Assignment 2: System-Wide FD Tables

For this assignment we will build a tool to display the tables used by the OS to keep track of open files, assignation of File Descriptors (FD) and processes.

How to compile

- With the help of a makefile, you can do `make program` or `make all` which will generate an executable called `showFDTables`. You can use `make clean` to remove the executable if already made.
- Without the help of a makefile, you can simply do `gcc main.c` which will generate an executable called `./a.out`. **You might see some warnings if you do it this way**. Do not worry, it does not alter or change the output in any way

Important Design Rules

Assume we compile with the help of the makefile. Here are some important rules that i stood by:

Priority

`./showFDTables` with any order of flags will still print in priority order. So even if i have `./showFDTables {...other flags} --per-process`, `--per-process` will still be outputted first.

The priority is: `per-process` , `systemWide`, `Vnodes`, `composite`, `threshold=X`. Keep in mind, the positional argument can go anywhere in between any of these. The priority will still hold.

For example, if we called `./showFDTables --threshold=x, --Vnodes --composite --per-process`. The output will still show the `per-process` table first, then `Vnodes`, then `composite`, then `threshold`.

Default Behaviour

`./showFDTables` or `./showFDTables pid` , where `pid` is a running, user-owned pid without any flags will just produce a composite table, as stated as one of the options on the handout

Threshold

To clarify, as confirmed by a TA, if any File Descriptor is over threshold, then its an offending process and you print the pid and the total.

For example, if a pid `foo` had `0` , `1` , `2` and `60` open for some reason and the threshold was `30`, `foo` would be an offending process and would be flagged as `foo(4)` in the output.

Also, threshold will still list all offending processes no matter whether we have a positional argument or not.

Duplicate Flags

Duplicate flags are allowed, but will only output it once. So for example, `./showFDTables --per-process` is the same as `./showFDTables --per-process --per-process --per-process --per-process`

Usage

```
./showFDTables --per-process #process FD table. No positional arguement so it will do it for all procceses

./showFDTables --systemWide #system-wide FD table. No positional arguement so it will do it for all procceses

./showFDTables --Vnodes #Vnodes FD table. No positional arguement so it will do it for all procceses

./showFDTables --composite #composite FD table. No positional arguement so it will do it for all procceses

./showFDTables --threshold=5 #flags proccesses that have file descriptors greater than 5, see Threshold section for more.


#we can chain flags together in any order
#will print process FD table, then Vnodes FD table, then composite FD table, then threshold FD table. See Priority for why

./showFDTables --threshold=x, --Vnodes --composite --per-process
```

```
#positional arguement where pid is a running user-owned process. check ps -u $USER for valid pids
./showFDTables pid

#will generate a process FD table like shown above except will only do it for a particular PID because of the positional argument
./showFDTables pid --per-process

#We can put the positional arguement anywhere. But as long as there is a positional arguement, any table generated will only be for the pid specified
./showFDTables --per-process pid --Vnodes threshold=39

#invalid, only one positional Arguement allowed
./showFDTables pid pid2

#default: will show composite table for all user-owned running procceses
./showFDTables

#default: will show composite table for just pid
./showFDTables pid
```

```

```

main.c

The code defines a main function that takes command line arguments and sets flags for various options that determine what functionality to execute.

The program uses the `getopt_long` function to parse the command line options and set the corresponding flag values. The options include per-process, system-wide, vnodes, composite, and threshold. The `getopt_long` function also handles any positional arguments and sets the value of `pid` if there is one.

The program then uses if statements to execute the desired functionality based on the values of the flags and the positional argument. If per_process is set to true, the program will call `getALLProcessFD()` or `print_fds(pid)` depending on whether a `pid` was provided. If system_wide is true, the program will call `allSWide()` or `systemWideFD(pid)` depending on whether a `pid` was provided. If `vnodes` is true, the program will call `printAllINode()` or `printINodeForPID(pid)` depending on whether a `pid` was provided. If composite is true, the program will call `printComposite()` or `printCompositeForPid()` depending on whether a `pid` was provided. If threshold is set to a value other than -1, the program will call the thresh function with the provided threshold value.

Finally, the program checks if none of the flags were set and if `pid` is -1, in which case it calls `printComposite()`. If `pid` is not -1, the program calls `printCompositeForPid()`. The program then returns `EXIT_SUCCESS`.

INode.h

This code consists of two functions, `printAllINode()` and `printINodeForPID()`, both of which print the inode numbers for file descriptors associated with a process.

The `printAllINode()` function prints the inode numbers for all processes currently running on the system. It first opens a directory stream to the `/proc` directory, which contains information about running processes. It then loops through the entries in this directory and checks if each entry name is a number (i.e., a process ID). If an entry is a process ID, the function creates a path string to the process's file descriptor directory and opens a directory stream to this directory. It then loops through the entries in this directory and checks if each entry name is a number (i.e., a file descriptor). If an entry is a file descriptor, the function creates a path string to the file descriptor and uses the `lstat()` function to get file status information and store it in the `sb` variable. Finally, the function prints the process ID and inode number of the file associated with the file descriptor.

The `printINodeForPID()` function prints the inode numbers for file descriptors associated with a specific process ID that is passed to the function as an argument. It first creates a path string to the process's file descriptor directory and opens a directory stream to this directory. It then loops through the entries in this directory and checks if each entry name is a number (i.e., a file descriptor). If an entry is a file descriptor, the function creates a path string to the file descriptor and uses the `lstat()` function to get file status information and store it in the `sb` variable. Finally, the function prints the process ID and inode number of the file associated with the file descriptor.

Both functions print their output in the same format, with a header row and a footer row consisting of a line of equal signs, and each row of output containing a process ID and its associated inode number.

The code includes several C standard library header files, including `unistd.h`, `stdio.h`, `stdlib.h`, `dirent.h`, `string.h`, and `sys/stat.h`. The `pid_t` data type is also used in the `printINodeForPID()` function, which is defined in the `unistd.h` header file.

processFD.h

This program is used to print out the file descriptors of a given process id or all process ids owned by the current user.

The program has several functions:

1. `printFDheader()`: This function is used to print out a header for the table of process ids and file descriptors.
2. `printFDfooter(): This function is used to print out a footer for the table of process ids and file descriptors.
3. print_fds(pid_t pid): This function is used to print out the file descriptors of a given process id. It first opens the directory /proc/<pid>/fd where the file descriptors are listed. Then it loops through each entry in the directory, skipping the . and .. entries. For each entry, it converts the entry name to an integer (the file descriptor number) and prints out the process id and the file descriptor number. Finally, it closes the directory.
4. printFDwoHead(pid_t pid): This function is similar to print_fds() but it doesn't print out the header of the table.
5. getALLProcessFD(): This function is used to print out the file descriptors of all process ids owned by the current user. It first gets the current user id using getuid(). Then it opens the directory /proc where all processes are listed. For each entry in /proc, it skips non-numeric entries (not process ids) and converts the entry name to an integer (the process id). It then reads /proc/<pid>/status to get the user id of the process owner. If the process is not owned by the current user, it skips it. If the process is owned by the current user, it calls printFDwoHead() to print out its file descriptors. Finally, it prints out a footer for the table and closes /proc directory.

composite.h

This defines two functions: `printComposite()` and `printCompositeForPid()`.

The `printComposite()` function prints the process ID, file descriptor number, filename, and inode number for all running processes on the system. It opens the `/proc` directory, reads its entries, and checks if each entry is a process ID by checking if its name starts with a number. For each process, it creates a path to its file descriptor directory, opens it, reads its entries, and checks if each entry is a file descriptor. For each file descriptor, it creates a path to the file descriptor and reads its symbolic link to get the filename associated with it. It then prints the process ID, file descriptor number, filename, and inode number for the file descriptor.

The `printCompositeForPid()` function does the same as `printComposite()`, but only for the process ID passed as an argument. It creates a path to the file descriptor directory for the given process ID and reads its entries to print the same information as `printComposite()`.

systemWide.h

This includes standard libraries such as <dirent.h>, <stdio.h>, <stdlib.h>, <unistd.h>, and <sys/types.h>. It also defines a macro for PATH_MAX with a default value of 4096 if it is not already defined.

The code contains three functions: `print_header()`, `print_footer()`, `systemWideFDPositional(pid_t pid)`, and `allSWide()`.

The `print_header()` function prints a header for a table that will be used later in the `systemWideFDPositional()` and `allSWide()` functions.

The `print_footer()` function prints a footer for the table.

The `systemWideFDPositional(pid_t pid)` function takes a process ID as an argument and prints out the file descriptors associated with that process. It does this by opening the directory `/proc/<pid>/fd`, where the file descriptors are listed, and looping through each entry in the directory. It skips `.` and `..` entries, converts the entry name to an integer (the file descriptor number), and gets the path of the file descriptor using `readlink()`. Finally, it prints out the process ID, file descriptor number, and path.

The `allSWide()` function prints out the file descriptors for all processes owned by the user. It does this by opening the `/proc` directory, looping through each entry in the directory, skipping entries that are not directories and entries that are not numbers (i.e., not process IDs). It then checks if the process is owned by the user and, if so, calls `systemWideFD()` function to print out the file descriptors associated with that process.

There is commented-out code at the end that calls the `printFDPath()` function with a process ID argument and returns 0.

thresh.h

This code defines a function `thresh` that takes an integer `x` as input.

The function opens the `/proc` directory and iterates over its contents using `readdir`. For each entry in the directory, the function checks whether the name of the entry can be read as an integer (i.e., whether it represents a process ID). If so, the function creates a path to the process's file descriptor directory by concatenating the process ID to the string `"/proc/"`. It then opens the file descriptor directory and iterates over its contents using `readdir`. For each entry in the file descriptor directory, the function checks whether the name of the entry can be read as an integer (i.e., whether it represents a file descriptor). If so, the function updates a variable `max_fd` with the maximum file descriptor number found.

After iterating over all processes and their file descriptors, the function prints a list of processes whose maximum file descriptor number is greater than `x`, in the format "pid (max_fd), ".

The code relies on several standard library functions (`opendir`, `readdir`, `sprintf`, `sscanf`, `closedir`, `printf`) and on the definition of several structures (`DIR`, `struct dirent`).