CS111 Practice Midterm Exam

This is a closed book, closed note, closed electronic device exam, except for one double-sided US-Letter-sized (8.5"x11") page of your own prepared notes which you may refer to during the exam. You have 120 minutes to complete all problems. You don't need to **`#include`** any header files, and you needn't guard against any errors or system call failures unless specifically instructed to do so. 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. 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.

Good luck!

SUNet ID (username): _____**`@stanford.edu`**

Last Name: _____

First Name: _____

I accept the letter and spirit of the honor code.

[signed] _____

Problems:

1. Short Answer	32 points
2. **`duet`**	40 points
3. Expression Evaluation	18 points
4. Multiprocessing	28 points
	120 points total

A reference sheet of function signatures and constants is included at the back of the exam.

Problem 1: Short Answer [32 points]

A) [5 points] Your colleague suggests modifying the Unix v6 filesystem design to store inodes scattered across the disk, placed such that they are closer to the data for the files they represent, instead of in one contiguous inode table. In at most 3 sentences, give one benefit and one drawback of this approach.

B) [5 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?

B) [5 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.

C) [7 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.

D) [5 points] In a few sentences, explain why increased crash recovery capability means tradeoffs with performance, and give one specific example of such a tradeoff.

E) [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.

Problem 2: duet [40 points]

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.

Use the next page to present your implementation of **duet**. 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).

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

Problem 3: Expression Evaluation [18 points]

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 to print later - there is an **evaluate** method on them to do this that returns the evaluated result as an **int**. They can write this without using threads, like this:

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

    // assume this prints the vector contents
    printResults(results);
}
```

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 {
    vector<int> v;
    // Add any fields here
} ThreadInfo;

static void evaluate(Expression& exp, ThreadInfo& info) {
    // Add your code here to evaluate the expression / store the result
}

static bool concurrentAnd(const vector<Expression>& expressions) {
    ThreadInfo info;

    vector<thread> threads;
    for (size_t i = 0; i < expressions.size(); i++) {
        threads.push_back(thread(evaluate, ref(expressions[i]),
                                 ref(info)));
    }

    // Add your code here to wait for threads to finish

    printResults(info.v);
}
```

They have a struct to bundle together any fields needed by each thread, then spawn off a thread for each expression to evaluate it and need to wait for the threads to finish and then print the

results (the results can be printed in any order). Implement the remainder of this code by completing the specified parts. You should not use the atomic class if you know it (if not, ignore this sentence).

```
typedef struct ThreadInfo {
    vector<int> v;
    // Add any fields here




} ThreadInfo;

static void evaluate(Expression& exp, ThreadInfo& info) {
    // Add your code here to evaluate the expression / store the result









}

static bool concurrentAnd(const vector<Expression>& expressions) {
    ThreadInfo info;

    vector<thread> threads;
    for (size_t i = 0; i < expressions.size(); i++) {
        threads.push_back(thread(evaluate, ref(expressions[i]),
                                 ref(info)));
    }

    // Add your code here to wait for threads to finish









    printResults(info.v);
}
```

Problem 4: Multiprocessing [28 points]

A) [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`** accidentally omitted its three calls to **`close`**.

B) [20 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 directly to compile two target programs 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.c 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.c gcc fork-
puzzle.o -o fork-puzzle
Elapsed time: 0.103602930 sec
```

To compute timing information, you should rely the following type and function declarations:

```
struct timespec {
long tv_sec; // seconds amount long tv_nsec; // nanoseconds amount

};

int clock_gettime(clockid_t clk_id, struct timespec *ts); // ignore return value
void print_elapsed_time(const timespec *start, const timespec *finish);
```

The first argument to **clock_gettime** should always be the constant **CLOCK_REALTIME**, and the second argument should be the address of a legitimate **timespec** record that, because the first argument is **CLOCK_REALTIME**, is populated with the number of seconds and nanoseconds that have elapsed since January 1st, 1970 at midnight. The **print_elapsed_time** function computes the difference between the two records addressed by **start** and **finish** and prints that difference on its own line in the format you need, as with **Elapsed time: 0.103602930 sec**.

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**, **mysystem**, **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.)

```
int main(int argc, char *argv[]) {
```