Task Manager Android Application
Technical Manual

November 12, 2020

Slippery Rock University

Xavier Julius

Olivia Warner

Table of Contents

1.0 Task Manager Android Application Introduction 3
1.1 Document Identification 3
1.2 Application Description 3
1.3 Challenges 3
1.4 System Requirements 3
2.0 Creating a New Project in Android Studio 4
2.1 Android Studio Download 4
2.2 Creating a New Project 4
3.0 Tab System using Fragments 5
3.1 Creating Blank Fragments 5
3.2 SectionsPagerAdapter Class 6
4.0 Application Permissions 7
4.1 Administrative Privileges 7
4.2 Requesting User Permissions 8
4.3 Application Details Button 9
5.0 Packet Manager 9
6.0 Rooting 10
6.1 Rooting Phone 10
6.2 Rooting Emulator 11
7.0 Linux Commands 12
7.1 Running Commands 13
7.2 Su Command Checking 13
7.3 Linux Commands Used 16
7.3.1 Details 16
7.3.2 Services 16
7.3.3 CPU Utilization 17
8.0 Displaying Output 17
8.1 Splitting the Output String 17
8.2 Array Adapter 18
8.3 ListView 18
8.4 SearchView 18
9.0 Services Floating Action Button 19
10.0 Killing Services and Processes 20
Conclusion 21

1.0 Task Manager Android Application Introduction

In our application we created an application that can be used on an Android device to give the user specific operating system information. This Task Manager Android Application is similar to the Windows Task Manager, but it focuses on the details, services, and CPU tabs.

1.1 Document Identification

This document serves as a Technical manual for the Task Manager Android Application.

1.2 Application Description

In our application we created a simple app that can be used on an Android device to present the user with important operating system information. It was designed with the intention of giving a user more information and control over the device it is installed on. Managing "tasks" as the title of our project states refers to the processes running on the device itself. Now these processes can be anything from a system process, another app running in the background, or a service that an app or the system has running. The app is set up in easy to use tab system where each tab or section displays important information about the device and the processes running on it.

1.3 Challenges

The biggest challenge for this project is related to the security of the privileged information we want to extract. The information we want is protected because the operation system relies on it. Any app installed will not have access to this operating system information. To continue with the progress of our application we needed to find a way to access this protected operating system information and display the output to the users. Android in the newer versions of their APIs are securing their security, so the operating system information gathered in this application was difficult to retrieve.

1.4 System Requirements

The system requirements to work on this project can differ depending on if you are using a real phone or an emulator like we ended up using. Here are what we think is required to work on a project like this one:

- At least 8GB of RAM(more if you are using an emulator)
- Android Studio IDE
- Emulator or device that is capable of being rooted

2.0 Creating a New Project in Android Studio

2.1 Android Studio Download

To get started you need to download the Android Studio IDE. Follow these steps to get started:

1. Navigate to the Android Studio Download Page via this link: https://developer.android.com/studio/?gclid=Cj0KCQiAy579BRCPARIsAB6QoIarS2GjQepoKZRFT9dlB-vyfwMIixZBiqP2CizLvgnUR4xDJdF0PBgaAnFjEALw_wcB&gclsrc=aw.ds

2. Click the download Android Studio button for Windows (for other systems select the download options and then select the appropriate download for your system).

3. Open the Android Studio Installer and follow the normal instructions .

4. Open Android Studio Application.

2.2 Creating a New Project

The next step is to create an Android Studio Project with a tabbed activity system generate the application with all the basic files needed. This includes basic fragments that will be used or deleted depending on how you with your tabs to be.

1. Open Android Studio IDE

2. Select New Project

3. On the tab called Phone and Tablet there will be multiple project types. Select “Tabbed Activity” and click Next.

4. On this screen you will give your application a name, package name, the location to be saved , the language (in this case java), and then the minimum Android SDK that you would like to have (we chose marshmallow android 6.0) .

3.0 Tab System using Fragments

Even though the project is created as a Tabbed Activity project, does not mean that the tabs are automatically created for you. These tabs need to show different output in each tab. The creation of the project does not do this for you.

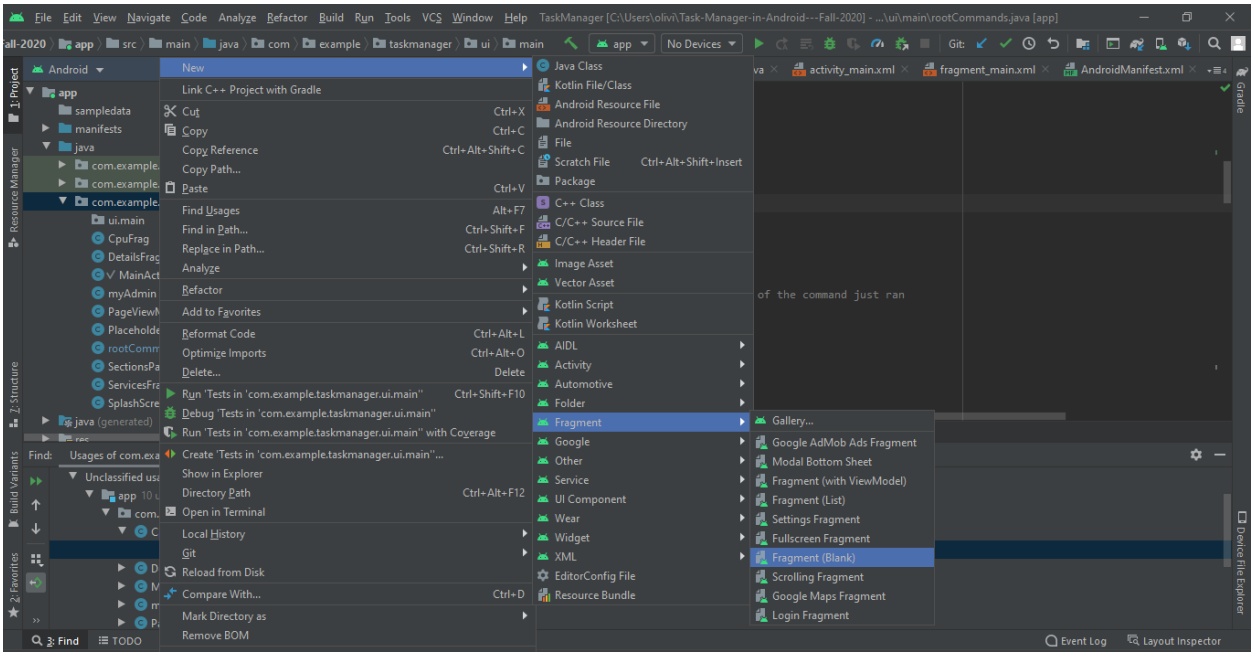
3.1 Creating Blank Fragments

1. Right click on the folder of all your .java classes

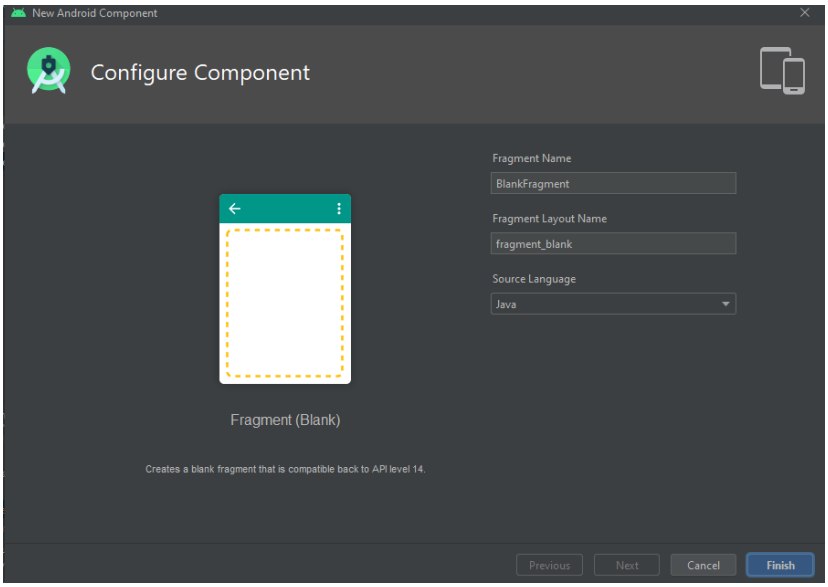
2. Hover over New

3. Navigate to Fragment

4. Select Fragment (Blank)



5. Now you are going to configure the name of the fragment and its layout. Fragments have their own layout and their own class. This becomes very important within a tabbed layout.



6. From here you can change the layout of your application within the xml file. It can be important to change the background color for testing purposes to show that each tab is showing their own information.
7. From here you can create as many fragments as you would like. In the Task Manager Android application, 3 different fragments were made for 3 different tabs.

3.2 SectionsPagerAdapter Class

SectionPagerAdapter.java class. This class is automatically created with the creation of a Tabbed Activity project. This class is set up for you to easily change and alter your tabs.

1. The tab titles were changed to the appropriate tab titles for the application. These "tab_text_#" definitions are created within the strings.xml file located in the res folder.

```
private static final int[] TAB_TITLES = new int[]{R.string.tab_text_1,
R.string.tab_text_2, R.string.tab_text_3};
```

2. The getCount() function needs to be changed to the # of tabs you are creating. For this application, we have 3 tabs.

```
@Override
public int getCount() {
    // Showing 3 total page count tabs
    return 3;
}
```

3. A case statement was created inside the getItem() function. This function gets the different items and returns them. It takes the parameter of position, which gets the number position of the fragment. This case statement switches the position parameter.

Each case sets a fragment to the fragment java class that was created. In this case, we have each case going to each of our tab fragments. Case 0 starts with the first tab that is located all the way to the left.

```
@Override
public Fragment getItem(int position) {
    Fragment tabFrag;

    //switch statement to go between the 3 different fragments. Case 0 starts
with the first tab position
    //[1] https://youtu.be/HHd-Fa3DCng
    switch(position) {
        case 0:
            tabFrag = new DetailsFrag();
            break;

        case 1:
            tabFrag = new ServicesFrag();
            break;

        case 2:
            tabFrag = new CpuFrag();
            break;

        default:
            throw new IllegalStateException("Unexpected tab value: " +
position);
    }
    return tabFrag;
}
```

4.0 Application Permissions

At the start of our application, our team started executing the idea of administrative privileges on the Android operating systems. If we already had administrative privileges, we thought that we could verify permissions by asking the user. This method was later removed because it was not the direction that our team was wanting to take to progress in this application.

4.1 Administrative Privileges

One of the ways we attempted to get the information we needed what to get access to the Admin API. This is done by registering your app as an admin app on the device. The first step in doing this is to create a class that we called admin class that extends the DeviceAdminReceiver class.

```
public class myAdmin extends DeviceAdminReceiver {
    @Override
    public void onEnabled(Context context, Intent intent) {
        super.onEnabled(context, intent);
        //toast message to show that the admin has been enabled
        Toast.makeText(context, text: "Device Admin is now Enabled", Toast.LENGTH_SHORT).show();
```

NOTE: In Source Document [5] https://medium.com/@isuru.2014033_41377/make-a-device-admin-application-android-emm-b08e6a134

1. The next step is register it in the manifest.xml file. This is done by going into the manifest file and adding a receiver section that list the name, description, label, and the permission. The permission should be andriod.permission.BIND_DEVICE_SERVICE. It should also have an intent filter in it that defines and action to enable the device admin.

2. Following that you will need to create a new xml file that will hold the policies that you with to use in it. To connect the receiver and the policy file you go back to the manifest file where you created the recover and inside the receiver creates meta data tag. Inside this tag you will give the meta data a name and then a resource to reference. The resource will be your policies xml file.

```
<meta-data
    android:name="android.app.device_admin"
    android:resource="@xml/policies"
```

   NOTE: In Source Document [5] https://medium.com/@isuru.2014033_41377/make-a-device-admin-application-android-emm-b08e6a134

3. Finally, you will need an instance of the DevicePolicyManager class the use the permission you have places in you polices xml file. To use the permission or access you just put the name of your DevicePolicyManager.nameofFunction() to use it.

Our team thought that this would give us access to the services and details information that we needed, but quickly found out that it only gave us access to user information like changing their password or resetting it and also the ability to do actions like lock and unlock the phone. We deemed this method unsuccessful because it got us no closer to the information that we wanted.

4.2 Requesting User Permission

At the start of our development of the application, we thought that we could ask the user permission to access some of the protect operating system information. The permissions are documented through Android Studio, were not relevant to what we wanted to accomplish. This

feature was later removed as the application progressed because of irrelevant built-in permissions.

The permission we kept in our application was to kill background processes. This permission was to be used to kill the processes. It uses the ActivityManager to get that system service for the application. Killing processes was not complete by our team, but we would like to add this to future work on this application.

Permission in AndroidManifest.xml:

```
<uses-permission android:name="android.permission.KILL_BACKGROUND_PROCESSES"/>
```

NOTE: In Source Document: [4] http://androidpermissions.org/android-permission-KILL_BACKGROUND_PROCESSES/

Setting Permission in MainActivity.java:

```
ActivityManager killB = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
killB.killBackgroundProcesses(KillPackage);
```

NOTE: In Source Document: [4] http://androidpermissions.org/android-permission-KILL_BACKGROUND_PROCESSES/

4.3 Application Details Button

Our team thought that it would be a good idea for the user to be able to see the permissions they granted, or the permissions being used within the application. Our team created a button at the top of the application that would directly go to the settings of the application. This screen would also allow you to force quit the application from directly inside the application.

This was done by creating a new intent that would start a new activity. A resource from inside the Android Studio Developer guide provided different setting preference objects. The one used for our application was the “Action_Application_Details_Settings”.

Setting Action to Preference Object Screen:

```
appInfo.setAction(Settings.ACTION_APPLICATION_DETAILS_SETTINGS);
```

NOTE: In Source Document: [3] https://developer.android.com/reference/android/provider/Settings

5.0 Packet Manager

One of the first methods we tried to use to get service and details tab information was using one of the built-in managers that the Android API provides called the packet manager. This manager is used to manage all installed packages on the device. The method we tried to use was to enumerate all installed packages and display the services that may or may not have been inside each package. We had assumed that the data for each service would be embedded inside each install package.

The reason this method of getting the services and details tab information is because the packet manager does not have direct access to the system services or details information. Instead the manager is used more as a method to track how many are active and what they are doing. So, when calling the toString() method we would get the wrong information.

6.0 Rooting

Rooting an Android device means that you are giving yourself or the user of the device root permission on the device. This allows us to run su or what are called root commands in our application. If the device is not rooted our app cannot run the commands that it needs to get all the information about the device. Rooting can also be a dangerous task; you can potentially ruin whatever device you are trying to root if it is done incorrectly.

6.1 Rooting Phone

There are plenty of resources on the internet to root an actual device, but they are all different depending on your actually device. Some phones or tablets cannot even be rooted at the current time. We attempted many ways to root the actual phone we had but had no luck rooting it.

1. Rooting directly from LGV30

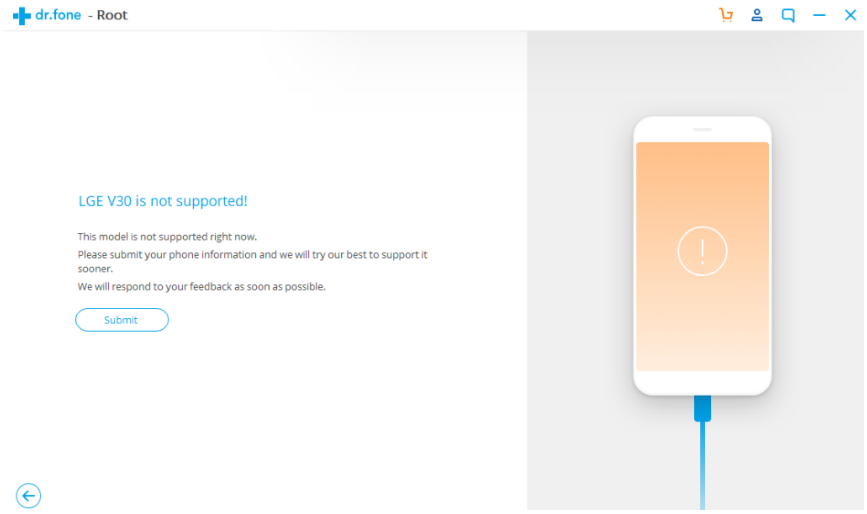
    - Kingo Root - at 90% it would give a root installation error

    - Towelroot - did not work at all
    - Framaroot - would not let you open the downloaded apk file
    - Magisk - in the process of trying to root but I cannot get it to run out of the zip file to install onto the device

2. Rooting from the PC

    - Kingo Root - have tried it multiple times and each time it says that it has failed

- Dr. Fone Root - said that the phone was not compatible



6.2 Rooting Emulator

When we had no luck rooting an actual device we turned to rooting and emulator. There are many emulators out there that actually come with the option to have them rooting. So, this method was much easier and safer considering there is no actual device that can be destroyed. Android studio has its own emulator that comes with the software, but we could not figure out how to root it. So, we decided to look into rooting an emulator which was the way we ended up working with a rooted device.

- Bluestacks Emulator - had trouble running the emulator
- Nox Player Emulator - directly connected with Android Studio with root privileges.
  - Slow
  - Uses a lot of memory storage
- MEmu Play Emulator - directly connected with Android Studio with root privileges. This emulator is the one we decided to progress with for our application

Steps Taken to Install MEmu Play Emulator:

1. Download the MEmu Android Emulator: https://www.memuplay.com/

2. Once installed...Go to the general settings and select the Root Option "ON"

3. Go to the SETTINGS app > About Tablet > Build Number

4. Press Build Number about 5 times to access Developer Mode

5. In Developer Mode > Enable USB Debugging. This allows Android Studio to connect and run the app on the emulator

6. Go to Android Studio and run the application in the emulator. The emulator NEEDS to be running before opening Android Studio to be able to connect

7. Download Root Checker apk file: https://apkpure.com/root-checker/com.joeykrim.rootcheck

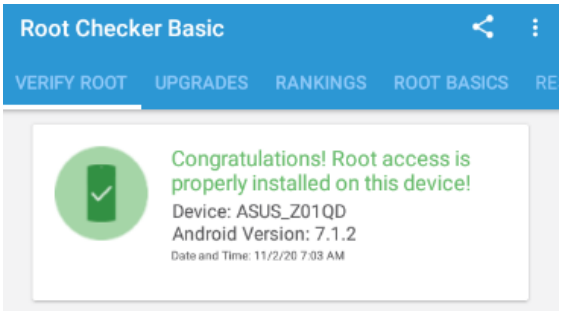
8. Add the apk file in the emulator on the right side

9. Open the apk file and verify the root (root will not be verified)

10. Download SuperSU apk file: https://apkpure.com/supersu/eu.chainfire.supersu

11. Open the apk file to the right

12. Set up SuperSU
13. Verify the root again in Root Checker



7.0 Linux Commands

To run Linux commands in an android app you need to understand that underneath the operating system of an android device is actually a Linux system. This means that most commands you would run there will work on an android device. Now not all normal Linux commands will work, for example the systemclt commands do not work when running from android device. Also, some commands that you would use the "su" to evoke super or root privileged commands the device that our android application is on must be rooted.

7.1 Running Commands in JAVA

To run Linux commands in JAVA we need to create an instance of the process class. You set the process to the Runtime method called getRuntime().exe(command). The command argument is just a string variable that holds the Linux command with the flags you desire. This command and the rest of the code I will talk about must be in a try and catch block to catch any invalid commands or flags.

Process Runtime Example:

```
Process process = Runtime.getRuntime().exec(command);
```

NOTE: In Source Document: [6] https://stackoverflow.com/questions/20932102/execute-shell-%20%20%20command-from-android

After passing the command in we create a bufferReader with an inputStream as the argument. Call the getInputStream method on the process to receive the output from the command. You will then loop through the buffer and call the read method to read each char and append it to the output string. Then you return the string at the end of the try.

- There is a method for the input stream called get line, but it has been depreciated with the recent updated to the android studio API.

```
BufferedReader reader = new BufferedReader(new InputStreamReader(process.getIn

int read;
char[] buffer = new char[24000];
StringBuilder output = new StringBuilder();
while ((read = reader.read(buffer)) > 0) {
    output.append(buffer, offset: 0, read);
```

NOTE: In Source Document: [13] https://stackoverflow.com/questions/7449515/run-shell-commands-from-android-program

7.2 Su Command Check

To make our code a little more reusable we created a class to handle checking if we have root privileges and to run as many commands as we need. To run commands, you just create an

instance of the class and then use the two methods that are associated with the rootCommands.java class.

1. runAsRoot() - this method takes in command string that you want to execute and then returns a string that is the entire output of that command. This is done using a buffer reader to store the output. This function is embedded in a try, catch.

runAsRoot() Code:

```
public static String runAsRoot(String command) {

    try {

        //[6] https://stackoverflow.com/questions/20932102/execute-shell-
%20%20%20command-from-android

        Process process = Runtime.getRuntime().exec(command);

        //[13] https://stackoverflow.com/questions/7449515/run-shell-commands-from-
android-program
        BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));

        int read;
        char[] buffer = new char[24000];
        StringBuilder output = new StringBuilder();
        while ((read = reader.read(buffer)) > 0) {
            output.append(buffer, 0, read);
        }
        reader.close();

        process.waitFor();
        return output.toString();
    }

    //exception catching
    catch (IOException | InterruptedException e) {
        try {
            throw new IOException("*** IO Exception was thrown ***");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        try {
            throw new InterruptedException("*** Interrupted Exception was thrown
***");
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    return "Command didn't work";
 }
```

NOTE: In Source Document: [6] http://muzikant-android.blogspot.com/2011/02/how-to-get-root-access-and-execute.html

2. canRunRootCommands() - this method is run when you need to check or ask for root command privileges for you app. We run this method in main activity before we attempt to run any commands. It is what prompts the user for access. If the user has already granted access then it will not ask for access upon reloading of the application.

canRunRootCommands() Code:

```
public static boolean canRunRootCommands()
{
    //sets the return value to false
    boolean retVal;
    Process suProcess;

    //anytime you want to write to the os it must be put in a try catch or java will
not let you do it
    try
    {

        //[6] https://stackoverflow.com/questions/33823794/running-root-commands-on-
android-device
        //invoking the su command
        //the first time su is run the user will get a message to deny or grant the
application super user access
        suProcess = Runtime.getRuntime().exec("su");

        //Allows for input and output to the command line that we opened
        DataOutputStream os = new DataOutputStream(suProcess.getOutputStream());

        //must user buffer reader because the readLine() method associated with
DataInputStream class has depreciated
        BufferedReader osRes = new BufferedReader(new
InputStreamReader(suProcess.getInputStream()));

        //command to get the id of the user
        os.writeBytes("id\n");
        //basically clearing the input buffer for next command
        os.flush();

        //reading the output form the command id above. this command should give the
current user id for the application
        String userID = osRes.readLine();
        //sets the exit su to false
        boolean exitSu;
```

```
        //these if statements ensure that the user id was given back and that it is 0
which means that we are now acting as super user.
        //if the device is not rooted or the user of the phone denies granting super
user access for the application then it will fail and the catch will execute
        if(userID == null)
        {
            retVal = true;
            exitSu = false;
        }
        else if(userID.contains("uid=0"))
        {
            retVal = true;
            exitSu = true;
        }
        else
        {
            retVal = false;
            exitSu = true;
        }
        if (exitSu)
        {
            os.writeBytes("exit\n");
            os.flush();
        }

        //catches the false return value
    } catch (Exception e) {
        retVal = false;
    }

    //returns the return value whether its true or false
    return retVal;
}
```

NOTE: In Source Document: [6] http://muzikant-android.blogspot.com/2011/02/how-to-get-root-access-and-execute.html

7.3 Linux Commands Used

Executing Linux commands is how we were able to retrieve the operating system information needed for this application. With SuperSU giving us permission to run root, we have to start the execution of the command like so: "su -c". The -c flag indicates that is command is next and it can execute.

7.3.1 Details

- /system/bin/ps - gave us some of the processes but not all of them
- ***su -c ps*** - works, gives us all the processes
- su -c ps-A - get output "bad pid A"

7.3.2 Services

- ***cmd -l*** - gives us a list of the services, but we want more information
- su -c systemctl - getting no output
- su systemctl --type=service - no output
- su systemctl -type=service state=all - no output
- /lib/systemd/system - no output
- ***service-list*** – gives a numbered list of all available services

7.3.3 CPU Utilization

- cat /proc/cpuinfo - *gives info about the cpu, but not what we are looking for*
- cat /proc/stat - *gives us CPU statistics but not exactly what we want*
- su -c sar - no output
- su -c mpstat - no output
- su -c top - crashes the entire application
    - You cannot run the full command because it will crash the application. Instead you have to specify the number of times it will analyze the CPU
    - ***su -c top -n 1*** - this iterates 1 time and allows itself to output into the fragment.

8.0 Displaying Output

When running the Linux commands, the output was presented as one big string. This made it very messy to read and understand the output. We split the output and put it in an ArrayAdapter where it was then displayed in a ListView.

8.1 Splitting the Output String

When using the rootCommands class that we created to handle executing Linux su or root commands the output will be stored in a string as we stated and then returned to the mainActivity where it was called. The only issue with this is that we need each line of the output to be its own string so we can store them in their own individual elements in the list view.

This was fairly easy issue to fix. Taking the string variable that we stored the output of the command into we can use a built in android method called split() on that string with the proper regular expressions as arguments to split the string up by each line.

Splitting Output String Code Example:

```
String serviceslines[] = rootCommandinfoServices.split( regex: "\\r?\\n");
```

NOTE: In Source Document: [11] https://stackoverflow.com/questions/454908/split-java-string-by-new-line

Finally, we loop over this array we just created to store the values into a global array. The global array we can reference from the appropriate fragment for that section of the app.

8.2 Array Adapter

The ArrayAdapter is a class that you use to display to a list view. To do this you create an instance of the class passing the layout and array you want to display. You create an activity instance calling the getActivity() method with the type MainActivity to get the instance of the activity currently running. Once this is done you can reference any global variable inside of main activity

Example of setting the ArrayAdapter:

```
servicesLIST.setAdapter(listViewAdapter);
```

NOTE: In Source Document: [10] https://guides.codepath.com/android/Using-an-ArrayAdapter-with-ListView

8.3 ListView

. To display the ArrayAdapter we link it to a ListView. You create a ListView class instance and link it to the ListView in the fragment xml file by using the R.id.Name_of_item which gets you the id of the item in the fragment's name you use. Then you set that views adapter to the ArrayAdapter you created, and now the contents of the array will be in the ListView displayed to the user.

```
final ArrayAdapter<String> listViewAdapter = new ArrayAdapter<>(getActivity(),
android.R.layout.simple_list_item_1, mainActivity.detailsInfo);
```

NOTE: In Source Document: [10] https://guides.codepath.com/android/Using-an-ArrayAdapter-with-ListView

We ran into some issues where the ListView would be cutting off the first or last element that was to be displayed. To fix this issue we set the layout width and height to "match_parent" in the connected xml layout.

8.4 SearchView

A SearchView widget was added to the top of each tab to be able to search for processes and services at a much quicker rate.

1. The widget needs to be added to each xml layout file. Make sure you have your constraints set and it is position in the way you would like it to.
2. Creating the .setOnQueryTextListner method
    - Android Studio will create the next two methods automatically
3. onQueryTextSubmit(String s) – this is not necessarily needed for our application because we are not submitting anything. It takes the parameter String s.
4. onQueryTextChange(String filteredDetails) – needed for our application, when text is changed within the SearchView, it will change the ListView ArrayAdapter. It uses the method of.getFilter() and .filter(). In the filter, we have to define what we are filtering and that is the string parameter.

SearchView Filter Code:

```
searchView.setOnQueryTextListener(new SearchView.OnQueryTextListener() {
    @Override
    public boolean onQueryTextSubmit(String s) {
        return false;
    }
    @Override
    public boolean onQueryTextChange(String filteredDetails) {
        listViewAdapter.getFilter().filter(filteredDetails);
        return false;
    }
});
```

NOTE: In Source Document: [12]
https://developer.android.com/reference/android/widget/SearchView

9.0 Services Floating Action Button

In our application, we wanted to be able to show a list of all the services available on the Android operating system as well as the running services. Before creating this Floating Action Button, the list was just laid out after the list of all services. Our team found this to be messy and hard for the user to find. A floating action button was created to switch between each ListView.

1. Put each ListView on top of each other in the xml layout file

2. Create the floating action button and position where you feel fits best within the application.

3. Create a second floating action button on top of the original to be able to switch back and forth between the two ListViews.

4. Once the layout is set up we can head to the ServicesFrag.java class. This is where the ListView ArrayAdapter is presented in the view.

5. Create an onClick method for each button in the ServicesFrag.java class.

6. Set the visibility of the running services ListView to invisible at start up and its associated button.

7. Change the visibility of each ListView that makes sense to present these at only one at a time.

Floating Action Button Visibility for Running Services Code:

```
getRunning.setVisibility(View.INVISIBLE);

getRunning.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        servicesLIST.setVisibility(View.INVISIBLE);
        //running list visible
        runningLIST.setVisibility(View.VISIBLE);
        //complete service list invisible
        getRunning.setVisibility(View.INVISIBLE);
        //running btn caption invisible
        running_caption.setVisibility(View.INVISIBLE);
    }
});
```

10. Killing Services and Processes

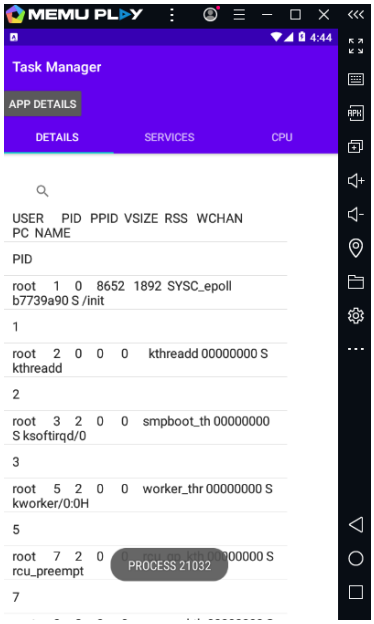
In our application, we wanted to enable the user to be able to kill the processes and services listed in each tab. Our team was unsuccessful in accomplishing this task.

We were able to get the PID by using a substring() method inside the for loop to display the output, but when each item was clicked in the ListView, it was not retrieving the correct item in the list, but only the last one in the list.

Substring PID Code:

```
mySplit = detailsLines[i].substring(10,15);
mySplit = mySplit.replaceAll("//s","");
detailsInfo.add(mySplit);
```

Our team would like to expand on this feature of the application. Although this task may be risky, it would be a great feature to include into this application.



Conclusion

Our application successfully displays the information in tabs similar in the Windows Task Manager. Our application requires root access to gather and execute certain Linux commands. This information pertaining to details, services, and CPU utilization. To gather information for each tab in the application, we executed different Linux commands in super user. After getting the output from these commands as one large string, we broke it up line by line to display the information in a ListView format with an ArrayAdapter. On the details tab being that these were so many processes we gave the user the ability to search the list and find the exact one they want. On the services tab we were able to use two sperate commands to determine which ones are active, so made it possible to just look at active or all services. In our CPU tab, it gives us CPU utilization information as well as the CPU% of each process running. There are many different aspects that we would like to improve on with our application. Our team would like to eventually find a way to be able to kill processes and services. Overall, this application gathers accurate protected operating system information and displays the output in a readable format for the user.