Process Manager

PROJECT 2

Harsh Bhatt | Operating Systems | Panther ID: 002-17-9394

I. Introduction

The process manager application uses key OS management functionalities to assist the user in managing running processes in any way he may like. The high-level description of the application is that it will display a list of running processes in a vertical list view on launch. From this list view, the user will have information regarding the running processes and their associated process identification numbers (PIDs). Then the user can touch which processes she may want to kill, and the processes will essentially "die" in the background. Upon reloading the process, it will be as if the process was restarted.

The application achieves the functionalities of listing running process, and killing the running processes upon the user clicking it. Some sub-functionalities that are important to mention is the running processes also listing their associated PIDs besides them. This is useful for seeing which processes are more important than others.

The application is very closely related to OS-level functionalities because it essentially mimics the job of the OS, but done in a manual way. Usually, it is the OS that deals with process management (creating and deleting processes). However, this app gives the user the chance to manage the running processes in the system for reasons ranging from resource management to power use efficiency.

II. Application Design

The environment of this application is pivotal to the overall functioning of the program. The process manager application will only work as intended on Android versions below 5.1.1. This is because as of Android 5.1.1 and above, getRunningAppProcesses() only returns a list of your own application. This is important because getRunningAppProcesses() should return a list of all running processes in the system, and without it we would not be able to select which processes we want to terminate. Google has severely limited the ease of getting process information in the newer versions of Android. So, to display the OS-level functionality of an android application, and given the time constraints, I found it more useful to use older versions of Android to host this application.

Now that the environment has been explained, let us move onto the building blocks of this application. This application uses two key APIs:

- Process.killProcess(pid) or Process.sendSignal(pid, Process.SIGNAL_KILL)
- ActivityManager.killBackgroundProcesses(PackageName)

The first API, Process.killProcess(pid) is only used to kill our process manager application. This is because although Process.killProces(pid) allows us to request to kill any process based on its PID, the kernel will still impose standard restrictions on which PIDs you are able to kill. Usually, this means that only the process running the caller's packages/application and any additional processes created by that app, and packages sharing a common UID will also be able to kill each other's processes. A user may only be able to kill processes that they own.

The other API, Process.sendSignal(pid, Process.SIGNAL_KILL), can be used interchangeably with Process.killProcess(pid). When this method is called, the signals is generated by the Operating System and sent to the process. Whenever a process receives a signal from the OS it must either handle that signal or immediately die. Signals such as SIGNAL_KILL cannot be handles and result in the immediate death of the recipient process. If we wanted to kill a process we do not have the privileges to kill, i.e. it is not our process, then we must escalate our privileges using sudo (this would require root privileges on the Android device).

For our process manager, since it is the calling application, these APIs can only be used to kill the application itself. However, that is why I have utilized another API for this application called ActivityManager.killBackgroundProcesses(PackageName). This API works by telling the ActivityManager that we want to kill processes associated with a specific package. This API gets around the need for your UID to match the UID of the process because it requires the user to accept the KILL_BACKGROUND_PROCESSES permission. I have done so in my application under AndroidManifest.xml.

```
19      </application>
20      <uses-permission android:name="android.permission.KILL_BACKGROUND_PROCESSES"/>
21
```

This permission signals to the OS that an app has been approved by the user as a task killer. When a task killer wants to kill an app, it tells the OS to do it getting around the problem of only being able to kill processes that you own. This is the same as the kernel killing those processes to reclaim memory. The system will take care of restarting these processes in the future as needed.

Now that I have explained handling how to kill the processes, it would be beneficial to explain how the application has access to list the running processes in the system. The method getRunningAppProcesses() does this task by utilizing the RunningAppProcessInfo interface. This interface is populated with information provided by the operating system. We use the information from getRunningAppProcess() and put it into a list view. To do this, I created a ListAdapter class. The ListAdapter class creates a list with the parameters activity_main context and the list of RunningAppProcess info.

```
16  public class ListAdapter extends ArrayAdapter<RunningAppProcessInfo> {
17      // List context
18      private final Context context;
19      // List values
20      private final List<RunningAppProcessInfo> values;
21      public ListAdapter(Context context, List<RunningAppProcessInfo> values) {
22          super(context, R.layout.activity_main, values);
23          this.context = context;
24          this.values = values;
25      }
```

This class will then override the getView method, which will be used to decide what information we want displayed in our list. We will first make the list be in a row view, and then we will add what text we want to appear in each of the rows. For this process manager application, I have decided to display the package name of the running

processes along with the associated PID.

```
27          @Override
28          public View getView(int position, View convertView, ViewGroup parent) {
29              LayoutInflater inflater = (LayoutInflater)
30                      context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
31              View rowView = inflater.inflate(R.layout.activity_main, parent,  attachToRoot: false);
32              TextView appName = (TextView) rowView.findViewById(R.id.appNameText);
33              int pid = getItem(position).pid;
34              appName.setText("Name: " + values.get(position).processName + ", PID: " + pid);
35              return rowView;
36          }
```

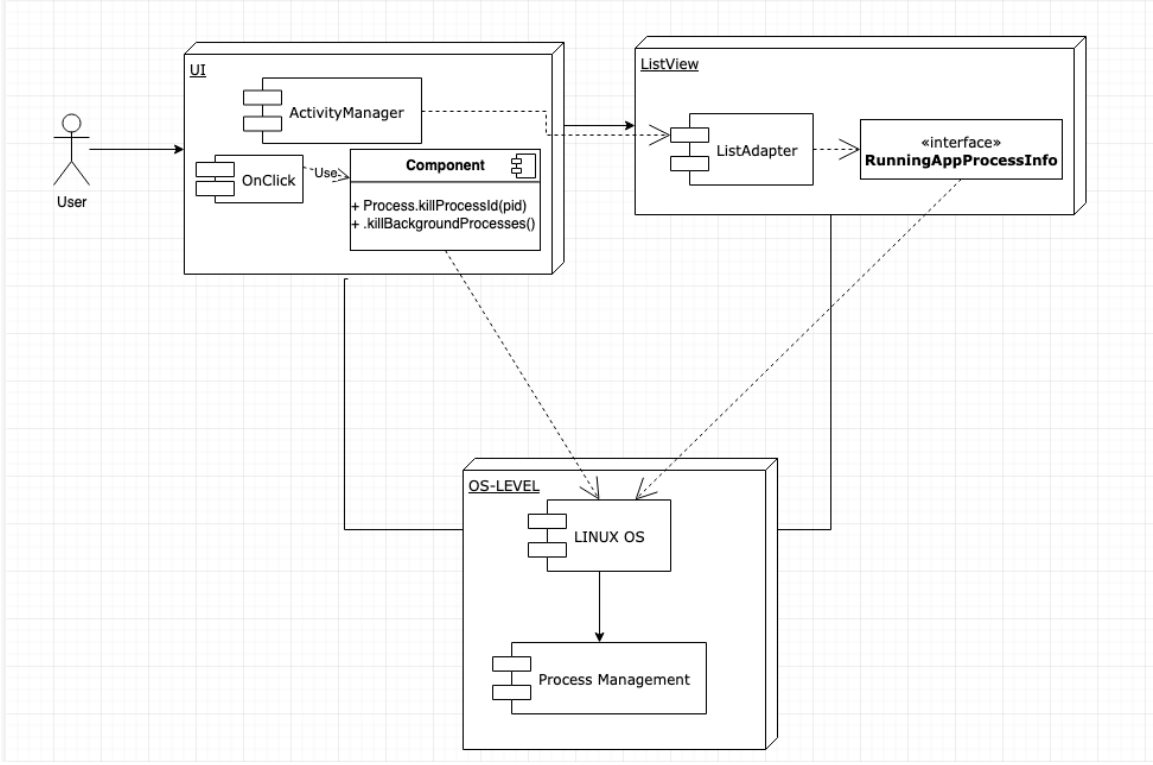
From this we can move onto the MainActivity.java class, which deals with putting all the pieces of the application together. The MainActivity will extend ListActvity since we are presenting a list to the user. Upon loading the application, the ActivityManager will get the running processes by using getRunningAppProcesses() and put that information in the ListAdapter. If there is no application that is running, then there will be a test saying, "No application is running." However, we will never reach this point because the OS always has at least one process running in the background.

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
// Get running processes
        ActivityManager manager = (ActivityManager) getSystemService(ACTIVITY_SERVICE);
        List<RunningAppProcessInfo> runningProcesses = manager.getRunningAppProcesses();
        if (runningProcesses != null && runningProcesses.size() > 0) {
// Set data to the list adapter
            setListAdapter(new ListAdapter( context: this, runningProcesses));
        } else {
// In case there are no processes running (not a chance :))
            Toast.makeText(getApplicationContext(),  text: "No application is running", Toast.LENGTH_LONG).show();
        }
    }
```

When the user clicks on a process, the first step is that the UID associated with that process will be fetched. The UID and the process name will be displayed in a pop-up box and alert the user that the process has been killed, and that upon loading the process again it will be as if the process was restarted. The main code for killing the process will be on line 44 and line 51. The call on line 44 will be to kill any running background processes using the killBackgroundProcess(PackageName) API which allows us to kill processes outside of our UID. The call on line 51 will use the Process.sendSignal() API to kill the calling process, i.e. our process manager application.

```
37          @Override
38          protected void onListItemClick(ListView l, View v, int position, long id) {
39              // Get UID of the selected process
40              int uid = ((RunningAppProcessInfo)getListAdapter().getItem(position)).uid;
41
42              String processName = ((RunningAppProcessInfo)getListAdapter().getItem(position)).processName;
43              ActivityManager activityManager = (ActivityManager) getSystemService(ACTIVITY_SERVICE);
44              activityManager.killBackgroundProcesses(processName);
45
46              // Display data
47              Toast.makeText(getApplicationContext(),  text: "UID " + uid + ", Process Name: " + processName +
48
49             // kill process in same package
50              int pid = ((RunningAppProcessInfo)getListAdapter().getItem(position)).pid;
51              Process.sendSignal(pid, Process.SIGNAL_KILL); // or Process.killProcess(pid);
52          }
53      }
```

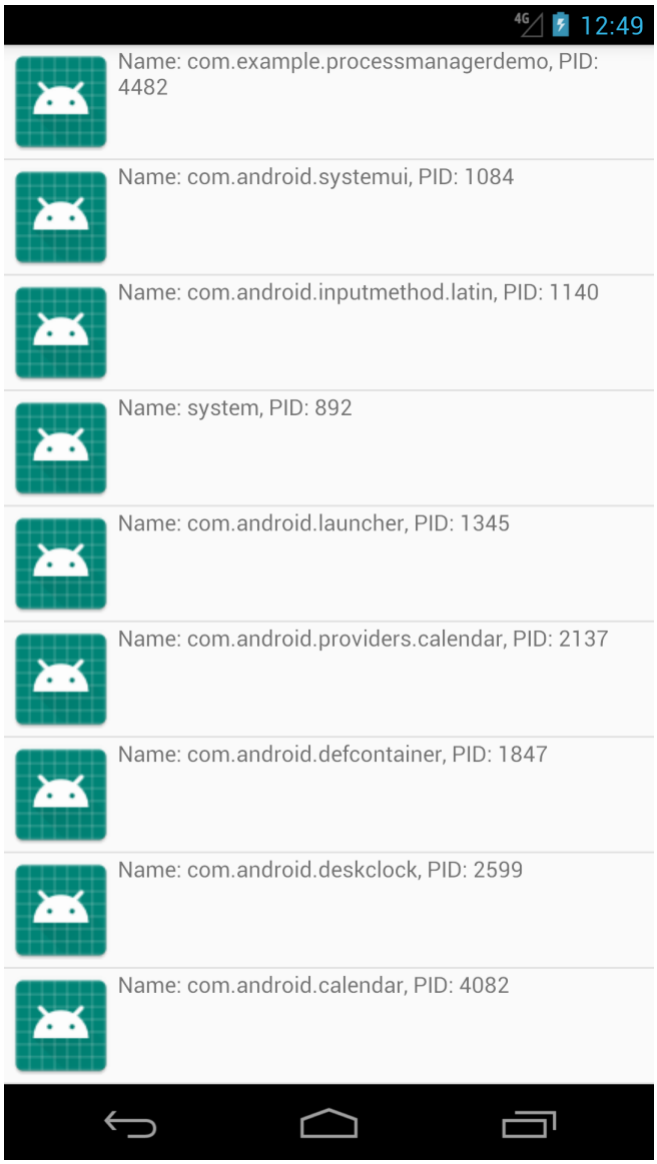
Now that I have explained the different components of the application, it is time to show a design architecture diagram that shows us how everything is connected.



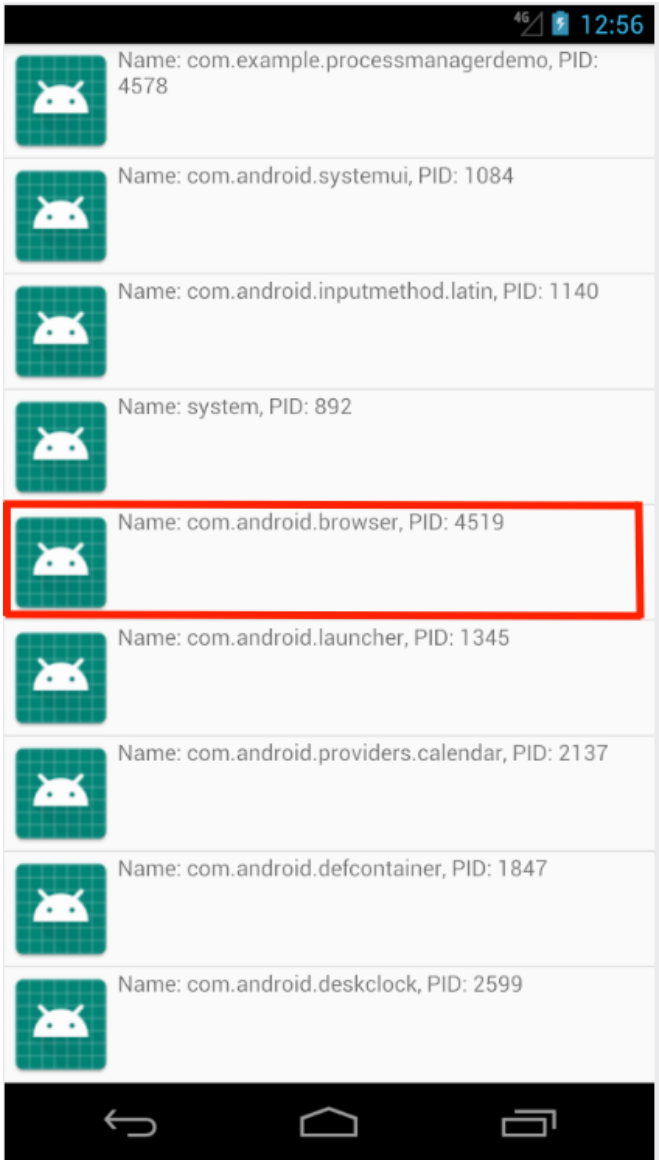
As depicted in the diagram, the entire application is dependent on the OS services. An important note arises from this revelation. The OS is the one that handles killing the processes, not the user. Usually, the users do not "terminate" any process, they just leave the application. No one can kill a process except the OS itself. Most of the task killers in the market do not kill the application, they just restart the process by using 'public void restartPackage(String packageName)'. When this method is called by an activity, the OS will save the state of the activity you wanted to kill and remove that process from memory. Next time the user starts the activity, it will essentially "restart" from where it was killed. Based on that information, it is not wise to use the process manager application by regular users because the OS knows best for what to do regarding process management. However, this application would be useful for those who know what they are doing regarding memory and process management.

The user is only faced with the UI which contains a list view and the option for the user to click on the listed running processes. The ActivityManager component from the UI called on the ListAdapter class from ListView. ListAdapter uses RunningAppProcessInfo interface which is retrieved form the Linus OS itself, specifically the process management component of the Linux OS. The onClick() method in the UI uses the APIs discussed in the report dealing with killing the running processes. These APIs are also directly from the Linux OS's process management component. The Linux OS provides the major functionalities regarding what the application can do.

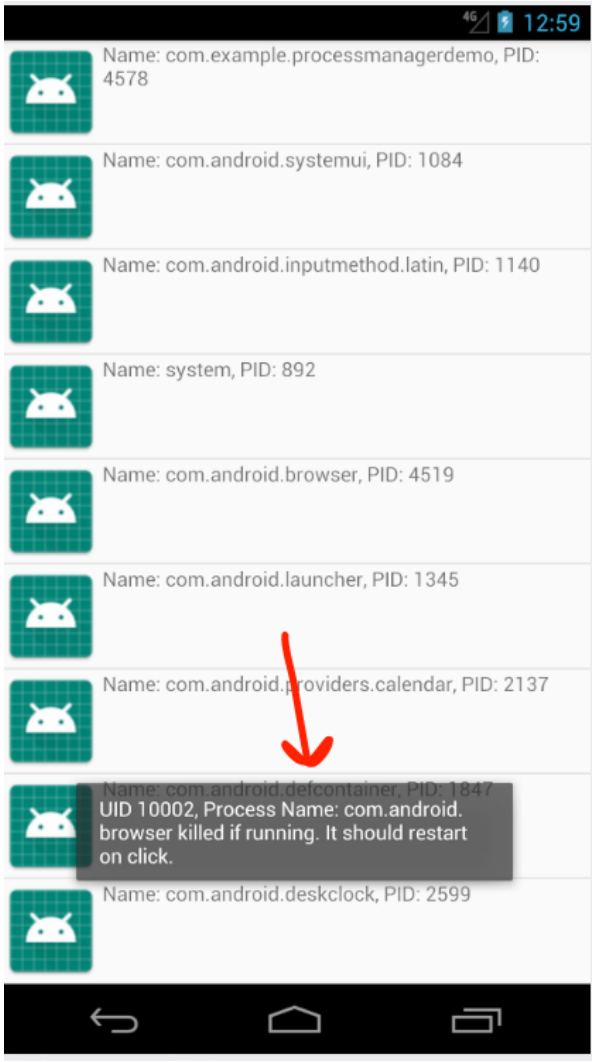
III. User Interface



This is what the application looks like upon loading with no other applications opened by the user. These are the running processes that are on in the background as soon as the device is booted. The very first process listed is the process management application itself. Right next to each process name is the associated PID. The lower the PID, the more important it is in the system.



This is what the application looks like after the user opens the browser application on the device. The name and PID associated with the browser is listed on the process management application.



This is what the application displays after the user clicks on the browser process. As displayed in the alert box, if the user were to open the browser app again, then it will be as if the application was "restarted."

Source Code

From ListAdapter and MainActivity.

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    LayoutInflater inflater = (LayoutInflater)
            context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
    View rowView = inflater.inflate(R.layout.activity_main, parent, false);
    TextView appName = (TextView) rowView.findViewById(R.id.appNameText);
    int pid = getItem(position).pid;
    appName.setText("Name: " + values.get(position).processName + ", PID: " + pid);
    return rowView;
}
```

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
    // Get UID of the selected process
    int uid = ((RunningAppProcessInfo)getListAdapter().getItem(position)).uid;

    String processName =
((RunningAppProcessInfo)getListAdapter().getItem(position)).processName;
    ActivityManager activityManager = (ActivityManager)
getSystemService(ACTIVITY_SERVICE);
    activityManager.killBackgroundProcesses(processName);

    // Display data
    Toast.makeText(getApplicationContext(), "UID " + uid + ", Process Name: " +
processName + " killed if running. It should restart on click.",
Toast.LENGTH_LONG).show();

   // kill process in same package
    int pid = ((RunningAppProcessInfo)getListAdapter().getItem(position)).pid;
    Process.sendSignal(pid, Process.SIGNAL_KILL); // or Process.killProcess(pid);
}
```