4.
This is the architecture of Android.
Android runs on top of a vanilla kernel with little modifications, which are called Androidisms.

The native userspace, includes the init process, a couple of native daemons, hundreds of native libraries and the Hardware Abstraction Layer (HAL).

The native libraries are implemented in C/C++ and their functionality is exposed to applications through Java framework APIs.

A large part of Android is implemented in Java and until Android version 5.0 the default runtime was Dalvik. Recently a new, more performant runtime, called ART has been integrated in Android (first made available as an option in version 4.4, default runtime from version 5.0).

Java runtime libraries are defined in java.* and javax.* packages, and are derived from Apache Harmony project (not Oracle/SUN, in order to avoid copyright and distribution issues.), but several components have been replaced, others extended and improved. Native code can be called from Java through the Java Native Interface (JNI).

A large part of the fundamental features of Android are implemented in the system services: display, touch screen, telephony, network connectivity, and so on. A part of these services are implemented in native code and the rest in Java. Each service provides an interface which can be called from other components.

Android Framework libraries (also called “the framework API”) include classes for building Android applications.

On top, we have the stock applications, that come with the phone. And other applications that are installed by the user.

6.

- Most Linux distributions take the vanilla kernel (from Linux Kernel Archives) and apply their own patches in order to solve bugs, add specific features and improve performance.
- For Android is the same thing, they take the vanilla kernel, apply some hundreds of patches that bring improvements, solve bugs and add specific features for the devices that run Android.
- In a previous lecture, we discussed about the Android Mainlining Project and Android Upstreaming that deal with including these features into the mainline. Some of these Androidisms have been already included in the mainline. Next, I will present the most important of these Androidisms: Wakelocks, Low Memory Killer, Anonymous Shared Memory, Alarm, Logger and Binder.

7.

- On desktops and laptops, the user decides when the systems enters into the suspend mode and when it wakes up. For example, when you close the lid of a laptop running Linux, it will go into suspend or sleep mode. In this mode, the state is saved in RAM, but the other hardware components are shut down in order to reduce the power consumption to maximum. When the lid is raised, the laptop wakes up and has the exact state.
- In contrast, the Android system, will sleep as often as possible, every time it's possible.
- So, there is a need of a mechanism to force the system to stay awake when critical operations take place or it's waiting for the user's input.
- This mechanism is implemented using wakelocks, and their purpose is to keep the system awake.

8.

- An application that wants to run and keep the system awake during that time, it must obtain a wakelock.
- In general, developers don't need to request wakelocks directly, because the abstractions used by applications handle locking automatically.
- However, applications can request wakelocks explicitly from the service called PowerManager.
- However, device drivers will call the wakelock primitives directly from the kernel in order to obtain and release a wakelock.
- In Linux version 3.5, in 2012, an equivalent to wakelocks and the correlated early suspend mechanism have been merged into the mainline. The replacement of early suspend is called autosleep and wakelocks are replaced by the new epoll() flag EPOLLWAKEUP.

9.

- In Android, it is very important to handle low-memory situations. This is why Android includes the Low-memory killer, which will run before the default Out of Memory Killer (OOM) from the Linux kernel.
- This OOM killer will be activated only when there is no memory left and the purpose of Low-memory killer is to prevent the activation of the OOM killer.
- This is done through killing processes with components that have not been used in a long time and don't have a high priority.
- Low-memory killer is based on the OOM adjustments mechanism through which we can have different priorities for different processes.
- These adjustments will allow userspace to control a part of the process killing policies.
- Userspace policies are applied by the init process at startup and can be re-adjusted and enforced at runtime by the ActivityManager (which is part of the System Server).

10.

- OOM adjustments are from -17 to 15, and a larger value means that the process will probably be killed by the system if the memory is low.
- Android will assign an adjustment level for each type of process, depending on the components that are running (within that app) and configures the killer to apply thresholds (MinFree) for each type of process. Here we have some types of processes.
- A process will be killed when memory reaches a certain threshold.
- This is how the Low-memory Killer activates before the OOM killer, because you never have the situation in which the memory is fully occupied.
- Low-memory killer has been included in the mainline from Linux 3.10.

11.

- Anonymous Shared Memory (ashmem) is an IPC mechanism. It is a file-based, reference-counted shared memory interface.
- The Android team from Google is against using System V IPCs because those may lead to resource leaks in the kernel, and allow malicious or misbehaving applications to tamper with the system.
- Ashmem is similar to Posix shared memory, but there are some differences.
- Ashmem uses a reference count in order to destroy a memory region when all processes which referred that region have terminated.
- In addition, the mechanism will shrink the mapped memory regions when the system needs more memory.
- If a region is pinned, it cannot be shrinked, therefore it has to be unpinned first.

12.

- Memory sharing is done using the following steps: the first process creates a memory region using ashmem, then it uses the Binder in order to share the associated file descriptor with other processes.
- Many system server components are based on ashmem, through IMemory interface, not directly (for example: Surface Flinger, Audio Flinger).
- The ashmem driver has been included in the staging tree from version 3.3, but it’s not yet included in the mainline.

13.

- The alarm driver is based on Real-Time Clock (RTC) and High Resolution Timers (HRT) mechanisms.
- setitimer() is a system call that determines the generation of a signal when the time expires. This system call is based, among others, on the ITIMER_REAL which uses the HRT from the kernel.
- However, it does not work when the system is suspended. So the application will receive the signal only when the system wakes up.
- Apart from that, there is the RTC driver that can be accessed through /dev/rtc and you can communicate with it through ioctl() calls. If we use RTC, the alarm will be generated

even if the system is suspended because the RTC device remains active even when the rest of the system is suspended.

- The alarm driver for Android combines both mechanisms. By default it will use the HRT driver to generate alarms, but when the system is about to suspend, it programms RTC to wake up the system at the right moment.
- This way, any userspace application can use this driver to generate an alarm, even if the system is awake or suspended.

14.

- The driver can be accessed from userspace through /dev/alarm, which is a character device.
- It will allow alarm setup and time configuration through ioctl() calls.
- Many key components of Android framework rely on this driver. For example, SystemClock uses on the alarm driver for obtaining and setting time. Also, the AlarmManager uses it in order to provide alarm services for applications.
- Both the driver and AlarmManager will use Wakelocks to keep consistency between alarms and the rest of the system. For example, when an alarm is generated, the application will be able to perform the necessary tasks before the system will go into suspend again (because it holds a wakelock).
- This driver has been included in the mainline from version 3.20.

15.

- Logging events is necessary on every system. Android has its own logging mechanism.
- First we are going to talk about the logger kernel driver that was default until Android version 5.0.
- The driver uses circular kernel buffers stored in the RAM.
- Each kernel buffer has a separate entry in /dev/log - they are called Events, System, Radio and Main. For example, when we are using logcat, it displays the contents of the Main buffer.
- On the Java side, we have 3 classes used for logging: Log, Slog and EventLog. The Log and EventLog classes are part of the public API but Slog is used only in system components.

16.

- The logger driver cannot be accessed directly, but through the liblog library. Every call passes through this library (from the java classes and from logcat). The library deals with formatting and filtering events.
- Each log has 3 components: a priority, a tag and data. The priority can be: verbose, debug, info, warn, error and assert. The tag is usually the name of the component that generated the event.

17.

- Here we have a diagram of the logging system on Android. At the bottom we have the logging driver with the 4 circular buffers, On top of that, we have the liblog library, and on the Java side, we have the three classes: Log, EventLog and Slog. The classes from the android.* package and the system_server are calling the java classes, which in turn call the liblog library, and then the logging driver. Logcat and Bionic also make calls to the liblog library which in turn calls the logging driver.

18.
- From Android 5.0, we have a new logging mechanism that has a daemon called logd. This daemon acts as a centralized user-mode logger.
- Logd addresses the disadvantages of using the circular buffers (their small size and requirement for resident memory).
- Logd can be integrated with SELinux, by registering itself as the auditd in order to receive SELinux messages from the kernel via netlink.

19.
- It uses 4 sockets: /dev/socket/logd, which is the control interface socket
- /dev/socket/logdw, which is a write-only socket used for writing log messages
- /dev/socket/logdr, which is a read-only socket, for reading log messages
- An unnamed netlink socket that is used when integrating logd with SELinux.

20.
- The sockets are not accessed directly, but through the liblog library. So the applications use the Log class (or EventLog) to send log messages, which in turn calls the liblog library through JNI, that opens the /dev/socket/logdw socket and writes the log message.
- For reading logs, logcat will connect to the /dev/socket/logdr through the liblog API and instruct the LogReader instance of logd to provide the logs by sending parameters to it.

21.
- In general, on Linux, all processes are allowed to create sockets and access the network.
- On Android, for security reasons, we must control which applications have access to the network.
- The paranoid networking mechanism restricts access to the network depending on the group of the calling process.
- In order to be able to create AF_INET and AF_INET6 sockets, a process must have the supplementary GID AID_INET.
- To create raw INET sockets, a process must have the supplementary GID AID_NET_RAW.
- Membership in the AID_NET_ADMIN group grants CAP_NET_ADMIN capability that allows the configuration of network interfaces and routing tables.
- Membership in the AID_NET_BT allows the creation of Bluetooth sockets (SCO, RFCOMM, L2CAP), while AID_NET_BT_ADMIN is able to manage BT connections.

23.

- The Binder implements a Remote Procedure Call (RPC) mechanism.
- At first, it was developed for BeOS, that was then bought by Palm.
- Then it was made open for developers through the OpenBinder project. There are some developers that worked on the original OpenBinder, that are now in the Android team from Google.
- The Android Binder is inspired from OpenBinder (it provides the same functionality), but it is not derived from it, it was written from scratch.
- So the OpenBinder documentation can be used for understanding the general mechanism, because there is little documentation for the Android Binder, which is true for most internal components of Android.
- The Binder driver has been included in the mainline from version 3.19.

24.

- The Binder extends the functionality of the system through the invocation of remote objects.
- Instead of creating a new daemon for each service, you can use a remote object implemented in a certain programming language (C, Java) that runs in the same process with other services or in a different process.
- All we need is an interface and a reference to it, and we can invoke remote methods.
- The Binder is a central component of the Android architecture. It is used for communicating with both system services and application services.
- Application developers will never use the Binder directly, but will use interfaces and stubs generated through the aidl tool.
- The public API will also call stubs when communicating through the Binder with the system services.

25.

- A part of the Binder is implemented as a kernel driver. This driver can be accessed through the character device /dev/binder.
- The components that communicate through the Binder will be able to exchange parceled data through ioctl() calls.

27.

- The Android framework runs on top of the native userspace, and includes several components: android.* packages, the system services and the Android runtime.
- The framework source code is included in the frameworks/ directory from the AOSP. The framework includes some key components: Service Manager, Zygote and Dalvik/ART.

28.

- System services in Android form what is sometimes called an object oriented operating system running on top of the Linux kernel.
- The main component is System Server, which runs in the process with the same name: system_server and includes a large number of services written in Java and 2 services written in C/C++.
- Here we have: PowerManager, ActivityManager, PackageManager, LocationManager and so on (implemented in Java). The Surface Flinger and Sensor Service are the ones implemented in C/C++.
- Another set of services is included in MediaService, which runs in the process with the same name: mediaservice. These services are written in C/C++ and deal with audio, video and camera, for example: Audio Flinger, Media Player Service and Camera Service.

29.

- Dalvik is the Runtime that was used as default before Android 5.0.
- It is a Java virtual machine optimized especially for mobile devices, so it will have a low memory footprint.
- It works with dex files, which are with 50% smaller than the jar files that contain the same classes.
- In contrast to the original Java virtual machine, which was stack-based, Dalvik is register-based. So it will work with a finite number of registers that store positive integers.
- Dalvik cannot run standard Java bytecode. But the Java bytecode can be translated into Dalvik bytecode using the dx command that builds the dex file.
- The Java bytecode includes 8 bits instructions specific for the stack. This means that all local variables must be copied on the stack before being used. In contrast, Dalvik works with 16 bits instructions and accesses all local variables directly. This leads to less instructions and higher execution speed.

30.

- From Android 2.2, Dalvik includes the Just-in-Time compiler for ARM. Then it was also available for x86 and MIPS.
- Practically, JIT optimizes application execution, by profiling applications every time they run.
- JIT translates sections of bytecode (that are executed most frequently - called traces) into machine instructions at runtime, that will run natively on the CPU of the device, instead of being interpreted instruction with instruction by the virtual machine.
- The rest of the bytecode is interpreted by the Dalvik VM, but the native execution of the traces brings performance improvements.

31.

- Android Runtime (ART) is available from Android 4.4, and comes to replace Dalvik, but it still works with dex files for backward compatibility.

- From Android 5.0 ART is the default runtime.
- First of all, ART includes Ahead-of-Time compilation (AOT), which brings considerable performance enhancements. ART will use the dex2oat tool at installation time, in order to translate the dex file into an executable for the target device (oat format embedded into an elf file). So the whole application will run natively on the CPU of the device, so AOT replaces JIT compilation and Dalvik interpretation.
- The disadvantages are that the installation takes a longer time because it includes the AOT compilation and the executables occupy more storage space.
- In addition, it provides a more strict verification of the applications at installation time.

32.

- ART also provides a more efficient garbage collection and support for a dedicated sampling profiler that generates accurate information about the application execution without affecting performance.
- It provides an additional number of debug options, mostly regarding monitoring and garbage collection.
- In addition, ART will generate more details and informations regarding the context when an exception is produced at runtime.

33.

- Zygote is a daemon used for starting applications, and is active only when a new application has to be started.
- It is practically the parent of all processes on Android.
- In the beginning, Zygote will preload in RAM all Java classes and resources that could be used by applications. Zygote listens for connections on a socket /dev/socket/zygote, so this is how it receives requests for starting applications.
- When it receives such a request, it forks itself and starts the application in the new process.
- The advantage of having all applications forked from Zygote, is that we have all classes and resources preloaded in the memory and the applications can start their execution directly.

34.

- This happens because of the Copy on Write mechanism (CoW) in the Linux kernel. When a new process is created using fork, it is a copy of the parent and will have all memory pages mapped so there is no need to copy them. Only when the parent or the child writes in a page, a copy will be created.
- The classes and resources used by applications are never modified at runtime, so all applications will have access to the ones preloaded by Zygote, so there is no need to copy them. A single version of the classes and resources is stored in RAM.
- There is a single process that is explicitly started by Zygote (without any request) - the system server, this is the first process started by Zygote.

- If we execute the ps command in the shell we will see that the Zygote’s PID is the PPID of all applications and of the system server.

36.

- Next, we will discuss about them most important managers: ServiceManager, ActivityManager, PackageManager and PowerManager.
- An important component in the framework is ServiceManager. It is responsible with identifying system services (making the operation called lookup). It is something like the Yellow Pages of every service in the system.
- A system service that has not been registered to the ServiceManager cannot be accessed, so it does not exist. Any service that wants to be accessible, needs to register first to the ServiceManager.
- This manager is started by the init process before any other service. When it starts, it accesses /dev/binder and uses an ioctl() call in order to make itself the ContextManager of the Binder.
- Why? In order to become the magic object (Binder ID 0). Any process that communicates with Binder ID 0 (the magic object or magic Binder), will communicate in fact with the ServiceManager through the Binder.

37.

- Each system service will register itself to the ServiceManager (through a Binder call). The Manager will maintain a list of available services.
- When an application wants to communicate with a system service, it will request from the ServiceManager a handle to that service (by calling getSystemService()) , and then it can call the methods of the service through that handle (also through the Binder - all operations are done through the Binder).
- This mechanism is available only for system services, and will not be used to access application services (in that case the call will go directly through the Binder without being looked up in the ServiceManager).
- In addition, the manager is used by several tools like dumpsys, that dumps the status of a service or of all services in the system. First of all it will request a list with all services, then it will request a handle for each service, and then it will call the dump method from each handle.

38.

- ActivityManager is one of the most important services in the SystemServer, and is responsible with handling the lifecycle of activities.
- This service will start activities and services in applications, will obtain content providers, and will send intents.
- The Application Not Responding (ANR) dialog is also generated by the ActivityManager.
- In addition, it is involved in many adjacent tasks - it verifies permissions, it helps compute the OOM adjustments for the Low-memory killer, it performs task management.

39.

- ActivityManager is the one that starts the launcher with an intent with the type CATEGORY_HOME.
- Let's see what happens when an application is started from the launcher: the onClick() callback in the launcher is called. In this callback, the method startActivity() from the ActivityManager will be called through the Binder. The service will call the method startViaZygote() which will open a connection on the socket with Zygote and will send a request to start a new process for that activity.
- In the command line, we have the am tool for sending commands to the ActivityManager. We can start an activity, a service, send an intent, start profiling or make debugging.
- A funny method name from the ActivityManager: isUserAMonkey :))

40.

- PackageManager is the service that handles the apk files. It provides the possibility to install, uninstall and upgrade packages.
- PackageManager works with several files found in the /data/system/, the most important are:
    - packages.xml which contains all permissions and information about the installed packages
    - packages.list contains all installed packages, their uid and data directory
- It runs in the system_server but uses the installd daemon in order to perform most of the operations (because installd has sufficient permissions).
- PackageManager also deals with solving intents, more exactly with identifying the component that should receive the intent. It will receive a resolving request and will use the information from the Manifest files in order to identify the most appropriate component.
- In the command line, we have the pm tool, that can be used for sending commands to the PackageManager. We are able to list all installed packages, view the permissions required by an application, install and uninstall packages, display the folder of an application, disable a package, and so on.

41.

- PowerManager is responsible with the power consumption control of the device. This is the place in the AOSP where Wakelocks are managed.
- The service includes the WakeLock class with the associated methods: acquire() and release().
- The applications will request Wakelocks from the PowerManager.
- The actual handling of power consumption is implemented in the kernel, but all calls must pass through the PowerManager.
- The service can also force the device to enter in sleep mode and configure the luminosity of the screen.