UF UNIVERSITY of FLORIDA
Herbert Wertheim
College of Engineering

COP4600

P2: POSIX Standards & Bindings

Prologue

Your work in the Legion has been highly praised, but you've had a creeping concern since. You'd always assumed that, as a faithful soldier, you would be rewarded and be permitted to serve as a vassal upon the subjugation of Earth. However, through the metadata you've analyzed, you've discovered that ultimately "lesser creatures" – including humans such as yourself – will likely be eliminated once Earth is conquered. As you continue your investigation, you take on another task with hopes you'll still be considered a useful pawn on the board. This time, you're tasked with creating a program capable of printing out the contents of specified files. Realizing the importance of the documents being stored on Sky Skink, the Legion's cloud system, you'll also create a backdoor that connects the file's contents to a GUI for readability for later access. It might come in handy down the road.

Overview

Modern operating systems are often built in layers with specific goals and limited privileges. Layering also facilitates the implementation of recognized standards by separating hardware and OS-specific implementations from generalized API calls. Often these layers are written in different languages and permit access via a binding layer to lower level functionality. In Android, POSIX functions can be called from Java applications via the Native Development Kit (NDK).

In this project, you will build a C++ language function that calls POSIX system library procedures to read a text file and return it as a C-style string (null-terminated character array). You will also build two programs that use your C++ functions – a simple program at the Android application layer that uses the NDK binding to call your C++ functions to read and display a text file in an application window, and another version that will print the text file to the screen in Ubuntu. We will provide a program that exercises your new call and program on the command line. You'll then create a short video to demonstrate your code. You'll submit the project via Canvas.

Structure

The project is broken into four main parts:

1) Create a C++ function that reads a text file via POSIX calls and returns a pointer to its contents
2) Create a short program that uses the function above to display a file via the command line
3) Create a simple Android GUI application that accepts a text file name as input and has a text box
4) Bind the function via the NDK to that the application to display the contents of the file in the text box

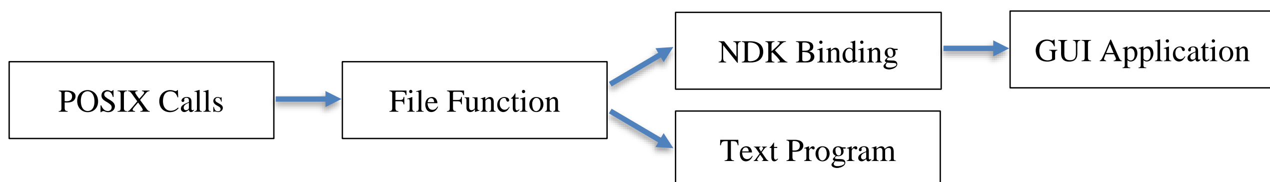


Figure 1: A function is called from a text program, and separately, bound to the GUI application.

While exact implementation may vary, the library functions must match the signatures laid out in this document.

Specification

Students will write several sections of code according the following specifications.

File Reader Library

The files **read_file.h** / **read_file.cpp** will contain declaration / definition of this function:

```
char *read_file(const char *filename)
```

Makes POSIX calls to read the contents of filename from disk to be stored in a null-terminated character array (C-style string). The array should be allocated dynamically. A pointer to the array will be returned. ***The caller will be expected to free the memory allocated for the array.*** If the file is not found, it should return **nullptr**.

Text Program

The text program, once built, should have the name **displayfile** and should take exactly one command line argument – the filename of the file to be displayed on the screen. It should use the **read_file()** function. If a file cannot be found, the text program should print "**Error: File Note Found**":

```
$ ./displayfile /sdcard/example.txt
Hello world!
This is the last line of the file.
$

$ ./displayfile nope.txt
Error: File Not Found
$
```

Contents of **/sdcard/example.txt**

```
Hello world!
This is the last line of the file.
```

*Note that **nope.txt** does not exist.*

GUI Program

The GUI program skeleton **provides** three GUI elements (without functional code). Students must add code as necessary to elicit the following behavior:

1) One-line text box where the file to be displayed will be typed by the user, named **filenameBox**
2) Button to submit the filename (which should trigger the read and display), named **submitButton**
3) Multi-line text box where the file will be displayed when the button is pressed, named **displayBox**

Students should modify the Java source to invoke the C++ functions. A simple example of transmitting a string from C++ to Java is included in the project base code. You must use the exact same source files created in the File Reader Library for this task to receive credit, *comments and includes are no exception*. As in the text program, if a file cannot be found, then the GUI must display the error message in the text box ("**Error: File Not Found**").

NOTE: You will need to add any new C++ source files to the **CMakeLists.txt** build file! When accessing files, it must be the absolute path starting with **/mnt/ubuntu/<abs_path>**, do NOT prepend this in your code!

Submissions

You will submit the following at the end of this project:

- Report (**p2.txt**) in man page format on Canvas, including link to unlisted screencast video
- Compressed tar archive (**displayfile.tar.gz**) containing source/build files for text program on Canvas

- Zip archive (`nativeapp.zip`) containing source/build files for the GUI program on Canvas

Report

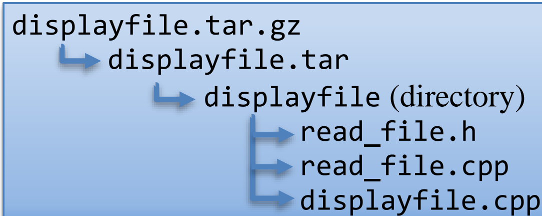
Your report will explain how you implemented the function and programs, including which POSIX calls were invoked and why. It will include description of how testing was performed along with any known bugs. The report should be created using `man` format and should be no more than 500 words (about two man pages), cover all relevant aspects of the project, and be organized and formatted professionally – **this is not a memo!**

Screencast

In addition to the written text report, you should submit a screencast (with audio) walking through the code you wrote to build the two applications and invoke the POSIX calls. Additionally, the screencast should include you showing/demoing your changes in action. (no more than 5 minutes).

Compressed Archive (displayfile.tar.gz)

Your compressed tar file should have the following directory/file structure:



To build the text program, we will execute these commands:

```
tar zxvf displayfile.tar.gz
cd displayfile
c++ displayfile.cpp read_file.cpp -o displayfile
cd ..
```

We'll run commands like this to test the text program:

```
displayfile/displayfile example.txt
displayfile/displayfile /home/reptilian/example.txt
```

To include the read_file() function in our tests, we'll using this directive in C:

```
#include "displayfile/read_file.h"
```

To test the read_file() function in a program, we will execute this command:

```
c++ program_name.cpp displayfile/read_file.cpp -o program_name
```

Zip Archive (nativeapp.zip)

To create a compact package from your Android project, select **File → Export to Zip File** from the Android Studio menu system. This will package your project for upload.

Please test your functions before submission! If your code does not compile it will result in **zero credit** (0, none, goose-egg) for that portion of the project.