ELEN E4901
Mobile Application Development with Android

Project 2

Name: Jingyi Yuan

UNI: jy2736

Email: jy2736@columbia.edu

Name: Yingqi Wang

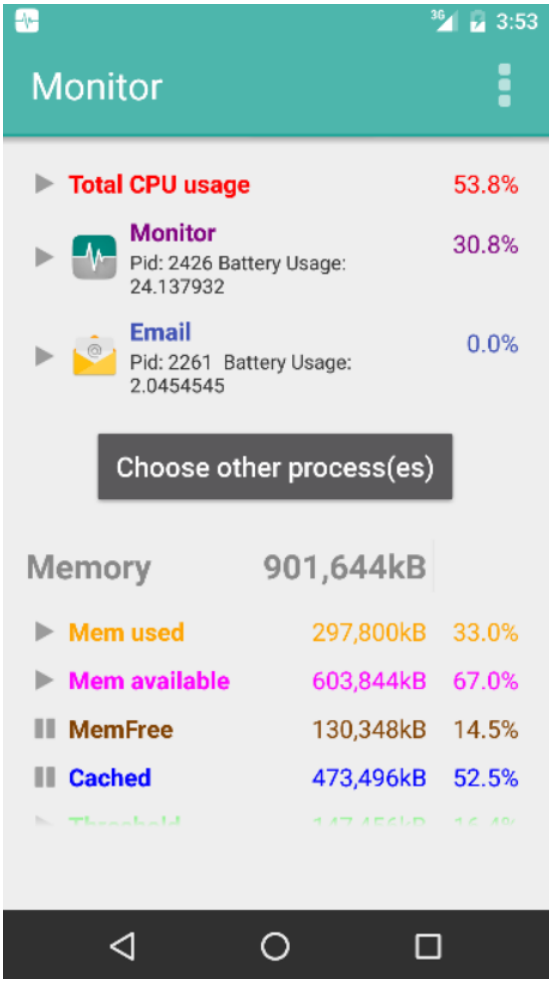
UNI: yw2810

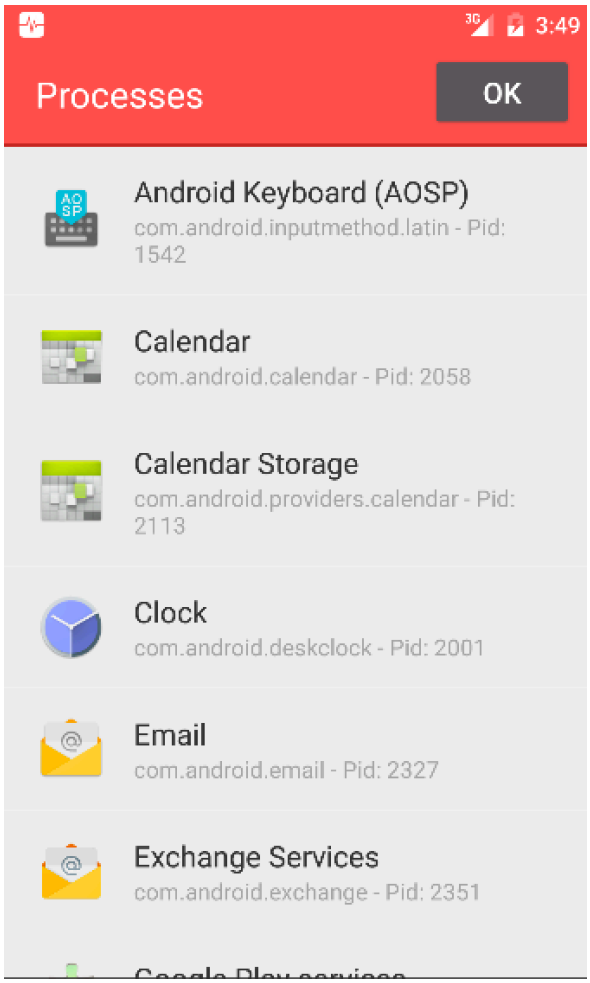
Email: yw2810@columbia.edu

Phase 1: Project Interface

The main interface is shown as left. There are four parameters: Mem used, Mem available, MemFree, and Cached.

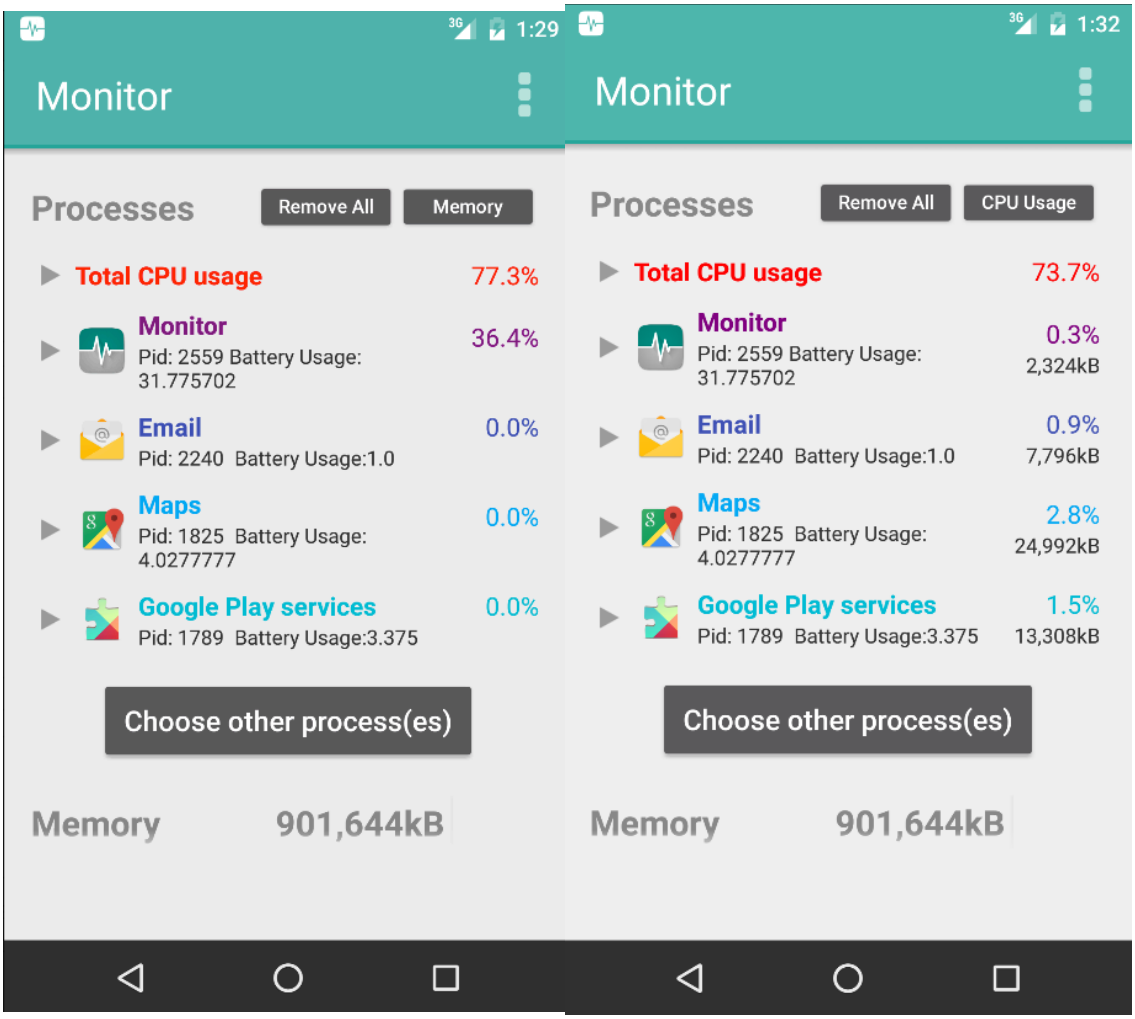
The right is that when we click the button ”Choose other process(es)”, there is another activity to show the running processes and we can choose some of them then they will be shown on the interface. The memory and CPU usage will change and we will analyze in phase 2.



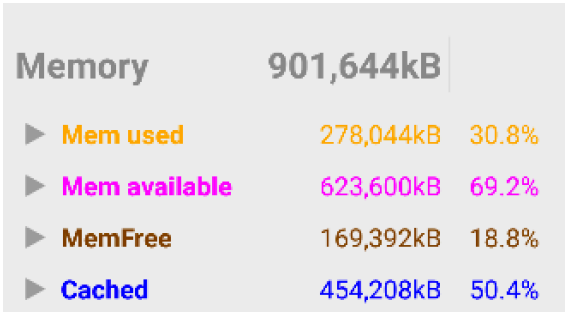


“Mem used” is the memory that already be used by the current processes. We use mSR to get the four parameters. “Mem available” is the only one that matters. Available shows what is capable of being used by Programs. It is a combination of both Cache and Free.

After choosing other processes, we can get pictures shown below. The Memory is shown as left and CPU usage shown as right.

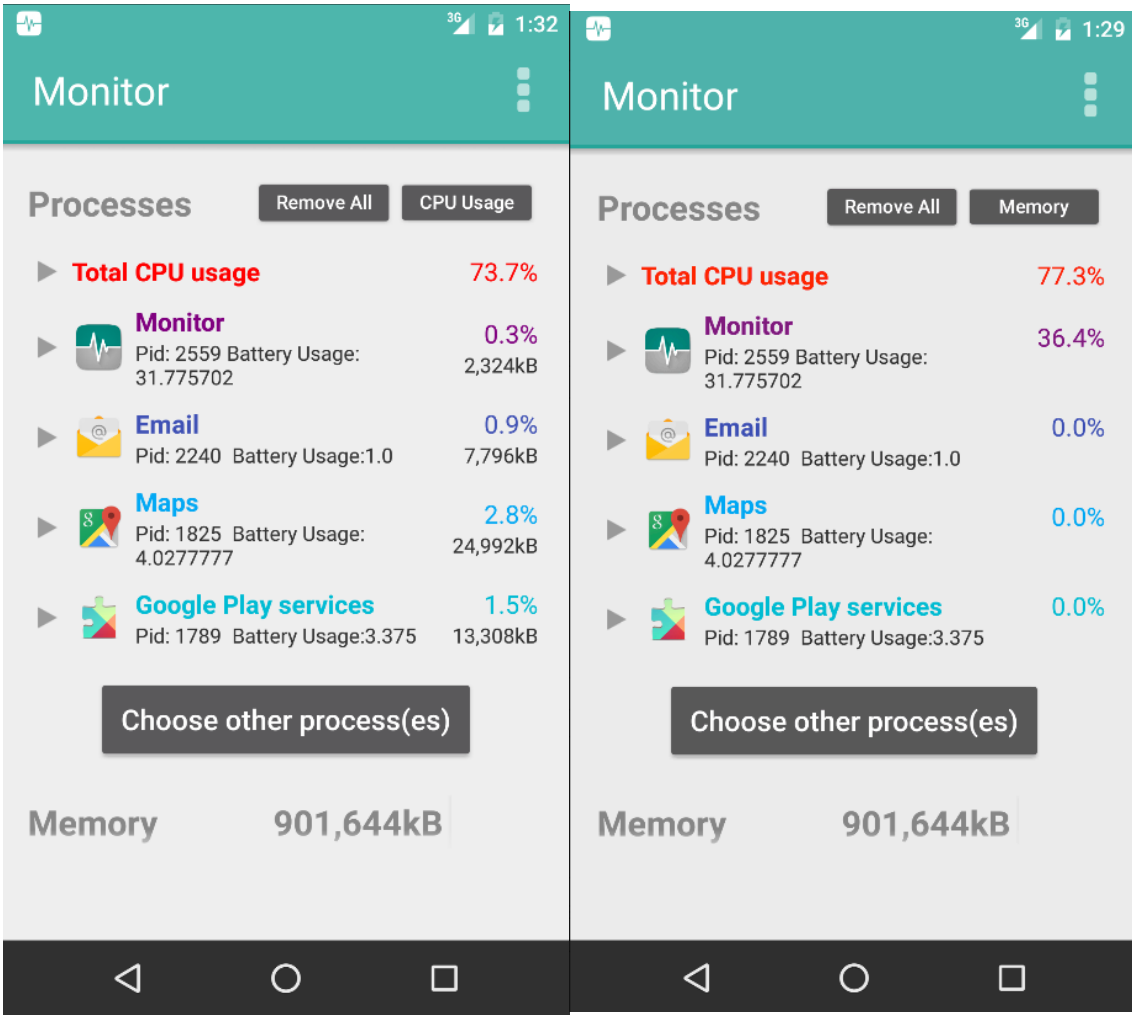


The detail of the parameters is shown below:



Phase 2: Analysis

Following reasults are running on an emulator Nexus 5 , API 23, X86 using Android 6.0.



Here are two print screens of the app we designed. As you can see the CPU usages of the following tasks: Monitor, Email, Maps, Google Play Services have been calculated and listed in our app, as well as the memory usages of these tasks. These information are retrieved from the process information pseudo-file system which contains runtime system information in the path "/proc/stat". Battery usage is calculated by approximation, we collect the CPU usage in a small amount of time and calculate the ratio of CPU usage for a specific tasks.

Let's analyze the information given about these processes one by one.

1. Monitor

Monitor is the app that are running to supervise those process information in real-time. It's the one that with active activities and using the screen, so it uses much of the CPU resources (36.4% from our print screen). And the battery usage is high for the same reason.

2. Email
   Email is the application running in the background to support Email fetching. And at this point we haven't logged into any email account so it doesn't provide much service. So the CPU usage is relatively low(0% in this case), and so is the battery consumption(only 1%).

3. Maps
   Maps is the application running in the background to provide information about user physical location. It is hibernated in the background so that it doesn't use any CPU resources. But it does use a lot of memory (like 2.8% of the total memory which is higher than the monitor and Email), so it contributes to a higher battery consumption (4.027 in this case). The battery usage may also come from the hardware it uses, such as GPS module.

4. Google Play Service
   Google Play Service is running in the background to provide services about google account. Same as Maps, it is hibernated in the background so it doesn't take any CPU usage. But it uses some memory (like 1.5% of the total memory which is higher than Email and lower than Maps), so the battery usage is between that of Email and Maps(3.375 in this case).

5. Matrix Multiplication
   We run a matrix multiplication in the MainActivity and manager CPU using the logcat. We use CpuManager to get the information of CPU and battery by modifying measurePowerUsage in CpuManager. In this process, we run a multiplication of two N*N matrixes. We add it into MainActiviy.java. Every time we click the button "Memory/CPU usage", we run this 200*200 matrixe multiplication and return the running time.

```
I/Serial result：  1600000000
I/Serial running time：  1003
D/dalvikvm：  GC_CONCURRENT freed 1442K, 14% free 12954K/15047K, paused 12ms+12ms, total 40ms
I/BatteryInfo：  appCpuTimeAfter 486.0
I/BatteryInfo：  appCpuTimeBefor  13.0
I/BatteryInfo：  totalTimeAfter 172564.0
I/BatteryInfo：  totalTimeBefor 168322.0
I/BatteryInfo：  ratio0.2814108674928503
I/BatteryInfo：  currentSteps.length 10
```

Here we use first method serial. We calculate each $c_{ij} = a_{ik} * b_{kj}$ by the definition of matrix multiplication using three levels of loop:

```
public void serial() {  //first method
    int i,j,k;
    for( i=0; i< end; i ++)
    {
        for( j=0;j<end;j++)
        {
            C[i][j]=0;
            for( k=0; k< end;k++)
            {
                C[i][j]+=A[i][k]*B[k][j];
            }
        }
    }
    for( i=0; i<end; i++)
        for( j=0; j<end; j++)
            sum += C[i][j];
}
```

We use the class MatrixMultiplicion to do and we get the Serial running time is 1003.

```
//--------------------Serial--------------------
startTime = System.currentTimeMillis();
MatrixMultiplication Serial = new MatrixMultiplication(a, b, 0, leng);
Serial.serial();
endTime = System.currentTimeMillis();
long result = Serial.getSum();
Log.i("Serial result", String.valueOf(result));
Log.i("Serial running time", String.valueOf(endTime-startTime));
return endTime - startTime;
```

```
I/Serial result：  1600000000
I/Serial running time：  1003
```

There are other processes running which are more or less like the scenarios of Maps and Google Play Service. So they are left out.

Phase 3: Advance

Method 1: Optimization by killing processes

Considering situations for those useless processes running in the background which use much memory and do nothing. We can set a timer for those processes (such as 30 seconds or so), if the timer has counted to 30 seconds and the processes are not waken(means they do nothing useful for the user), we can kill these processes to vacant those memory, and reduce battery consumption.

Method2: Optimization by Using Parallel Threads.

For the matrix multiplication, to improve the running time and reduce the consumption, we use a second method, two parallel threads.

A Thread is a concurrent unit of execution. It has its own call stack for methods being invoked, their arguments and local variables. Each application has at least one thread running when it is started. Here we use two threads to run this matrix multiplication parallel, with each work (N/2)*(N/2)

```
//--------------------Parallel--------------------
MatrixMultiplication thread1 = new MatrixMultiplication(a, b, 0, leng);
MatrixMultiplication thread2 = new MatrixMultiplication(a, b, 1, leng);
startTime = System.currentTimeMillis();
thread1.start();
thread2.start();
thread1.join();
thread2.join();
endTime = System.currentTimeMillis();
long result = (thread1.getSum() + thread2.getSum());
Log.i("Thread result", String.valueOf(result));
Log.i("Thread running time", String.valueOf(endTime-startTime));
return endTime - startTime;
```

And we get Thread running time, which is approximately half of the serial running time.

```
I/Thread result: 1600000000
I/Thread running time: 551
D/dalvikvm: GC_CONCURRENT freed 1782K, 16% free 12927K/15367K, paused 1ms+3ms, total 21ms
I/BatteryInfo: appCpuTimeAfter 450.0
I/BatteryInfo: appCpuTimeBefor  14.0
I/BatteryInfo: totalTimeAfter 191258.0
I/BatteryInfo: totalTimeBefor 186962.0
I/BatteryInfo: ratio0.2036311120280586
I/BatteryInfo: currentSteps.length 10
```

To sum up, we use two threads rather than serial to run the matrix multiplication thus the work of each thread is deducted to (N/2)*(N/2) rather than N*N. This method uses synchronization by two parallel threads in order to reduce the data consumption.