

P2: POSIX Standards & Bindings

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 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.

NOTE: Take Snapshots in VirtualBox! You will most likely brick your machine at some point during this or other projects, and you will not want to start from scratch. No, seriously – take snapshots!

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

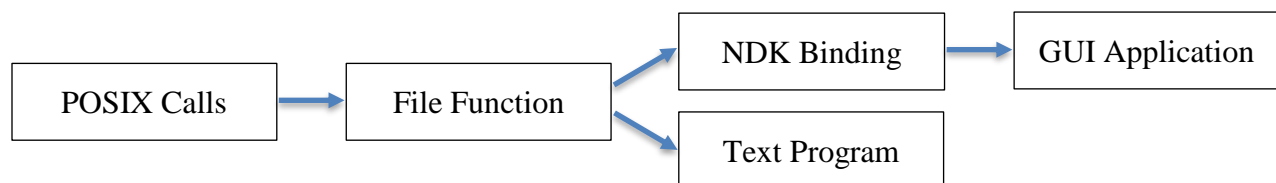


Figure 1: A function is called from a text program, and separately, bound to the GUI application.

While exact implementation may vary, the library functions must match the signatures laid out in this document, and the system calls must apply the security model properly.

Specification

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

File Reader Library

The file `read_file.h` and `read_file.cpp` will contain declaration and definition, respectively, 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.***

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:

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

Contexts of /sdcard/example.txt:

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

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.

NOTE: You will need to add any new C++ source files to the `CMakeLists.txt` build file!

Submissions

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

- Report on Canvas
- Screencast on Canvas
- Compressed tar archive (`displayfile.tgz`) containing source/build files for text program on Canvas
- Zip archive (`nativeapp.zip`) containing source/build files for the GUI program on Canvas

Report

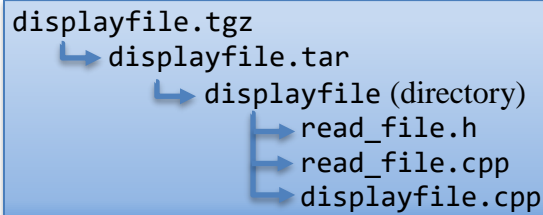
Your report will explain how you implemented the function and programs, including which POSIX calls were invoked and why. It will include description of how testing was performed along with any known bugs. The report may be in Portable Document Format (pdf) or plain-text (txt). The report should be no more than a page and should cover all relevant aspects of the project.

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 (~5 minutes).

Compressed Archive (displayfile.tgz)

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



```
displayfile.tgz
├── displayfile.tar
│   └── displayfile (directory)
│       ├── read_file.h
│       ├── read_file.cpp
│       └── displayfile.cpp
```

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

```
tar zxvf displayfile.tgz
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 use 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.