Exam 2 (hash pipeline)

Overview

My program has a lot of small functions for doing various tasks such as creating a pipe, creating a child process, reading from and writing to a pipe, and performing the two hashing operations. My main looks like this:

```
int main() {
	int pipe2[2];
	createpipe(pipe2);
	int child2 = createchild();
	if(!child2) child2code(pipe2);

	int pipe1[2];
	createpipe(pipe1);
	int child1 = createchild();
	if(!child1) child1code(pipe1, pipe2);

	parentcode(pipe1, pipe2, child1, child2);

	return 0;
}
```

The createpipe() and createchild() are nothing special, they just help avoid repeated code and check for errors on the pipe() and fork() functions. Beyond these rudimentary functions, the real work gets done in child1code() and child2code(), and a little bit in parentcode().

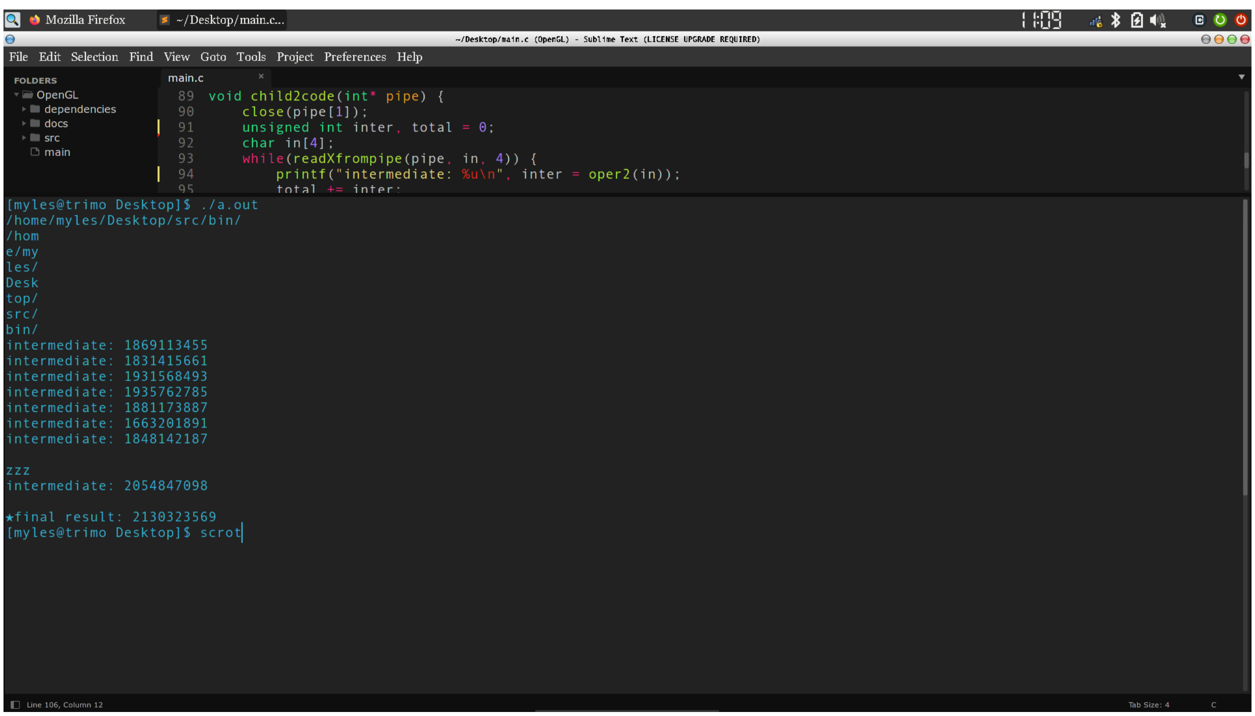
Extra Credit

Because of this property of modulus:

$$(a+b) \ mod \ n \equiv (b+a) \ mod \ n \equiv (a \ mod \ n + b \ mod \ n) \ mod \ n$$

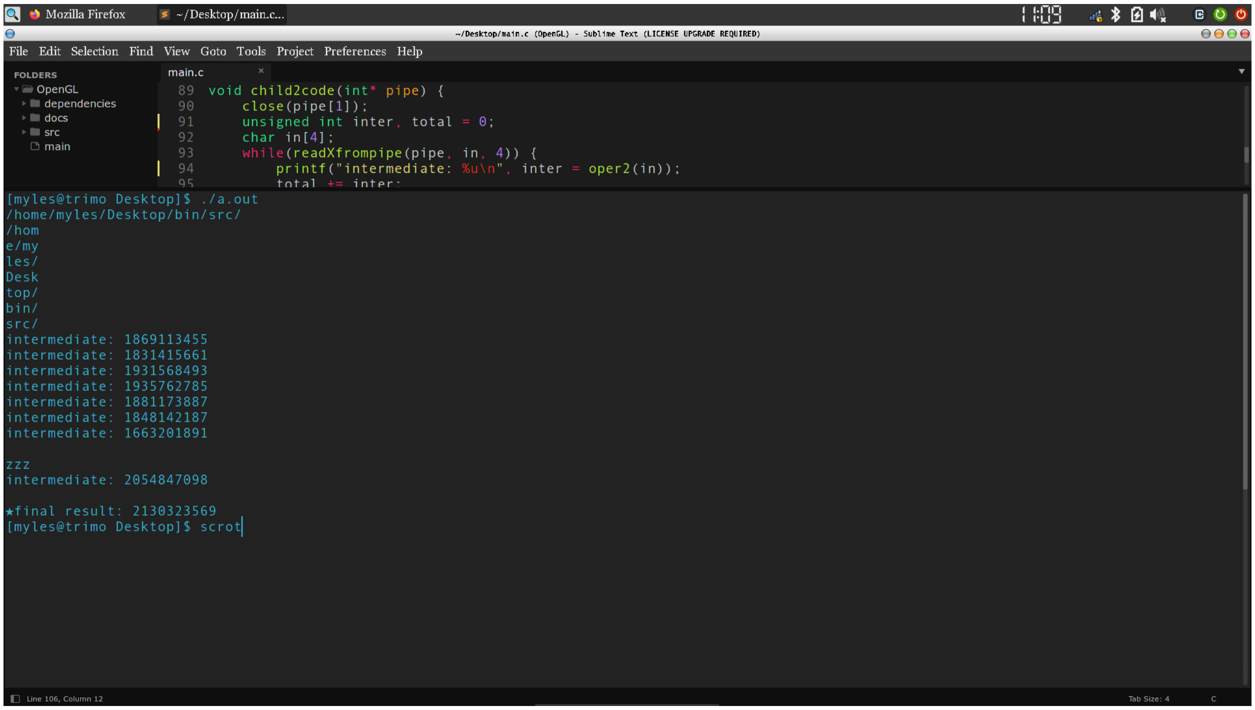
It doesn't matter what order we add our intermediate results (a and b) in. This is a problem if you want to get a unique hash key every time, because if you input two different sets of data, but these two sets contain the same a and b (just in a different order), you will get the same hash value back. This may very well be the case when you are giving the program pathnames:

Input: /home/myles/Desktop/src/bin/



Output: 2130323569

Input: /home/myles/Desktop/bin/src/



Output: 2130323569

As you can see, our two inputs are not identical and yet we received the same hash value of 2130323569. This is because the group of intermediate results in both cases is the same, except that bin/ and src/ are in a different order. Since addition is associative, we end up with the same running total and thus the same final result. Modulo is also associative, so even if we made our program do the modulo operation before adding a and b like this:

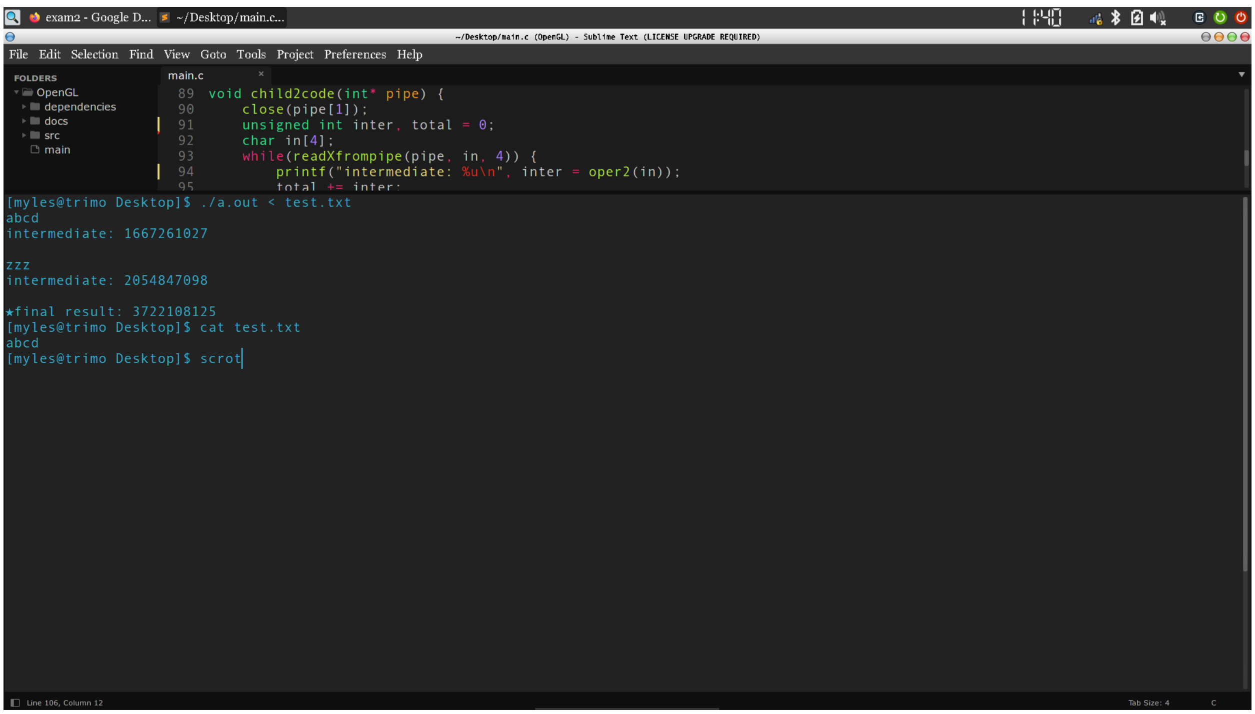
(a % n + b % n) % n

*where a and b are our intermediate results, and n is 2^32

We would get the same result: identical hash values even though our inputs are not identical. This is a big problem if we are relying on unique hash keys (for instance in cybersecurity), because it is not uncommon for two different input strings to be broken down into the same intermediate 4-byte groups by our program.

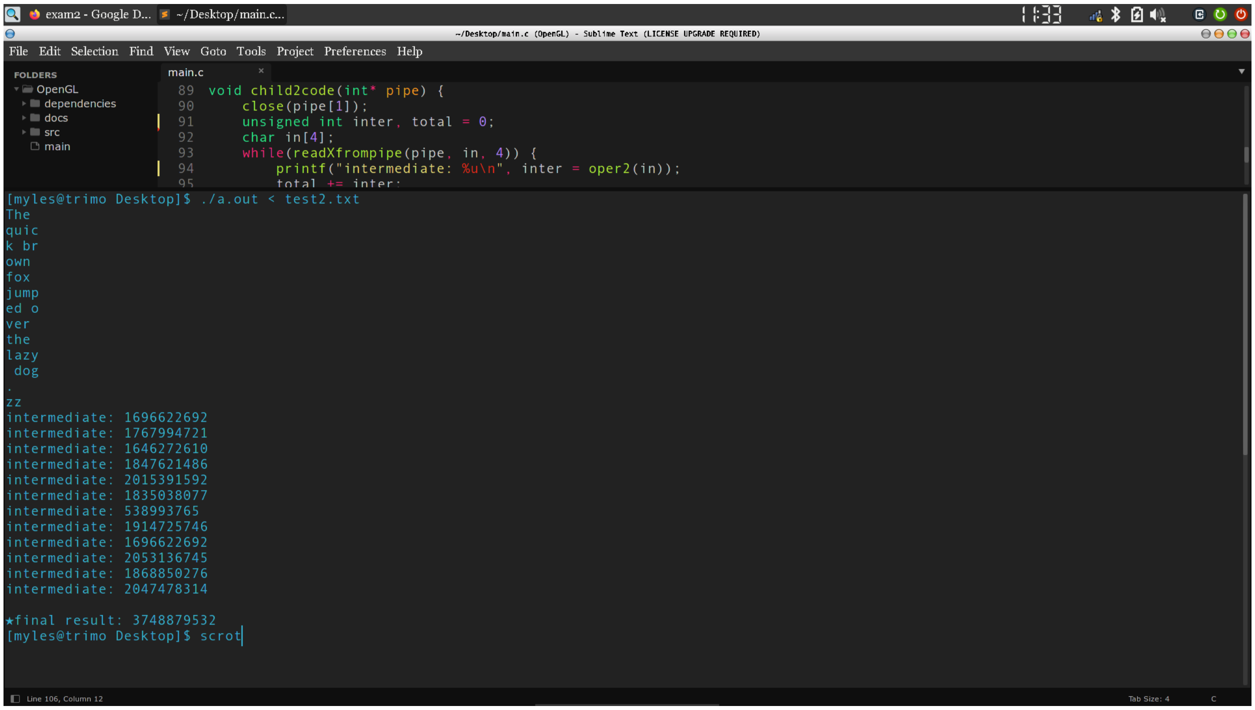
Sample Output

Input: test1.txt



Output: 3722108125

Input: test2.txt



Output: 3748879532

Parent

The parent code is what gets our pipeline started. The parentcode() function takes two int arrays representing pipe1 and pipe2, and also two ints representing our two children. The code looks like this:

```
void parentcode(int* pipe1, int* pipe2, int child1, int child2) {
        close(pipe1[0]);
        close(pipe2[0]);
        close(pipe2[1]);
        char c;
        while((c = getchar()) != EOF) writetopipe(pipe1, &c, 1);
        close(pipe1[1]);
```

```
    waitpid(child1, NULL, 0);
    waitpid(child2, NULL, 0);
}
```

It's pretty simple, the parent first closes both ends of pipe2 since it doesn't need it and closes only the read end of pipe1 since it only has to write to it. It then uses getchar() to receive input from stdin one char (or byte) at a time, and then immediately sends this input to pipe1 through the use of the writetopipe() function. When an EOF is received from stdin (which means the user hit ctrl-d or the file ended if a file was piped into the program), the write end of pipe1 can be closed and then the parent will wait for both children to exit before this function is allowed to return to main.

Child 1

The child1code() function takes two int arrays representing pipe1 (to read from) and pipe2 (to write to). The code looks like this:

```
void child1code(int* rpipe, int* wpipe) {
    close(rpipe[1]);
    char in[4];
    int numread;

    while((numread = readXfrompipe(rpipe, in, 4)) == 4)
        printchangewrite(in, wpipe);

    close(rpipe[0]);

    if(numread) {
```

```
            for(int i = numread; i < 4; i++) in[i] = 'z';
            printchangewrite(in, wpipe);
      }

      close(wpipe[0]);
      close(wpipe[1]);
      exit(0);
}
```

First it closes the write end of the read pointer since it doesn't need it, then it uses the readXfrompipe() function to keep reading 4 bytes over and over again until it reads less than 4 bytes. The readXfrompipe() makes use of the read1frompipe() function, which just reads one byte from a pipe into a char, in order to do its job. Each time 4 bytes is successfully read and stored in our char in[4], they are sent the printchangewrite() function:

```
void printchangewrite(char* in, int* pipe) {
   printf("%s\n", in);
   oper1(in);
   writetopipe(pipe, in, 4);
}
```

This function simply prints the four chars that were read, puts them through the first hashing operation, then sends the result of that operation to pipe2. Back in child1code(), after the while loop breaks as a result of not having read 4 bytes, the remaining space in our char in[4] array is filled with 'z' and then this padded array is sent to printchangewrite() as well. Finally, pipe1 and pipe2 are closed and child1 exits successfully.

Child 2

The child2code() function takes an int array representing pipe2. The code looks like this:

```
void child2code(int* pipe) {
	close(pipe[1]);
	unsigned int inter, total = 0;
	char in[4];
	while(readXfrompipe(pipe, in, 4)) {
		printf("intermediate: %u\n", inter = oper2(in));
		total += inter;
	}
	close(pipe[0]);
	printf("\n★final result: %u\n", total % ((unsigned
long)UINT_MAX + 1));
	exit(0);
}
```

First it closes the write end of pipe2 since it only needs to read from it, then it goes into a similar readXfrompipe() while loop as child1code(), except it doesn't care about the number of bytes read (it only cares about when it returns 0, so the loop can break). Each run around the loop, four bytes are read (this is guaranteed because of the padding that the first child did), and then they are run through the second hash operation and this intermediate result is printed and added to a running total. When the loop breaks, pipe2 is closed since it is no longer needed.

Finally, the sum of all results of the second hash operation is modulo'd by the maximum size of an unsigned int + 1, which is equivalent to 2^32. It has to be casted to an unsigned long first, though, since an unsigned int can only hold UINT_MAX and not UINT_MAX + 1. This final result is printed, and the second child exits successfully. After this, both children should have exited and so the waitpid()'s in parentcode() will stop waiting and allow our program to exit, having done its job.

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <limits.h>
#include <errno.h>
#include <string.h>

void createpipe(int* buf) {
    if(pipe(buf) == -1) {
        fprintf(stderr, "pipe returned -1 in createpipe: %s\n", strerror(errno));
        exit(-1);
    }
}

int createchild() {
    int pid;
    if((pid = fork()) == -1) {
        fprintf(stderr, "fork returned -1 in createchild: %s\n", strerror(errno));
        exit(-1);
    } else return pid;
}
```

```
void writetopipe(int* pipe, char* from, int amt) {
    if(write(pipe[1], from, amt) == -1) {
        fprintf(stderr, "write returned -1 in writetopipe: %s\n", strerror(errno));
        exit(-1);
    }
}

int read1frompipe(int* pipe, char* into) {
    int ret;
    if((ret = read(pipe[0], into, 1)) == -1) {
        fprintf(stderr, "read returned -1 in readchar: %s\n", strerror(errno));
        exit(-1);
    } else return ret;
}

int readXfrompipe(int* pipe, char* into, int amt) {
    char c;
    int count = 0;
    while(read1frompipe(pipe, &c)) {
        into[count] = c;
        count++;
        c = 0;
        if(count == amt) break;
    }
    return count;
}

void oper1(char* in) {
    char tmp[4] = {in[0], in[1], in[2], in[3]};
    in[0] = (tmp[0] | tmp[1]) & (tmp[2] | tmp[3]);
    in[1] = tmp[2];
    in[2] = tmp[1];
    in[3] = tmp[1] & tmp[2] & tmp[3];
}
```

```
unsigned int oper2(char* in) {
    unsigned int out = 0;
    out = out | (in[1] << 24);
    out = out | (in[3] << 16);
    out = out | (in[2] << 8);
    out = out | in[0];
    return out;
}

void printchangewrite(char* in, int* pipe) {
    printf("%s\n", in);
    oper1(in);
    writetopipe(pipe, in, 4);
}

void child1code(int* rpipe, int* wpipe) {
    close(rpipe[1]);
    char in[4];
    int numread;
    while((numread = readXfrompipe(rpipe, in, 4)) == 4) printchangewrite(in,
wpipe);
    close(rpipe[0]);
    if(numread) {
        for(int i = numread; i < 4; i++) in[i] = 'z';
        printchangewrite(in, wpipe);
    }
    close(wpipe[0]);
    close(wpipe[1]);
    exit(0);
}

void child2code(int* pipe) {
    close(pipe[1]);
    unsigned int inter, total = 0;
```

```
    char in[4];
    while(readXfrompipe(pipe, in, 4)) {
        printf("intermediate: %u\n", inter = oper2(in));
        total += inter;
    }
    close(pipe[0]);
    printf("\n★final result: %u\n", total % ((unsigned long)UINT_MAX + 1));
    exit(0);
}

void parentcode(int* pipe1, int* pipe2, int child1, int child2) {
    close(pipe1[0]);
    close(pipe2[0]);
    close(pipe2[1]);
    char c;
    while((c = getchar()) != EOF) writetopipe(pipe1, &c, 1);
    close(pipe1[1]);
    waitpid(child1, NULL, 0);
    waitpid(child2, NULL, 0);
}

int main() {
    int pipe2[2];
    createpipe(pipe2);
    int child2 = createchild();
    if(!child2) child2code(pipe2);

    int pipe1[2];
    createpipe(pipe1);
    int child1 = createchild();
    if(!child1) child1code(pipe1, pipe2);

    parentcode(pipe1, pipe2, child1, child2);

    return 0;
```

```
}
```