc, char *argv[]) {
    int x = 0;
    thread t1 = thread(func, ref(x), 10);
    thread t2 = thread(func, ref(x), 5);
    t1.join();
    t2.join();
    cout << "The value of x is " << x << endl;
}
```

Give 3 examples of possible outputs for this program.

2) Duet **25 Points/120 Total**

Leverage your `pipe`, `fork`, `dup2`, and `execvp` skills to implement `duet`, which has the following prototype:

```
static void duet(int incoming, char *one[], char *two[], int outgoing
);
```

`incoming` is a valid, read-oriented file descriptor, `outgoing` is a valid, write-oriented file descriptor, and `one` and `two` are well-formed, NULL-terminated argument vectors. `duet` launches two child processes, the first of which executes the program identified in `one`, the second of which executes the program identified in `two`.

The first process's standard input is rewired to draw bytes from `incoming`, and its standard output is rewired to feed bytes to the standard input of the second process, which itself directs its standard output to whatever resource is bound to `outgoing`. The function waits for the two processes (and only those two processes) to run to completion before returning. Write your implementation of `duet` in the space provided. You may assume that all system calls succeed, and that the executables identified by `one` and `two` always run to completion without crashing. You should close all unused file descriptors (including `incoming` and `outgoing` once you've leveraged their resources).

3) Concurrent Evaluation **27 Points/120 Total**

A colleague has a program where they have a vector of `Expression`s (some variable type they have made), and they want to compute the result of each of them - there is an `evaluate` method on them to do this. They can write this without using threads, like this:

```
static void evaluateAll(const vector<Expression>& expressions) {
    for (int i = 0; i < expressions.size(); i++) {
        expressions[i].evaluate();
    }
}
```

However, they know that each expression can be evaluated independently, so they think that writing this with threads can speed up the computation by having each expression concurrently evaluated in its own thread. They have the following scaffolding to do this, but need your help completing it:

```
typedef struct ThreadInfo {
  // Part A here
} ThreadInfo;

static void evaluateOne(Expression& exp, ThreadInfo& info) {
    // Part C here
}


static void evaluateAll(const vector<Expression> expressions) {
  ThreadInfo info;

  // Part B here

  for (size_t i = 0; i < expressions.size(); i++) {
    thread(evaluateOne, ref(expressions[i]), ref(info));
  }

  // Part D here
}
```

They have a struct to bundle together any fields needed by each thread (part A), know they need to initialize the fields in that struct (part B), then spawn off at thread for each expression to evaluate it (part C), and lastly wait on all the threads to finish (part D). However, your friend insists on **not using the `join()` method on threads** to wait for them, and asks you to implement this waiting in another way (they do not even store the spawned threads anywhere for later).

Implement the remainder of this code by completing the parts below. Your implementation should not have any busy waiting. You should not use the `atomic` class if you know it (if not, ignore this sentence).

A) [4 points] Fill in the definition of the `ThreadInfo` struct with any state you would like to pass by reference to all threads (e.g. `mutex` es, `condition_variable_any` s, other state, etc.) so that all threads have the needed shared variables. You may consider returning to this part as you implement later parts.

B) [3 points] Fill in blank B in the code above with one or more lines to initialize any needed fields in the struct `info`.

C) [11 points] Implement the **atomic** `evaluateOne` function that takes in the expression to evaluate, and the `ThreadInfo` struct, and calls `evaluate` on the expression. The function should take all actions to allow for the main thread to properly wait on it to finish (without calling `.join()`). This function should run **in parallel** as much as possible (hint to minimize one-thread-at-a-time operations).

D) [9 points] Fill in blank D with one or more lines to have the main thread wait for the spawned threads to finish, without calling `join` on each of the threads.

4) Multiprocessing **29 Points/120 Total**

A) [9 points] Consider the following C program and its execution. Assume all processes run to completion, all system and `printf` calls succeed, and that all calls to `printf` are atomic. Assume nothing about scheduling. List all possible outputs of this program:

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }

    if (counter > 0) printf("%d", counter);

    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }

    return 0;
}
```

B) [8 points] Recall the implementation of the `subprocess` function and test program we presented in lecture to illustrate how the `pipe` function worked:

```
subprocess_t subprocess(const char *command) {
    pipeline p(command);

    int fds[2];
    pipe(fds);
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        close(fds[1]);
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);
        execvp(p.commands[0].argv[0], p.commands[0].argv);
    }
    close(fds[0]);

    subprocess_t returnStruct;
    returnStruct.pid = pidOrZero;
    returnStruct.supplyfd = fds[1];
    return returnStruct;
}

int main(int argc, char *argv[]) {
    subprocess_t sp = subprocess("/usr/bin/sort");

    const char *words[] = {
        "felicity", "umbrage", "susurration", "halcyon",
        "pulchritude", "ablution", "somnolent", "indefatigable"
    };

    for (size_t i = 0; i < sizeof(words) / sizeof(words[0]); i++) {
        dprintf(sp.supplyfd, "%s\n", words[i]);
    }

    close(sp.supplyfd);
    waitpid(sp.pid, NULL, 0);
    return 0;
}
```

Explain why the test program would stall without printing anything if the implementation of `subprocess` acidentally omitted its three calls to `close`.

C) [12 points] The `thyme` program runs another program in a child process, and once the child process finishes, thyme publishes the number of seconds it took for the child process to run from start to finish. Assume, for instance, that I can invoke the make executable like this:

```
myth15> make fd-puzzle fork-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fd-puzzle.o fd-puzzle.c
gcc fd-puzzle.o -o fd-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fork-puzzle.o fork-puzzle.
gcc fork-puzzle.o -o fork-puzzle
```

I can do precisely the same thing using `thyme` to execute `make fd-puzzle fork-puzzle`, get the same output and generate the same compilation products, and also get the number of seconds it took to execute make using this:

```
myth15> thyme make fd-puzzle fork-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fd-puzzle.o fd-puzzle.c
gcc fd-puzzle.o -o fd-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fork-puzzle.o fork-puzzle.
gcc fork-puzzle.o -o fork-puzzle
Elapsed time: 0.103602930 sec
```

To compute timing information, assume there is a variable type called `struct timespec` that stores info about a single time. You could get the current time by calling `clock_gettime`, and it would fill in the struct at the specified location with info about the current time:

```
struct timespec time;
clock_gettime(CLOCK_REALTIME, &time);
```

Also assume there is a function `print_elapsed_time` that takes in pointers to two `struct timespec` parameters, calculates the difference between them, and prints it out in the format "Elapsed time: XXX sec":

```
void print_elapsed_time(struct timespec *start, struct timespec *finis
```

Implement the full `thyme.c` program. Your implementation should execute the program being timed, wait for it to finish, and then print how long it took. (You may not use `system`, `popen`, `subprocess`, or any other functions implemented in terms of `fork`, `execvp`, and so forth. You must explicitly call `fork`, `execvp`, etc. in the code you write.)