Sistemas Operacionais I N

Estudo de Caso: Políticas de Processos no Android

Wellington Espindula

Setembro de 2021

1 Implementação de Processos e Escalonador

O Sistema Operacional Android - mantido pela Open Handset Alliance e pela Google LLC - é o sistema operacional *mobile* mais utilizado em todo o mundo, sendo executado nos mais diversos tipos de dispositivos, tais como Smartphones, Tables, Smartwatches e Smart TV's.[1] Para tanto, este é composto de um *kernel* desenvolvido pelo próprio Android, conhecido como Android Commom Kernel (ou ACK), que é, de forma simples, o *kernel* comum para todos os dispositivos. Este posteriormente pode ser modificado pelos fornecedores e para o dispositivo, ativando periféricos e *drivers* adicionais e personalizações. O ACK, por sua vez, é um *fork* do *kernel* Linux LTS. Na sua versão mais nova, Android 12, a *release* utiliza o *kernel* Linux 5.2 e 5.10.[2] Na figura 1, pode-se visualizar a hierarquia de desenvolvimento do *kernel*.

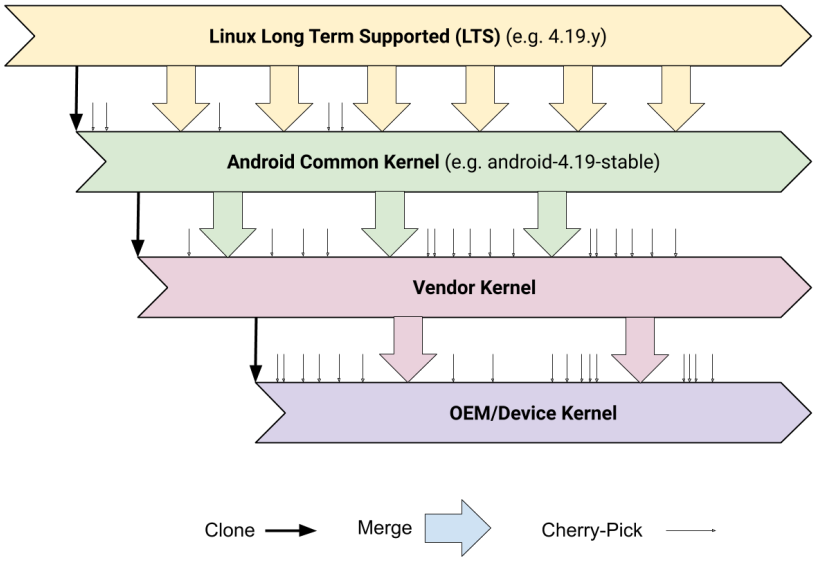


Figura 1: Hierarquia do desenvolvimento do *Kernel* Android. Fonte: Android Source (2021). Disponível em: ⟨https://source.android.com/devices/architecture/kernel/generic-kernel-image⟩.

Por definição, um processo é uma instância de um programa em execução; deste modo, um papel principal dos sistemas operacionais é fornecer o suporte - através de rotinas - para criação, execução e gerenciamento os processo. Do ponto de vista de um sistema operacional, um processo pode ser representado através de uma estrutura de dados formado por, entre outros atributos, seu PID (identificador de processo), uma árvore Heap - que irá ser responsável por alocação de memória dinâmica -, uma pilha - que armazenará, por exemplo, variáveis locais e

endereços de retorno de sub-rotinas - e informações de estado - Estado Atual, *Program Counter*, Dados de Registradores.[3] Dito isso, existem diferentes algoritmos que implementam políticas de escalonamento do processos. O Android, por ter um *kernel* derivado do Linux, apresenta o mesmo escalonador que este, de tal modo que o escalonador do Linux é incluído no *kernel* do Android.[4]

O *Kernel* Linux, embora implemente diferentes políticas de escalonamento (como o *Round-Robin* com *Aging*), introduziu em sua versão 2.6.23 (2007) o algoritmo de escalonamento *Completely Fair Schedule* (CFS) e internamente o CFS é o escalonador padrão, definido no cabeçalho com a macro `SCHED_NORMAL` .[5, 6] O *Round-Robin* e o CFS são ambos algoritmos de escalonamento preemptivos, o que significa que um processo pode ser interrompido para dar lugar à outro com a finalidade de maximizar a vazão de processos executados e o uso de CPU. O CFS modela um sistema multi-tarefas ideal e traz como benefícios a equidade de processos e a melhora da fluidez do sistema. Como contrapartida, o CFS tem complexidade $O(n)$ para encontrar o próximo processo.[5]

2 Processos Android e Ciclo de Vida

A partir do Android 5.0, cada aplicativo Android é executado em um próprio processo Linux através de uma instância do *Android Runtime* (ART). O processo é criado assim que alguma parte de seu código precisa ser executado e será mantido em execução até que este não seja mais necessário e o sistema reivindique a sua memória para executar outras aplicações.[7]

Uma aplicação Android pode utilizar três tipos de componentes de aplicação. São esses ***(i)*** **Activity** - Componente de Aplicação responsável por prover uma interface na qual usuários podem interagir com; ***(ii)*** **Service** - Componente responsável por execução de operações de longa duração em plano de fundo, muito embora existam *Foreground Services* que executam serviços em plano de fundo de forma notificável ao usuário; e ***(iii)*** **BroadcastReceiver** - Receptor de eventos de sistema, tais como o `BOOT_COMPLETED` , `CALL` e `BATTERY_LOW` .

A fim de determinar um nível de hierarquia de prioridade entre os processos, o Android determina os seguintes tipos de processos:[8]

1. **Processo em Primeiro Plano:** também conhecido como *Foreground*, é aquele processo que é necessário para as atividades correntes do usuário e pode ser uma *Activity* executando na parte visível da interface gráfica, um *BroadcastReceiver* que está rodando o código de recebimento de evento e um *Service* executando seu código de inicialização/destruição.

2. **Processo Visível:** realiza tarefas que mesmo não estejam em primeiro plano ou em execução direta, mas que o usuário está consciente de sua execução, como, por exemplo, quando uma *Activity* foi pausada, quando existe uma *Foreground Service* em execução ou quando algum serviço específico de sistema (desde que o usuário esteja ciente) está executando tal como um papel de parede *live*.

3. **Serviço:** processos cuja execução não é visível para o usuário, muito embora sua execução seja importante para as aplicações. Alguns exemplos são serviços de sincronização e compressão de dados.

4. **Processo em Cache:** processo que não é mais necessário, mas que fica em *cache* para execução mais rápida.

Um ponto diferencial quando se fala em um sistema operacional *mobile* em relação aos sistemas operacionais *desktops* é que os recursos são bem mais escassos e este também acaba

sendo responsável por gerenciar os recursos de forma a otimizá-los para os diferentes contextos dos usuários. Temos como exemplo de recursos a memória e a bateria de um *smartphone* que o SO precisa gerenciar com muita cautela, habilitando e desabilitando serviços em diferentes cenários, a fim de maximizar a experiência do usuário.

Por conseguinte, esta hierarquia faz-se importante não apenas a nível de escalonamento de processos, mas também em termos de distribuição de recursos. Portanto, ela também é utilizada quando é necessário determinar qual processo precisa ser matado em condições de baixa memória disponível.[8]

2.1 Do ponto de vista do Desenvolvedor de Sistemas

Assim como supracitado ao longo do texto, cada aplicação Android roda seu próprio processo. Então o desenvolvimento de sistemas em Android perpassa sempre o Android SDK; este, através de suas APIs, dará a interface para a criação e gerenciamento de processos dentro de um ponto de vista de um desenvolvedor. Vale relembrar e ressaltar também que as aplicações executam dentro do ART. Através dessa perspectiva, nota-se que o desenvolvedor não mais poderá realizar chamadas de criação de processos e comunicação intra-processual de forma nativa, mas, sim, precisará de um respaldo do ambiente Android para tanto - que se encarregará de realizar as chamadas nativas.

Em seguida, portanto, explorar-se-á como se pode desenvolver os diferentes tipos de processo por componente de aplicação e como se dá cada um dos ciclos de vida.

2.1.1 Componentes de Aplicação

Activity A *Activity* é um componente responsável por interação com o usuário. Para a sua implementação, necessita-se criar uma especialização da classe - ou de uma das subclasses - `android.app.Activity` .[9] Desse modo, ao criar uma activity, o código gerado, em Java, ficaria da seguinte forma:

```
public class MainActivity extends Activity {
    // Aqui voce define os campos os componentes de interfaces
    private Button mButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Aqui se poe o layout (xml) que essa activity correspondera
        setContentView(R.layout.activity_main);
        // Assignment dos campos com base nos componentes do layout
         mButton = findViewById(R.id.button);
    }

    @Override
    protected void onDestroy(){}
    ...
}
```

Existem diversos outros métodos que a classe *Activity* permite a sobrescrita[10], mas no código trouxe apenas alguns métodos relacionados ao ciclo de vida desta. O ciclo de vida está bem descrito na imagem 2.

O `onCreate()` é chamado quando o sistema cria a sua atividade, o `onDestroy()` seria seu oposto, ou seja, quando o sistema destrói a atividade; o `onStart()` quando a atividade entra no modo Start e se torna visível para o usuário, em contrapartida o `onStop()` é chamado quando a atividade não está mais visível e o `onRestart()` quando a atividade volta à sua visibilidade; o `onResume()` é chamado quando o usuário pode interagir com a atividade e o `onPause()` logo quando a atividade perde o foco.[9, 10]

Além disso, conforme mencionado no código, quando uma *Activity* é desenvolvida, normalmente, também desenvolve-se um layout - escrito em XML - para a UI desta.

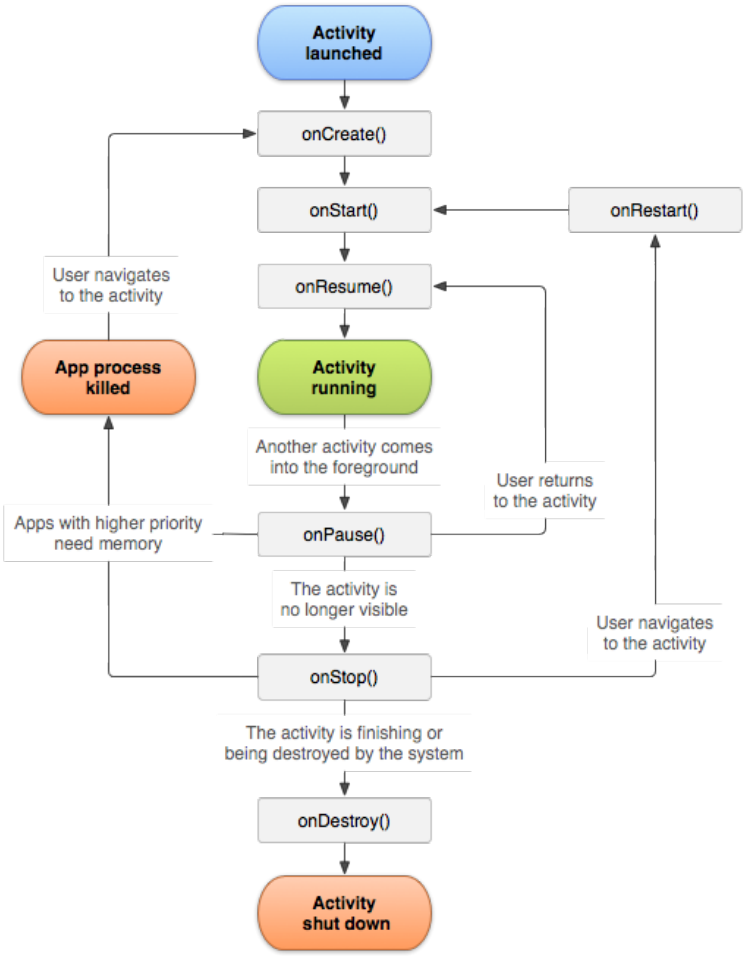


Figura 2: Ciclo de vida de uma *Activity*. Fonte: Android Developer (2021). Disponível em: ⟨https://developer.android.com/reference/android/app/Activity⟩.

Service *Services* são componentes muito relevantes quando se trata de realizar tarefas em plano de fundo. Existem três tipos de *Services*:[11]

- *Foreground: Services* em que usuários são notificados que estão em execução. Geralmente acompanham uma notificação na tela. Alguns exemplos de uso comuns são o backup diário do *Whatsapp* ou a sincronização do Gmail para Android. Isto é, são úteis para atividades de segundo plano que necessitam de prioridade e da atenção do usuário.

- *Background:* Serviço que roda sem notificação do usuário. Muito embora seja útil para muitos casos, a documentação oficial desencoraja seu uso devido uma série de restrições de processamento em plano de fundo.

- *Bound:* Vincula-se este serviço à outro componente através de uma interface de interação inter-processos permitindo assim a interação entre esses, por exemplo, para enviar requisições e receber resultados.

A implementação de um Serviço é muito similar à de uma Atividade. Deve-se estender a classe abstrata `android.app.Service` . Por ser uma classe abstrata, deve-se sobrescrever necessariamente o método `onBind(Intent)` que servirá para realizar as interações entre componentes. Para os demais casos, sobrescreve-se os métodos de ciclo de vida do Serviço conforme a necessidade.[12]

```
public class MyService extends Service {
    @Override
    public IBinder onBind(Intent intent) {}

    @Override
    public void onCreate() {}

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {}
}
```

O ciclo de vida de um *Service* está representado na imagem 3. O método `onCreate()` é chamado quando o sistema cria o *Service* e o `onDestroy()` quando ele o destrói. O `onStartCommand()` é executado quando um *Service* recebe um comando para começar sua execução `Context.startService(Intent)` . Este fica executando até parar ou receber um sinal de parada. Por fim, o `onBind(Intent)` é recebido quando algum outro componente deseja se vincular com este serviço, através do `Context.bindService(Intent)` .[12]

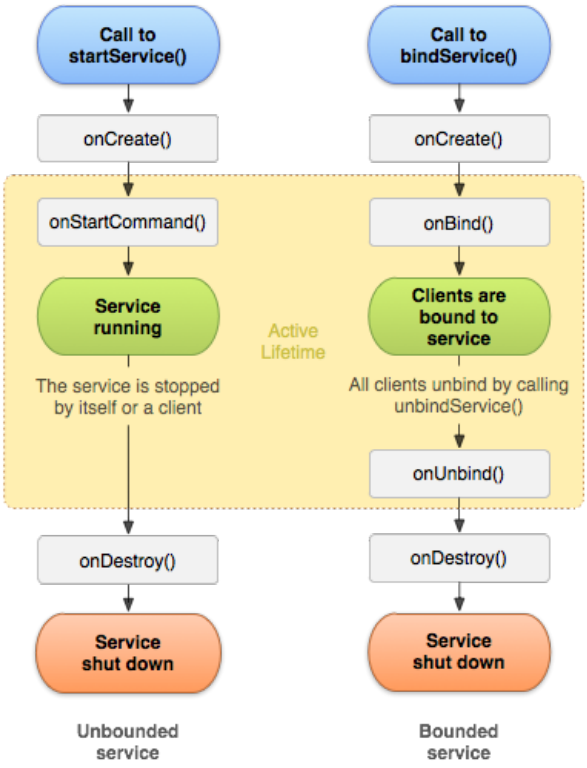


Figura 3: Ciclo de vida de um *Service*. Fonte: Android Developer (2021). Disponível em: ⟨https://developer.android.com/guide/components/services⟩.

BroadcastReceiver O *BroadcastReceiver* é o mais simples de todos os componentes. Ele irá entrar em execução quando algum evento acontecer. Um *BroadcastReceiver* pode ser utilizado para agendar e receber alarmes de sistema ou para eventos do dispositivo como `BOOT_COMPLETE` .[13]

A implementação do *BroadcastReceiver* envolve a especialização da classe abstrata `android.content.BroadcastReceiver` . Dessa forma, faz-se necessário a implementação do método `onReceive(Context, Intent)` .[13]

```
public class MyReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
    }
}
```

2.1.2 Declaração de Componentes de Aplicação

A fim de utilizar os componentes de aplicação supra-citados, faz-se necessário declará-los através do *Manifest* da aplicação. No exemplo abaixo, pode-se notar a presença dos 3:

```
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <application
        <activity android:name=".MainActivity"
                  android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

         <service android:name=".MyService" android:enabled="true"
                android:exported="true" android:process=".myownprocess">
        </service>

        <receiver android:enabled="true" android:name=".MyReceiver"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED"/>
            </intent-filter>
        </receiver>
    </application>
```

Neste exemplo, temos uma *Activity* que será a tela principal da aplicação e executar-se-á na inicialização da aplicação, temos um *Service* que tem o atributo `android:process` que irá indicar que ele terá um processo separado do processo padrão da aplicação principal e teremos um *Receiver* que irá ser executado assim que o Sistema Operacional finalizar sua inicialização. O atributo `android:process` , embora tenha sido utilizado no exemplo em uma *Service*, também pode ser utilizado em *Activities* e em *BroadcastReceivers*.[10, 12, 13, 14, 15, 16]

2.1.3 Comunicação Inter-Processos

A comunicação inter-processos pode ser dada através de três mecanismos principais:[17]

1. *Intents* são mensagens que um componente pode utilizar para receber e enviar dados e

ações, isto é, um mecanismo universal de passagem de dados entre processos. *Intents* utilizam *Bundles* dentro deles para a serialização dos dados passados.[18]

2. *Bundles* são entidades de dados transmitidos, muito similares à serialização de um objeto.[19]

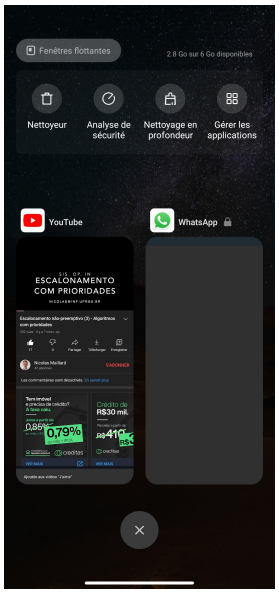
3. *Binders*, já citado, permitem que processos se interliguem através de referências à outros processos.[20]

2.1.4 Classe Process

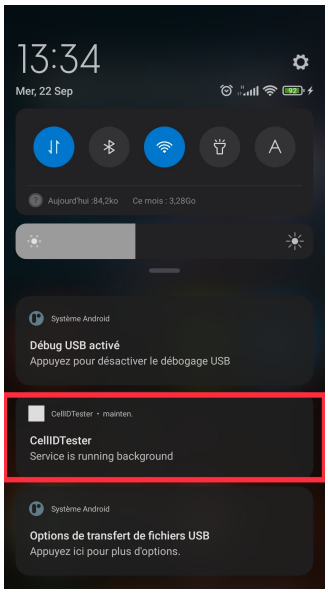
A Classe `android.os.Process` do Android SDK dá aos desenvolvedores um conjunto de métodos para os desenvolvedores sobre informações de processos. Dentre as funções que oferece, temos `Process.myPid()` e `Process.getElapsedCpuTime()` que, respectivamente, fornece o PID (Identificador de Processo do processo em execução) e o tempo que o processo corrente está em execução.[21]

2.2 Do ponto de vista do Usuário

O usuário pode ver, bem como destruí-las com um simples gesto, as atividades que estão em execução através do menu de aplicações recentes, como na figura 4(a). Além disso, ele também pode verificar quais serviços em primeiro-plano estão executando, figura 4(b).



(a) Aplicativos em Execução.



(b) Serviço de primeiro plano em execução.

Figura 4: Aplicativos e Serviço de Primeiro-Plano em execução. Fonte: O Autor (2021).

3 Monitoramento de Processos

Enquanto sistema baseado em Linux e, portanto, um sistema Unix, o Android tem diferentes formas de monitorar e visualizar processos através de ferramentas Unix. Em ambientes Unix, a leitura do arquivo virtual `/proc/stat` e o comando `top` - e suas variações - são abordagens clássicas para observar as estatísticas de uso de processos.

Essas abordagens também são possível no Android através do uso de uma ferramenta chamada de `adb` - *Android Debug Bridge* -, que é muito comum quando se trata de desenvolvimento. O adb disponibiliza diversos recursos para comunicação com o dispositivo tal como executar comandos em um ambiente de shell diretamente no dispositivo através da execução de `adb shell` .

Muito embora, o *adb* necessite uma permissão específica no dispositivo e seja voltado para desenvolvedores, nele é possível fazer esse *hard use* e utilizar as ferramentas Unix de monitoramento de processos. Nas figuras 5 e 6, verifica-se, respectivamente, a execução do `cat /proc/stat` e do comando `top` no adb shell.

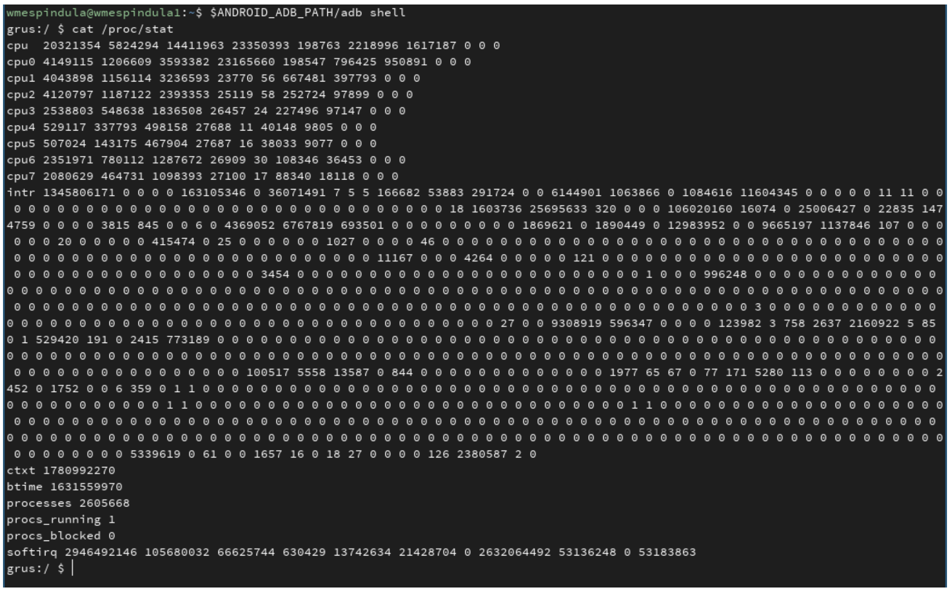


Figura 5: Execução do comando “cat /proc/stat” no adb shell. Fonte: O Autor (2021).

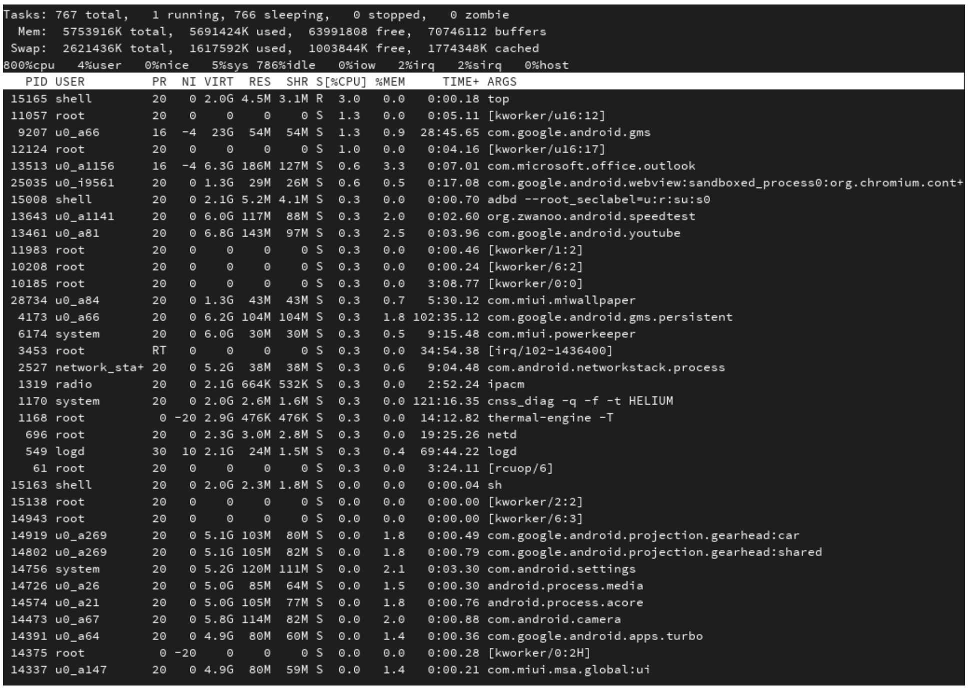


Figura 6: Execução do comando “top” no adb shell. Fonte: O Autor (2021).

Referências

1 WIKIPÉDIA. *Android — Wikipédia, a enciclopédia livre*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://pt.wikipedia.org/w/index.php?title=Android&oldid=61738499⟩.

2 GOOGLE; ALLIANCE, O. H. *Android Source: Generic Kernel Image*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://source.android.com/devices/architecture/kernel/generic-kernel-image⟩.

3 SILBERSCHAT, A. *Fundamentos de sistemas operacionais (9a. ed.)*. [S.l.]: Grupo Gen - LTC, 2015. v. 1. ISBN 978-85-2163-000-5.

4 GOOGLE; ALLIANCE, O. H. *Android Source: sched.h*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://android.googlesource.com/kernel/common/+/refs/heads/android12-5.4-lts/include/linux/sched.h⟩.

5 JOSHI, K. R. *Process Scheduling II*. 2013. Acessado em: 21 de setembro de 2021. Disponível em: ⟨http://www.cs.columbia.edu/~krj/os/lectures/L12-LinuxSched.pdf⟩.

6 WIKIPÉDIA. *Linux kernel — Wikipédia, a enciclopédia livre*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://en.wikipedia.org/wiki/Linux_kernel#Scheduling_and_preemption⟩.

7 GOOGLE; ALLIANCE, O. H. *Android Developers: Android Runtime*. 2020. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/platform?hl=en-us#art⟩.

8 GOOGLE; ALLIANCE, O. H. *Android Developers: Processes and Application Lifecycle*. 2020. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/components/activities/process-lifecycle⟩.

9 GOOGLE; ALLIANCE, O. H. *Android Developers: Introduction to Activities*. 2019. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/components/activities/intro-activities⟩.

10 GOOGLE; ALLIANCE, O. H. *Android Developers: Activity*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/reference/android/app/Activity⟩.

11 GOOGLE; ALLIANCE, O. H. *Android Developers: Services overview*. 2019. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/components/services⟩.

12 GOOGLE; ALLIANCE, O. H. *Android Developers: Service*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/reference/android/app/Service⟩.

13 GOOGLE; ALLIANCE, O. H. *Android Developers: Broadcasts overview*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/components/broadcasts⟩.

14 GOOGLE; ALLIANCE, O. H. *Android Developers: <activity>*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/topics/manifest/activity-element#proc⟩.

15 GOOGLE; ALLIANCE, O. H. *Android Developers: <service>*. 2020. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/topics/manifest/service-element#proc⟩.

16 GOOGLE; ALLIANCE, O. H. *Android Developers: <receiver>*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/topics/manifest/receiver-element#proc⟩.

17 IVANOV, V. *What are the IPC mechanisms available in the Android OS?* 2011. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://stackoverflow.com/questions/5740324/what-are-the-ipc-mechanisms-available-in-the-android-os⟩.

18 GOOGLE; ALLIANCE, O. H. *Android Developers: Intent*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/reference/android/content/Intent⟩.

19 GOOGLE; ALLIANCE, O. H. *Android Developers: Bundle*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/reference/android/os/Bundle⟩.

20 GOOGLE; ALLIANCE, O. H. *Android Developers: IBinder*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/reference/android/os/IBinder⟩.

21 GOOGLE; ALLIANCE, O. H. *Android Developers: Process*. 2021. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/reference/android/os/Process⟩.

22 GOOGLE; ALLIANCE, O. H. *Android Developers: Processes and Threads*. 2020. Acessado em: 21 de setembro de 2021. Disponível em: ⟨https://developer.android.com/guide/components/processes-and-threads⟩.