Augustana Campus, University of Alberta
AUCSC 380 Operating System Concepts
Winter Term, 2019

Programming Assignment 2
UNIX Process Control and File I/O System Calls

Due date: Friday, 2019 March 15, by midnight.

Objectives:

- To become familiar with the UNIX process control and file I/O system calls.
- To learn about the UNIX file pointer, seeking beyond the end of a file, and files with holes.
- To experience the implications of a parent process and child processes having the same file open for reading and initially sharing a file read pointer and the need for atomic updates to a file.

Assignment:

Write two C programs, the first of which uses the UNIX process control system calls (**fork**, **exec**, **wait**, **getpid**, and **exit**) and file I/O system calls (**open**, **close**, **lseek**, **read**, and **write**) to create multiple child processes, each of which will execute the second program to write binary data to a file and update data in the file.

In particular, your program should simulate the updating of a simple database of customer account balances by processing transaction files and updating customer account balances accordingly. It should also keep track of the number of transactions processed for each account.

The name of the binary file that will serve as the database of customer account balances should be provided to the program as its first command-line argument. The program should terminate with an error message and error condition if no filename is provided. Open the file using the `open` system call with appropriate flags such that your program can both read and write to the file and create the file if it doesn't exist. Do *not* use the `O_TRUNC` flag when opening the file and do *not* initialize every balance to zero on startup, since we should be able to process different transaction files (e.g., one each day or each month) in order to keep the account balance database up-to-date.

If the specified accounts file exists but is not readable or if it cannot be created, an error message should be written to the standard error stream, and the program should terminate without producing other output.

The program should accept additional command-line arguments, each of which is the name of a transaction file—a text file that will be read and processed by a child process. Each line of the file represents a single customer account transaction. Each transaction will consist of a pair of values: a non-negative integer representing the account number and a real value representing the transaction amount. Since transactions can be charges to the customer's account, refunds, or payments on the account, transaction amounts can be either positive or negative. In order to simplify the processing of transactions, you may use the C

Standard I/O package (i.e., the functions `fopen`, `fclose`, `fscanf` and the constant `EOF`) to read the transaction file until end-of-file is encountered.

Your account database update program might be invoked as follows:

```
acctdb accounts.db transactions1.txt transactions2.txt
```

In this example, two child processes would be spawned: one to process the transactions in the file `transactions1.txt` and the other to process `transactions2.txt`. Both child processes would update the account records in the binary file `accounts.db`.

For each transaction file named on the command line, the main program should `fork` a child process. The child process should begin by writing a line to the standard output, announcing its existence and its process ID (retrieved with `getpid`) with a message such as:

```
Child 1: 32856
```

The child should then call one of the `exec` system calls to replace its process image with the transaction-processing program you have written, passing it the name of the binary data file (shared by the parent process and all child processes) and the name of the transaction file that it should process.

Note: You should ensure (a) that the message printed by the child process is newline-terminated OR (b) that the child process calls the Standard I/O function `fflush` prior to the call to `exec`. If it doesn't do one (or both) of these, the line sent to the standard output (`stdout`) may not actually be printed. (The Standard I/O package buffers output sent to `stdout` until a newline is printed, `fflush` is called, or the process exits.)

Use `execlp` or `execvp` to facilitate the instructor's testing of your program. These versions of the `exec` system calls search the `PATH` of the user who invoked the main program to find the executable named (by base name only) as the argument to `execlp` or `execvp`. The instructor can compile each student's programs in separate directories, then test each pair of programs by declaring a custom `PATH` prior to running each student's main program.

The transaction-processing program that the child process will execute following the `exec` should open both the binary database file and the transaction file named in the arguments passed to the process. It should write an error message to `stderr` and terminate if either of the files does not exist or are not readable, if the database file is not writeable, or if there is a format error in the transaction file (e.g., only one value on a line, or alphabetic characters in the file that prevent parsing a field as an integer or a float).

The binary database should store each account balance as a C `double`, and should also keep track of the number of transactions processed for each account. You should create a `struct` to enable you to update account balances and transaction counts with a single read and write.

The account number should specify the offset of the corresponding account balance (and transaction count) in the database. Note that the account number is not the byte position of the corresponding account record in the file, but only the (zero-based) relative offset of the specified account record; the account number must be multiplied by the size of each data record (the `struct`) to determine the byte offset of the record in the file.

Once the entire transaction file has been processed, or if a read or write error occurs, the child process should exit with the appropriate exit status.

After spawning a child process for each transaction file named on the command line, the main process should wait for each child process to terminate, printing a message after each child terminates, indicating the process ID of the terminating child and its termination status. (You may use the function `exit_status()` in the file `exitstat.c` that is provided in *eClass* in the course section for Programming Assignment 2.)

The main program should then print a table of account numbers and corresponding transaction counts and account balances, but only for accounts with non-zero transaction counts. Conclude the table with the totals of all transactions and all account balances.

For example, using a single child process and a simple transaction file such as

```
3       27.5
82     -18.75
123   1234.56
18      50
27    2000.00
56     187.99
18     -30
3      -54.71
27    2239.22
18     -20
82     118.75
123   -234.56
27    -500
```

the output could be formatted as follows:

```
  3          2       -27.21
 18          3         0.00
 27          3      3739.22
 56          1       187.99
 82          2       100.00
123          2      1000.00
           ----    ---------
             13      5000.00
```

If the main program is invoked with a single command-line argument—the name of the binary database file—then it will not create any child processes, but will still print a table of account numbers, transaction counts, and account balances, with totals. This allows a user to review the contents of the accounts database after previous executions of the program.

Note that the binary database of account balances and transactions counts should only be processed using UNIX system calls (`open`, `close`, `lseek`, `read`, and `write`). The C Standard I/O library may be used to process the transaction files and to produce the output summary, but not to read or write the database file. (In general, it doesn't work to mix system calls with Standard I/O functions in any case, since the file position pointer will be changed by each system call and I/O function in incompatible ways.)

Error messages due to system errors should include the system-generated error message using the `perror` subroutine. Be sure to check that each system call was successful, and use `perror` to display a useful error message prior to terminating if a system call failed. (See the function `fatalsys()` in the file `cpfile.c`, provided in *eClass* in the course section for Programming Assignment 2, for an example of how to use `perror` to display system error messages.) Note that reaching end-of-file is *not* a system error, so a call to `perror` following EOF will produce a bogus error message.

References:

Most of the system calls mentioned above are discussed in *Beginning Linux Programming*, 4e: `fork`, `exec`, and `wait`, pp. 470–478; `open`, `close`, `read`, `write`, pp. 98–103; `lseek`, p. 106. See also functions `fscanf`, pp. 115–117, and `perror`, pp. 127–128.

`Fscanf` is also discussed in *C in a Nutshell* (pp. 200–205 and 334–335 in the 1st edition). The `exit` function and the macros to use as arguments to exit (`EXIT_SUCCESS` and `EXIT_FAILURE`), are discussed in *C in a Nutshell* (pp. 304–305 in the 1st edition).

For details regarding `getpid`, execute the following command at a Linux console:

```
man 2 getpid
```

Concurrent Execution:

Try testing your program to perform shared updates to the database file by running multiple child processes concurrently on the same transaction file, or on different transactions files, each of which access some of the same accounts as the other transaction files. Do you always get correct results? If not, why not? How might this problem be addressed?

Grading:

The weightings of the various factors for grading will be approximately as follows:

Correctness	60%
Design and Efficiency	20%
Style and Documentation	20%

Submission of Solution:

Submit the C source code for both programs via *eClass* by midnight on the due date specified. Your submission should include the main program `acctdb.c`, the transaction-processing program that will be executed by the child processes (with a file name of your own choosing), and any header file you might choose to create (e.g., to declare a `struct` to represent an account record). Assignments will be accepted after the due date and time, but late submissions will be assessed a penalty of **1% per hour** or portion of an hour.