

**Chapter One****1.1 MOBILE SYSTEM**

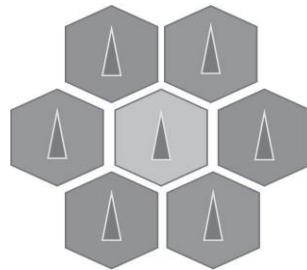
**Mobile system** includes **mobile device, mobile operating system, wireless network, mobile app, and app platform.**

The mobile device consists of not only smartphones but also other handheld computers, such as a tablet and Personal Digital Assistant (PDA). A mobile device has a mobile operating system and can run various types of apps. **The most important parts of a mobile device** are Central Processing Unit (CPU), **memory, and storage**, which are similar to a desktop but perform weaker than an on premise device. Most mobile devices can also be **equipped with Wi-Fi, Bluetooth, and Global Positioning System (GPS) capabilities**, and they can connect to the Internet, other Bluetooth-capable device and the satellite navigation system. Meanwhile, a mobile device can be equipped with some human - computer interaction capabilities, such as camera, microphone, audio systems, and some sensors.

All kinds of mobile devices run on various mobile Operating Systems (OS), also referred to mobile OSs, such as iOS from Apple Inc., Android from Google Inc., Windows Phone from Microsoft, Blackberry from BlackBerry, Firefox OS from Mozilla, and Sailfish OS from Jolla. Mobile devices actually run two mobile operating systems. Besides the mobile operating systems that end users can see, mobile devices also run a small operating system that manages everything related to the radio. Because of the high time dependence, the system is a low-level

proprietary real-time operating system. However, this low-level system is security vulnerable if some malicious base station gains high levels of control over the mobile.

Mobile devices can connect to the Internet by wireless networks. There are **two popular wireless networks for mobile devices: cellular network and Wi-Fi.** The **cellular network** is peculiar to portable transceivers. A cellular network is served by at least one fixed-location transceiver, called cell site or base station, as shown in [Fig. 1.1](#). Each mobile device uses a different set of frequencies from neighboring ones, which means a mobile device must connect to the base station before it accesses to the Internet. Similarly, when a mobile device using a cellular network wants to connect another mobile device, it must connect to some base stations before it communicates with the target device via the base stations.



[Figure 1.1](#) Structure of a cellular network.



[Figure 1.2](#) Logo of Wi-Fi.

Wi-Fi is a local area wireless technology, which allows mobile devices to participate in computer networks using 2.4 GHz and 5 GHz radio bands. Fig. 1.2 represents two common logos of Wi-Fi. Mobile devices can connect to the Internet via a wireless networking access point. The valid range of an access point is limited, and the signal intensity descends as the distance increases. Wi-Fi allows cheaper deployment of Local Area Networks (LAN), especially for spaces where cables cannot be run. Wi-Fi Protected Access encryption (WPA2) is considered a secure approach by providing a strong passphrase. A Wi-Fi signal occupies five channels in the 2.4 GHz band. Any two channel numbers differ by five or more. Many newer consumer devices support the latest 802.11ac standard, which uses the 5 GHz and is capable of multistation WLAN throughput of at least 1 gigabit per second.

*1. Hz is the unit of frequency in the International System of Units and is defined as one cycle per second. One gigahertz (GHz) represents 10<sup>9</sup> Hz.*

*2. IEEE 802.11ac was approved in January 2014 by IEEE Standards Association.*

A **mobile app** is a program designed to run on smartphones, tablet



computers, and other mobile devices. Mobile apps emerged in 2008 and are operated by the owner of the mobile operating systems. Currently, the most popular digital distribution platforms for mobile apps are App Store, Google Play, Windows Phone Store, and BlackBerry App World, as shown in [Fig. 1.3](#). These platforms are developed by Apple Inc., Google, Microsoft, and BlackBerry Ltd., respectively, and provide different apps, which only can be used on their own operating systems.

[Figure 1.3](#) Four dominate platforms for mobile apps.

## 1.2 MOBILE INTERFACE AND APPLICATIONS

Mobile devices, to some extent, are much more powerful than desktops. They are highly personal, always on, always with users, usually connected, and directly addressable. Furthermore, they are crawling with powerful sensors with various functions that detect location, acceleration, orientation, movement, proximity, and surrounding conditions. The portability of mobile devices combined with powerful sensors makes mobile interface extremely valuable for using mobile devices.

The User Interface (UI) is the look and feel of the on-screen system, including how it works, its color scheme, and how it responds to users' operation. The interactions include not only users' active operations, but also the passive ones. Users' passive operations include users'

locations, movements, and other information that does not need users' active operations. We will take telehealth as an example of mobile interface. Telehealth is the delivery of health-related services and information via telecommunications technologies [7].

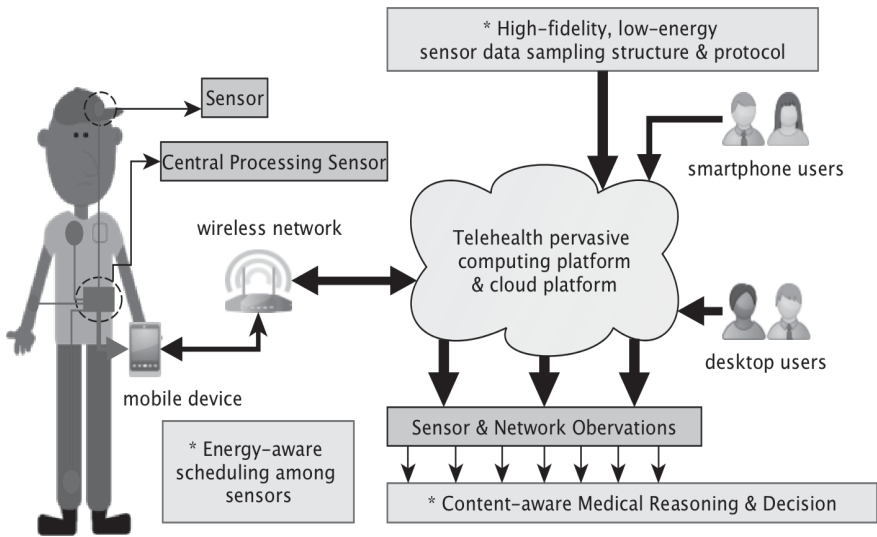


Figure 1.4 Structure of the telehealth systems.

We can separate telehealth system into several modes: store-and-forward, real-time, remote patient monitoring, and electronic consultation, as shown in Fig. 1.4. Each mode finish their job respectively and achieve the whole process of collecting data from users, transmitting this data to medical or clinical organizations, medical reasoning and decision, and sending back to users. In the first step, observations of daily living and clinical data are captured and stored on the mobile device. All the sensors that collect and record data are heterogeneous medical devices with different cost and time features. Then the mobile

device transmits this information to the Telehealth pervasive computing platform and cloud platform by wireless network.

Consequently, **main challenges** include finding out the approach of collecting data from users by using sensors and scheduling sensors for achieving energy-aware purposes. The process of transmitting data is a part of real-time system. Different to normal real-time systems, the data transmitting in telehealth is under a wireless condition. Similar to the first step, there are various network paths with different cost and time requirements, which results in a great challenge to security and data integrity.

Furthermore, context-aware medical reasoning and decision is another important issue in telehealth system. Context can refer to real world characteristics, such as temperature, time or location. Combining with users' personal information, the medical reasoning and decision focus on data analytic, mining, and profiling issues. In conclusion, all the challenges mentioned above can be summarized as a general problem: how to minimize the total cost of heterogeneous telehealth while finishing the whole diagnosis within certain time constraints .

### **1.2.1 Optimizations in Mobile Systems**

All current mobile devices are battery-powered devices. The high usage of mobile devices makes them hard to keep on charging like desktops, so the improvement of battery life on mobile devices is gaining increasing attention. Besides some energy-saving operations by users, there are some researches focusing on the optimization in mobile system. The

optimization problem, to some extent, is a tradeoff among multiple constraints. Before talking about the optimization, let us discuss some constraints in mobile systems.

The first and the most important constraint is the energy. The second one is the performance. The third one is the networking speed to the Internet. The fourth one is the resources of the mobile device. These constraints are interrelated and mutually restrict to each other. Suppose in an extreme situation, someone keeps his/her mobile device off. In this situation, the battery life can last an almost unlimited time without considering the self-discharge of the battery. However, the mobile device in that situation is useless, and no one buys a mobile device just for decoration. It is obvious that the more functions users use, the more energy devices consume. Similarly, the performance is related to the networking speed while constrained by the energy and resource. To solve this problem, many researchers proposed various optimization algorithms and frameworks.

### **1.2.2 Mobile Embedded System**

An embedded system is a computer system with a dedicated function, which is embedded as a part of a complete devices including hardware and mechanical parts. Embedded systems are driving an information revolution with their pervasion. These tiny systems can be found everywhere, ranging from commercial electronics, such as cell phones, cameras, portable health monitoring systems, automobile controllers, robots, and smart security devices, to critical infrastructure, such as telecommunication networks, electrical power grids, financial institu-

tions, and nuclear plants. The increasingly complicated embedded systems require extensive design automation and optimization tools. Architectural-level synthesis with code generation is an essential stage toward generating an embedded system satisfying stringent requirements, such as time, area, reliability, and power consumption, while keeping the product cost low and development cycle short.

A mobile device is a typical embedded system, which includes mobile processors, storage, memory, graphics, sensors, camera, battery, and other chips for various functions. The mobile device is a high-level synthesis for real-time embedded systems using heterogeneous functional units (FUs). A functional unit is a part of an embedded system, and it performs the operations and calculations for tasks. As a result, it is critical to select the best FU type for various tasks.

### **1.3 MOBILE CLOUD**

Limited resources is another critical characteristic of mobile devices. With the development of cloud computing, mobile cloud computing has been introduced to the public. Mobile cloud computing, as shown in [Fig. 1.5](#), is the combination of cloud computing, mobile computing, and wireless networks to bring rich computational resources to the mobile system. In general, a mobile device with limited resources can utilize computational resources of various cloud resources to enhance the computational ability of itself. There are several challenges in mobile cloud computing, such as moving computational processes from mobile devices to the cloud, networking latency, context processing, energy management, security, and privacy.



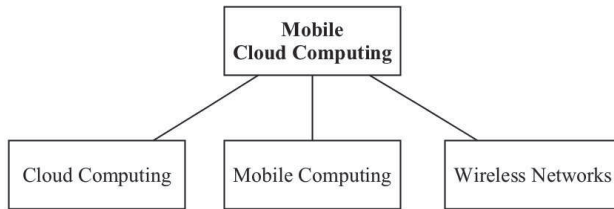


Figure 1.5 Main structure of mobile cloud computing.

Currently, some research and development addresses execution code offloading, seamless connectivity and networking latency; however, efforts still lack in other domains.

**Architecture.** The architecture for a heterogeneous mobile cloud computing environment is crucial for unleashing the power of mobile computing toward unrestricted ubiquitous computing.

**Energy-aware transmission.** Offloading executive codes into the cloud can greatly reduce the burden and the time of local mobile devices, but increase the transmission between mobile devices and the cloud. The transmission protocol should be carefully designed for saving energy.

**Context-aware computing.** Context-aware and socially aware computing are inseparable traits of mobile devices. How to achieve the vision of mobile computing among heterogeneous converged networks among mobile devices is an essential need.

**Live Virtual Machine (VM) migration.** A virtual machine is an emulation of a particular computer system. Executive resource

offloading involves encapsulation of a mobile app in a VM instance, and migrating in the cloud is a challenging task.

**Security and privacy.** Due to lack of confidence in the cloud, many users are concerned with the security and privacy of their information. It is extremely important to improve the security and the privacy of mobile cloud computing.

### **1.3.1 BigData Application in Mobile Systems**

Big data is an all-encompassing term for any collection of data sets so large or complex that it becomes difficult to process them using traditional data processing applications. Data sets grow in size in part because they are increasingly being gathered by mobile devices. There are 4.6 billion mobile phone subscriptions worldwide and between 1 billion and 2 billion people accessing the Internet.

With billions of mobile devices in the world today, mobile computing is becoming the universal computational platform of the world. These mobile devices generate huge amounts of data every day. The rise of big data demands that we be able to access data resources any- time and anywhere about every daily thing. Furthermore, these kinds of data are invaluable and profitable if used well.

However, a few challenges must be addressed to make big data analytics possible. More specifically, instead of being restricted to single computers, ubiquitous applications must be able to execute on an ecosystem of networked devices, each of which may join or leave the shared ubiquitous space at any time. Moreover, there exist analytics tasks that are too computationally expensive to be performed on a

mobile device ecosystem. Also, how can we harness the specific capabilities of each device, including varying display size, input modality, and computational resources?

### **1.3.2 Data Security and Privacy Protection in Mobile Systems**

Due to the universality and the particularity of mobile systems to desktop system, the security in mobile systems is much more complicated and important than that in desktop systems. The security in mobile systems can be separated into a few parts.

The first threat is the malware (virus). Mobile malware is a malicious software that targets mobile devices and results in the collapse of the system and loss or leakage of information. According to the June 2014 McAfee Labs Threat Report, new mobile malware has increased for five straight quarters, with a total mobile malware growth of 167 percent in the recent past years. Security threats are also growing with 200 new threats every minute. In addition to 2.4 million new samples of mobile malware, 2013 also brought 1 million new unique samples of ransomware, 5.7 million new malicious signed binaries, and 2.2 million new Master Boot Record (MBR)-attack-related samples. The most frequent two incentives are exfiltrating user information and premium calls or SMS. Furthermore, there are some other incentives, such as sending advertisement spam, novelty and amusement, and exfiltrating user credentials.

Another research issue is the security frameworks or approaches for detecting mobile malware. There are several approaches for monitoring mobile devices and detecting mobile malware. The signature-based

solution is an approach used for detecting attacks, but it fails miserably in detecting the sophisticated cyber-criminal who targets specific organizations with exploits tailored to those victims. From a process perspective, when it comes to validating a threat and subsequent root cause analysis, first-level responders have to send all data that looks like malicious code to the reverse engineers. This process often causes delays, because these malware teams are typically inundated.

Meanwhile, with the development of technology, an efficient representation of malware behaviors using a key observation often reveals the malicious intent even when each action alone may appear harmless. The logical ordering of an application's actions are often over time. Based on this idea, researchers present various approaches to monitor and detect malicious behavior using static analysis on data flow.

Next security problem is the data over-collection behaviors in mobile apps. Current mobile phone operating systems only provide coarsegrained permissions that determine whether an app can access private information, while providing few insights into the scope of private information being used. Meanwhile, only a few users are aware of permissions information during the installations. Furthermore, some users choose to stop installing or to uninstall an app when the system warns them and asks for permission, even though they know it may bring some hidden security troubles. For example, we take location data and analyze the current status and discuss the risks caused by over collecting it.

Location data are the most frequently used data in smartphones. It can be used in apps whose main functions include maps, photo or-

ganization, shopping and restaurant recommendations, and weather. From the report of Appthority, 50% of the top iOS free apps and 24% of the top iOS paid apps track a user's location. Although users are warned whenever an app intends to capture their locations, they usually choose to allow the permission for the function offered by the app. Apps that over collect location data can be separated into two main types: location service as main function and location service as the auxiliary function. The first type of apps normally ask users for permissions to their location information, while the other app type can collect users' location information without noticing users. The first and the most direct risk is a physical security concern. Users' tracks are easily exposed to those who have users' real-time and accurate location data. Users' habits and customs are easy to be inferred by using simple data mining methods.

Furthermore, solving the data over collection problem is also a research issue in mobile apps. PiOS, presented by M. Egele et al., to detect privacy leaks in iOS applications, used static analysis to detect sensitive data flow to achieve the aim of detecting privacy leaks in applications in iOS. Sharing a similar goal with PiOS, TaintDroid, is a system wide dynamic taint tracking multiple sources of sensitive data. The main strategy of TaintDroid is real-time analysis by leveraging Android's virtualized execution environment. Another secure model via automated validation uses commodity cloud infrastructure to emulate smartphones to dynamically track information flows and actions. This model automatically detects malicious behaviors and sensitive data misuse via further analysis of dependency graphs based on the tracked

information flows and actions.

These approaches or techniques mentioned above only focus on monitoring and detecting apps. The prerequisites are that apps already gain permissions from users. However, these solutions only provide methods of monitoring and detecting behaviors of data over-collections. This approach leaves remedying operations to users, such as disabling the permissions of apps or uninstalling those apps. Users have to manually disable permissions of these apps that over collect users' data or uninstall them. Furthermore, running these approaches or tools adds the consumption of energy, which is particularly valuable for smart-phones with limited resources. As a result, the active method of avoiding data over collection behaviors in mobile apps is a crucial challenge that needs to be solved.

### **1.3.3 Concept of Mobile Apps**

Mobile apps were originally developed to offer general productivity and information retrieval, including email, calendar, contacts, and weather information. However, with the rapid increment of public requirement, mobile apps expand into lots of other categories, such as games, music, finance, and news.

A lot of people distinguish apps from applications in a perspective of device forms. They think that applications are used on a desktop or laptop, while apps are used on a phone or tablet. Nevertheless, this simplistic view is too narrow and no longer the consensus, because apps can be used on desktops, and, conversely, applications can run on phones. At Gartner Portals, Content and Collaboration Summit 2013, many experts and developers participated a roundtable discussion titled

“Why an App is not an Application”. They proposed that the difference between app and application is not about the delivery mechanism and landed on a consensus that:

*App = software designed for a single purpose and performs a single function.*

*Application = software designed to perform a variety of functions.*

From the view of users, they do not care whether it is an app or an application by definition, and they just want to accomplish their tasks easily. Meanwhile, from the view of developers, the question they should answer is not whether they should be building an app or an application, but how they can combine the best of both into something users love.

### **1.3.4 Brief Introduction of Android and Its Framework**

#### **1.3.4.1 A Brief History of Android**

Android was founded in Palo Alto, California, in October 2003 by Andy Rubin, Rich Miner, Nick Sears, and Chris White in an effort to develop a smarter mobile device that is more aware of its owner’s location and preferences. Then to Google acquired Android Inc. and key employees, including Rubin, Miner, and White, on August 17, 2005. At Google, the team, led by Rubin, developed a mobile device platform powered by the Linux kernel. Google had lined up a series of hardware components and software partners and signaled to carriers that it was open to various degrees of cooperation on their part. On November 5, 2007, the Open Handset Alliance unveiled itself with a goal to develop open standards for mobile devices. This alliance includes technology companies, like Google, device manufacturers such as HTC, wireless carriers such as T-Mobile, and chipset makers such as Qualcomm. Then, on October 22, 2008, the first commercially available smartphone

running Android came out with a fantasy name: HTC Dream. Since 2008, Android has seen numerous updates that have incrementally improved the operating system, adding new features and fixing bugs in previous releases. There are some milestones of Android SDK, such as Android SDK 2.0 (Eclair) in 2009, Android SDK 3.0 (Honeycomb) for tablets only in 2011, Android SDK 4.0 (Ice Cream Sandwich) in 2011, Android 4.1 to 4.3 (Jelly Bean) in 2012, Android SDK 4.4 (KitKat) in 2013, and Android SDK 5.0 (Lollipop) in 2014.

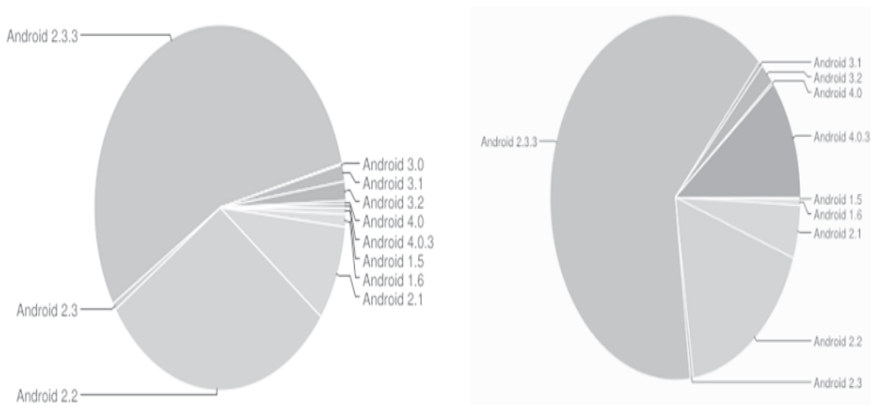


Figure 1.6 Android device distribution in January and July 2012.

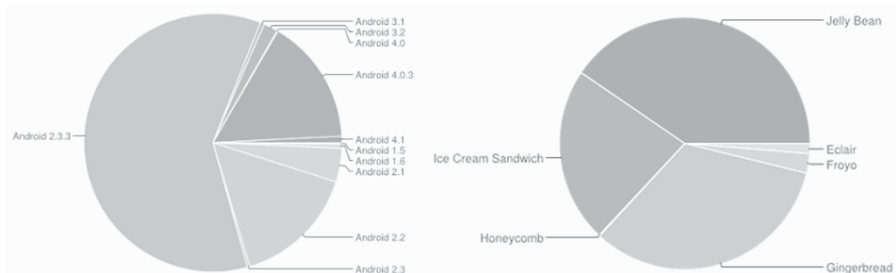


Figure 1.7 Android device distribution in August 2012 and August 2013.



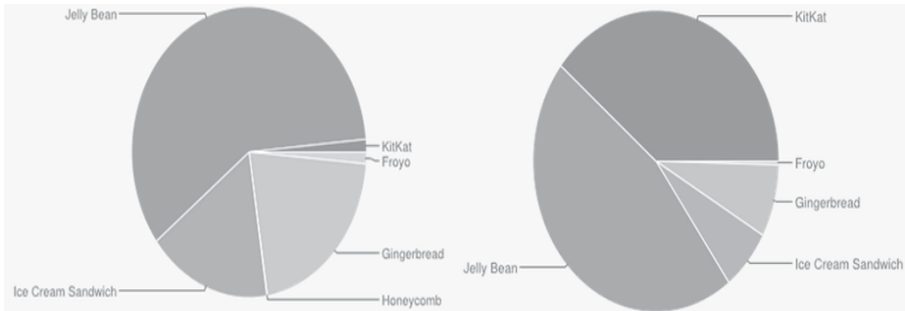


Figure 1.8 Android device distribution in January 2014 and January 2015

**1.3.4.2 Android Device Distribution**

Fig. 1.6 shows the Android device distributions in 2012. We can see that Android 2.3.3 and 2.2 dominate more than half of the market. Nonetheless, in the second half of 2012, Android 4.0.3 became more and more popular. In August 2013, Android 4.0 and 4.1, named Ice Cream Sandwich and Jelly Bean, respectively, surpassed Android 2.0s and dominated the Android market, as shown in Fig. 1.7. In January 2014, Android 4.1 to 4.3 still dominated the Android market. However, after one year, Android 4.4, named KitKat, rapidly occupied 39.1% of the whole market, as shown in Fig. 1.8.

**1.3.4.3 Android SDK**

Android SDK is open-source and widely used, which makes it the best choice for teaching and learning mobile development. Android is a software stack for mobile devices, and it includes a mobile operating system, middleware, and some key applications. As shown in Fig. 1.9, there are Linux kernel, libraries, application framework, and applications and widgets, from bottom to top. We will introduce them one by one.

The Linux kernel is used to provide some core system services, such as security, memory management, process management, power management, and hardware drivers. These services cannot be called by Android programs directly and is transparent to users. The next layer above the kernel is the native libraries, which are all written in C or C++. These libraries are compiled for the particular hardware architecture used by the mobile devices. They are responsible for handling structured data storage, graphics, audio, video, and network, which only can be called by higher-level programs. Meanwhile, Android run-time is also on top of the kernel, and it includes the Dalvik virtual machine and the core Java libraries.

*What is Dalvik? Dalvik is the process virtual machine in Google's Android operating system, which specifically executes applications written for Android. Programs are written in Java and compiled to bytecode for the Java virtual machine, which is then translated to Dalvik bytecode and stored in .dex and .odex files. The compact Dalvik executable format is designed for systems with limited resources.*

The application framework layer provides the high-level building blocks used for creating application. It comes preinstalled with Android, but can be extended with its own components as needed. We will introduce some basic and important building blocks of Android.

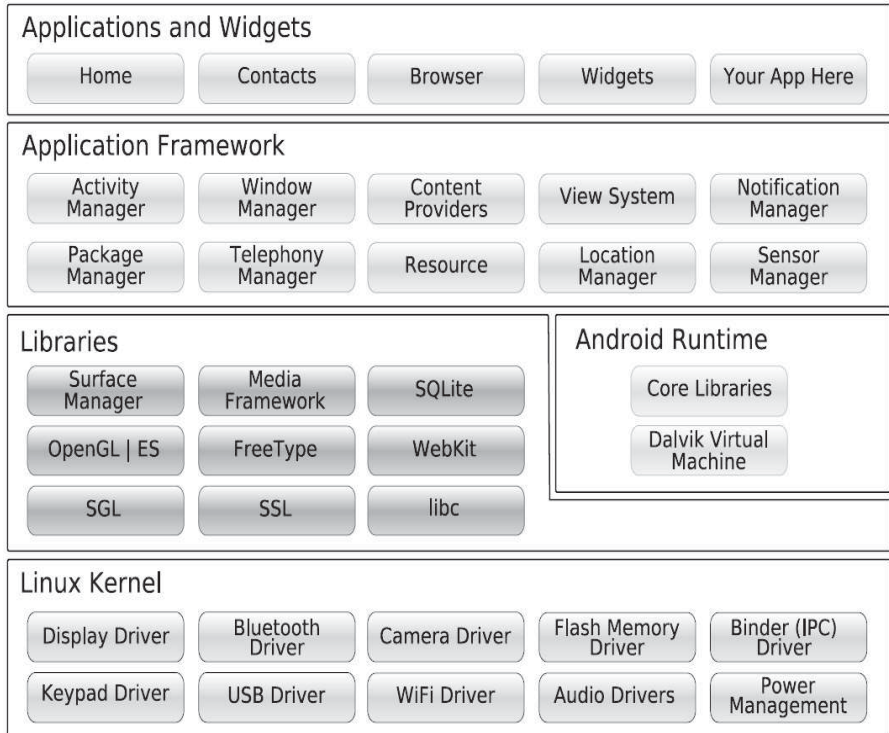


Figure 1.9 Android system architecture.

**Activity.** An activity is a user interface screen. A single activity defines a single screen with a user interface, and it defines simple life cycle methods like **onCreat**, **onResume**, and **onPause** for handling interruptions. Furthermore, applications can define one or more activities to handle different phases of the program.

**Intent.** An intent is a mechanism for describing a specific action, such as “pick a photo”, or “phone home”. In Android, everything goes through intents, and developer, have plenty of opportunities to replace or reuse components. Intents can be implicit or explicit. An explicit intent can be to invoke another screen when a button is pressed on the *Activity* in

context. An implicit intent is when you create an intent and hand it off to the system to handle it.

**Service.** A service is a task that runs in the background without the user's direct interaction. In fact, it does the majority of processing for an application. Developers can sub-class the *Service* class to write their own custom service.

**Content Provider.** A *Content* provider is a set of data wrapped up in a custom Application Programming Interface (API) to read and write it. This is the best way to share global data between applications. The content provider provides a uniform singular interface to the content and data and provides a consistent interface to retrieve/store data via RESTful model supporting create, read, update, and delete (CRUD) operations.

**An Android Emulator,** as shown in [Fig. 1.10](#), called Android Virtual Device (AVD), is essential to testing Android app but is not a substitute for a real device. AVDs have configurable resolutions, RAM, SD cards,



skins, and other hardware. If you have installed Android SDKs, the AVD Manager can allow you to create AVDs that target any Android API level.

[Figure 1.10](#) Android Emulator.

**An Android emulator has the following basic functions:**

- Host computer's keyboard works as keyboard of device.
- Host's mouse acts as finger.
- Connecting to the Internet using host's Internet connection.
- Buttons: Home, Menu, Back, Search, Volume up and down.
- Ctrl-F11 toggle landscape to portrait.
- Alt-Enter toggle full-screen mode.

**However, emulators have some limitations. They do not support for:**

- Placing or receiving actual phone calls. USB and Bluetooth connections.
- Camera or video capture as input. Device-attached headphones.
- Determining connected state.
- Determining battery charge level and AC charging state.
- Determining SD card insert or eject. SD card is a nonvolatile memory card used extensively in portable devices.
- Simulating the accelerometer.

Then we will introduce the process of producing an Android app. In [Fig. 1.11](#), an android app is written in Java and generates .java file.

Then *javac* compiler `.java` reads source files and transforms java code into byte code. Then Dalvik takes responsibility for handling these byte codes combining with other byte codes for other `.class` files, and generates `classes.dex`. At last, `classes.dex`, resources, and `AndroidManifest.xml` cooperate and generate an `.apk` file, which is a runnable Android app.

Every Android app must have an `AndroidManifest.xml` file in its root directory. The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. The `AndroidManifest.xml` file names the Java package for the application and describes the components of the application, including the activities, services, broadcast receivers, and content providers that the application is composed of. The file also names the classes that implement each of the components and publishes their capabilities. These declarations let the Android system know what the components are and under what conditions they can be launched.

Furthermore, `AndroidManifest.xml` file determines, which processes will host application components, and it declares which permissions the application must have in order to access protected parts of the API and interact with other application. The file also declares the permission that others are required to have in order to interact with the application's components and lists the instrumentation classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is published. It declares the minimum level of the Android

API that the application requires, and it lists the libraries that the application must be linked to.

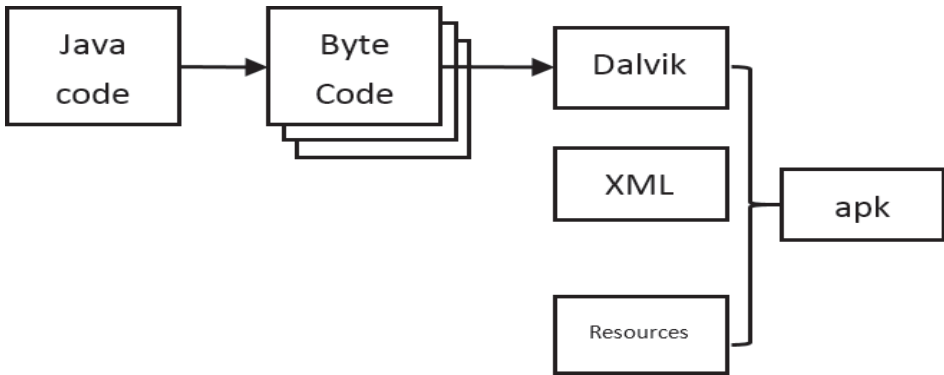


Figure 1.11 Process of producing an Android app.

## **Chapter Two**

### **Quick Start on Android**

#### **CONTENTS**

---

- 2.1 Installing Java
- 2.2 Installing Integrate Development Environment
- 2.3 Installing Android SDK
- 2.4 Creating an Android Application
- 2.5 Android Virtual Device

Before we jump into the Android world, let us have a quick review about Android installations, project creations, and application

executions. Introduce the process of installing Android and creating an Android project in this chapter. Main contents include:

Installing Java

Installing integrate development environment

Installing Android SDK

Creating an Android application project

Creating an Android Virtual Device

Running an Android application on the emulator

Running an Android application on a real phone

## 2.1 INSTALLING JAVA

The Android Software Development Kit (SDK) can work on any operating system, such as Windows, Linux, and Mac OS X. Before starting our installing Android and coding programs, we need to install Java. All the Android development tools require Java, and programs will be using the Java language. From the latest version of the Android Developer website, we suggest that Java 7 or 8 is the best choice.

We recommend getting the Java runtime environment (JRE) 8 from <http://java.com/en/download/manual.jsp>. For Windows users, there are two kinds of versions offered, which are 32-bit and 64-bit. You can choose the 32-bit download to use with a 32-bit browser, and choose the 64-bit download to use with a 64-bit browser. For Mac OS users, there is only one choice, which needs Mac OS X 10.7.3 version and above. For Linux users, there are four choices, and users can download one of them based on users' operating system.



It is not enough to just have a JRE, and you need the full development kit. We recommend downloading *Java Development Kit (JDK) 8* from <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>. To verify you have the right version, go to your shell window or terminal and type in “java ?version”. The result should be something similar to what is shown in [Fig. 2.1](#).

```
C:\>java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

[Figure 2.1](#) Verify the version of Java.

## 2.2 INSTALLING INTEGRATE DEVELOPMENT ENVIRONMENT

A Java development environment is recommended to make Android programming easier. There are many optional *Integrate Development Environments* (IDE), but we only introduce the most widely used one, which is Google’s Android Studio.

Android Studio is the official IDE for Android application development. You can download it from <http://developer.android.com/sdk/index.html>. After downloading and installing Android Studio, you can see a similar screen figure, as shown in 2.2, when you open it.

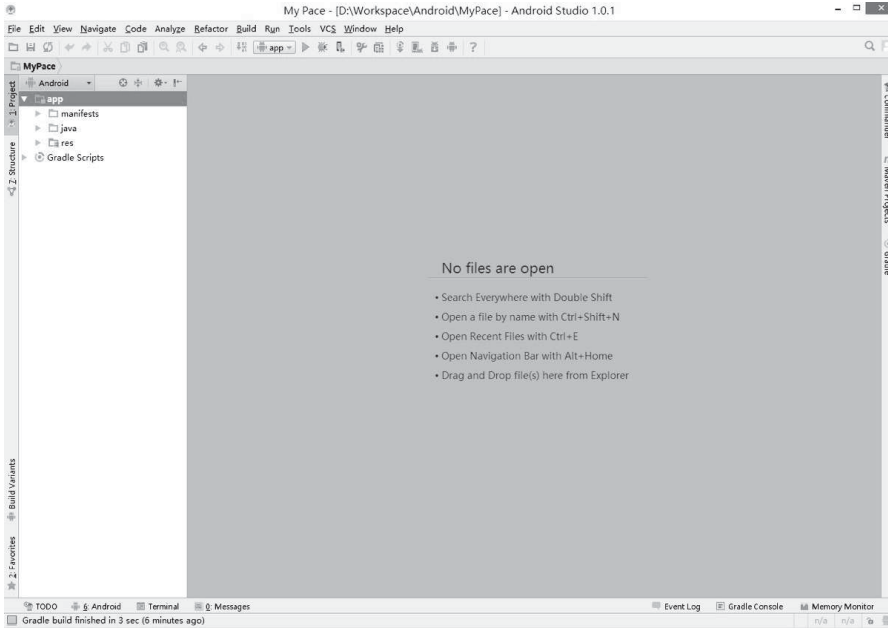


Figure 2.2 Blank interface of Android Studio.

## 2.3 INSTALLING ANDROID SDK

---

The Android SDK includes a comprehensive set of development tools. These tools include a debugger, libraries, a handset emulator, documentation, sample code, and tutorials. Using the installed IDE, Android SDK can be downloaded and installed conveniently.

In Android Studio, on the top of the screen, select the Tools menu, then Android, and then SDK Manager (Tools → Android → SDK Manager), as shown in Fig. 2.3. Then we can see the interface of Android SDK Manager, similar to Figure 2.4.

Install Android SDK Tools, Android SDK Platform-tools, at least one Android SDK Build-tools, and at least one Android API, as shown in Fig. 2.5. API is a set of routines, protocols, and tools for building

software applications. The Android 5.0.1 (API 21) is the newest version of Android SDK. We suggest installing Documentation for Android SDK, SDK Platform, ARM EABI v7a System Image, and Google APIs. The documentation for Android SDK can help solve programming problems. The ARM EABI v7a system image is a virtual mobile operating system image running on virtual devices.

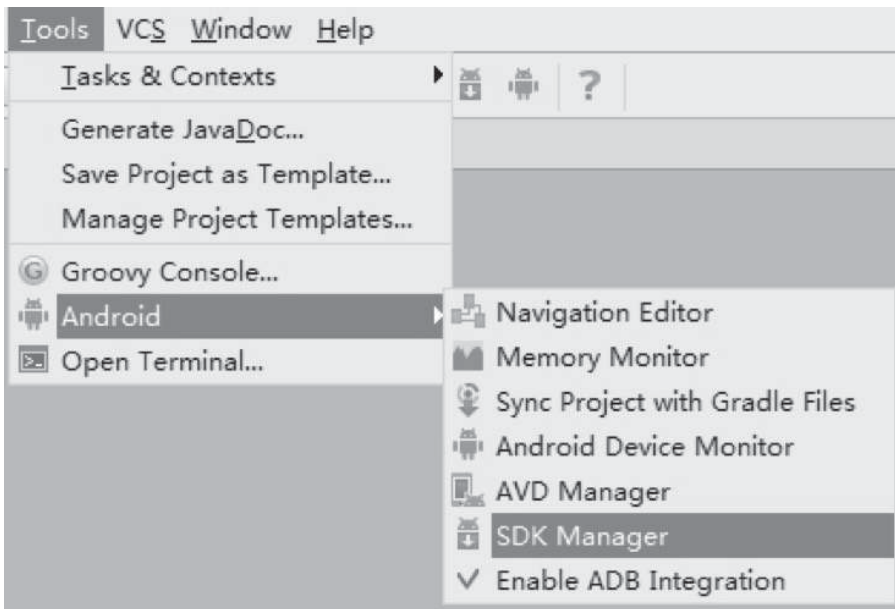


Figure 2.3 Android SDK Manager in Android Studio.

## 2.4 CREATING AN ANDROID APPLICATION

---

After installing the Android SDK, we can create our first Android Application.

On the top left corner of the Android Studio, select File, and then New Project (File → New Project). You will see the “Create

New Project” dialog. In the first step of creating a new Android application, type in the application name, such as “My Application,” as shown in Fig. 2.5. You can type in company domain, such as “my.android.example.com”. Furthermore, you can choose a directory to store your Android project.

In the second step of creating a new Android application, you can choose which kind of device your application runs on. You can choose more than one device, such as phone and tablet, TV, and Wear. In this Android application, only select “Phone and Tablet”, as shown in Fig. 2.6.

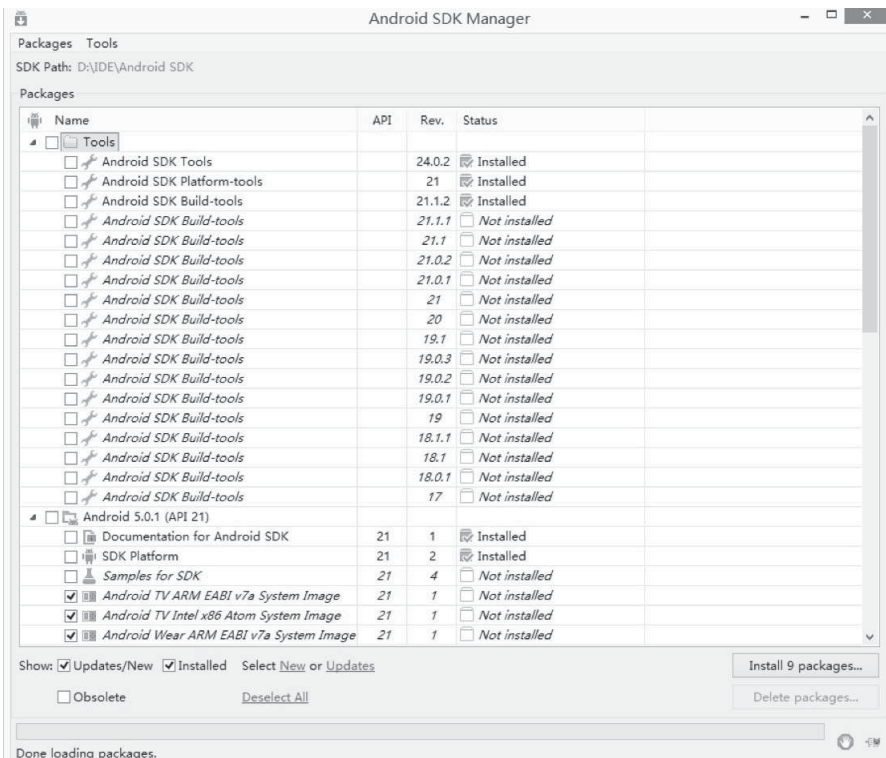


Figure 2.4 Details of Android SDK Manager in Android Studio.

In the third step of creating a new Android application, you can add an activity to your Android application, and you have many choices, such as blank activity, blank activity with fragment, fullscreen activity, Google maps activity, login activity, navigation drawer activity, setting activity, and tabbed activity, as shown in [Fig. 2.7](#). In the latest version of Android Studio, fragment was integrated into activity.

In the last step of creating a new Android application, you can change the name of the activity added in the third step, as shown in [Fig. 2.8](#). Then click “finish,” the interface of Android Studio will be similar to [Fig. 2.9](#).

## **2.5 ANDROID VIRTUAL DEVICE**

After creating the first Android application, we need to create an *Android Virtual Device* (AVD) to run it. First, on the top of the interface, select Tools, then Android, and then AVD Manager (Tools → Android → AVD Manager). The AVD Manager is similar to [Fig. 2.10](#).

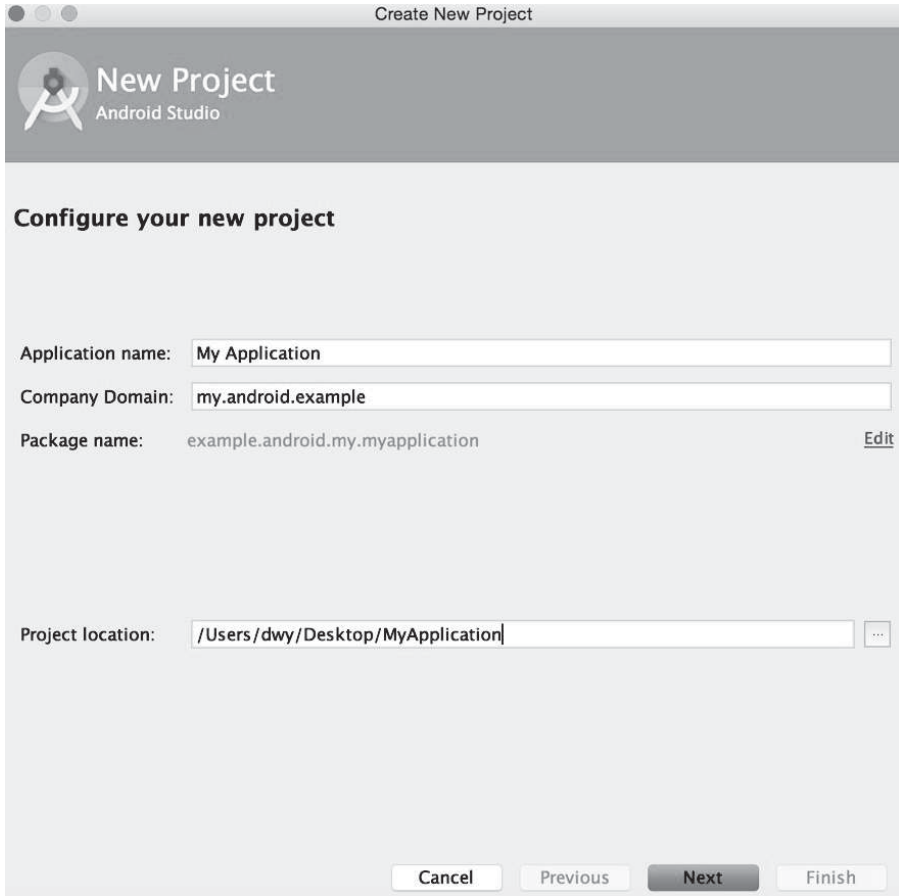


Figure 2.5 First Step of creating an Android Application in Android Studio.

Click “Create a virtual device”; the interface will be similar to Fig. 2.11. Choose *Phone* in the category list, and Nexus S as the device. Then click “Next.”

In the second step of creating an AVD in Android Studio, you can choose the version of Android SDK which you want to use, as shown in Fig. 2.12. Then click “Next.”

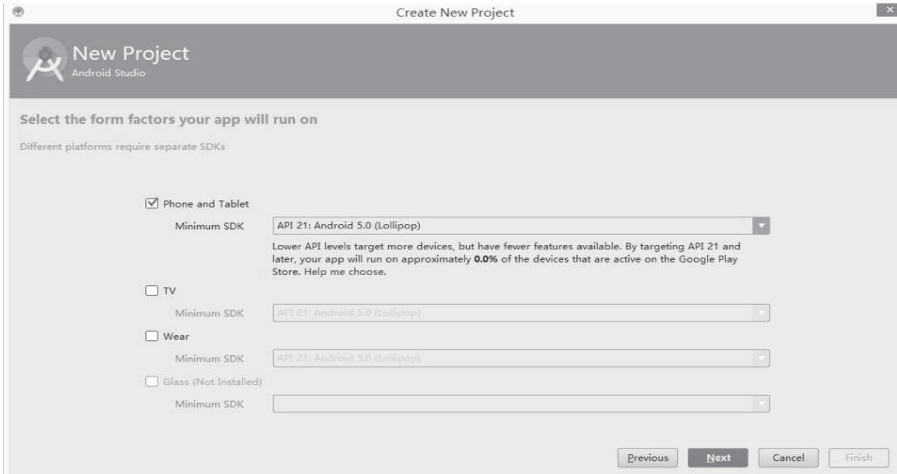


Figure 2.6 First step of creating an Android application in Android Studio.

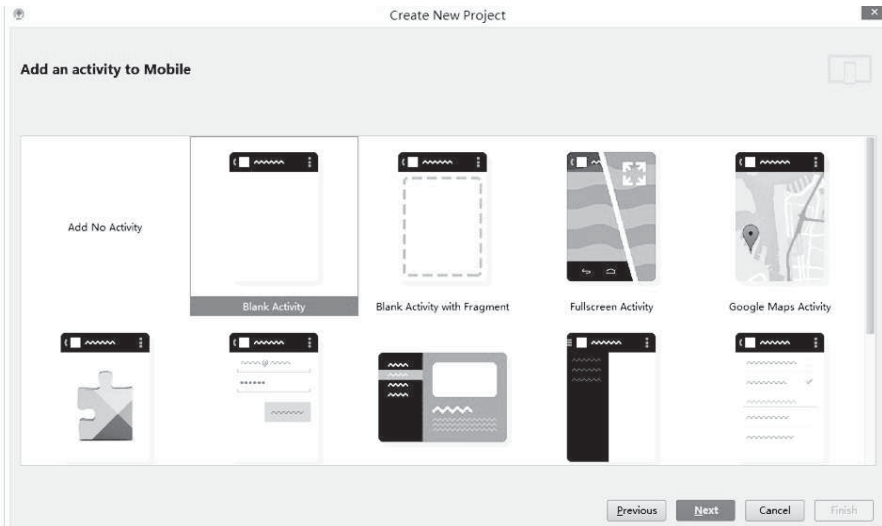


Figure 2.7 Second step of creating an Android application in Android Studio.

30 ■ Mobile Applications Development with Android

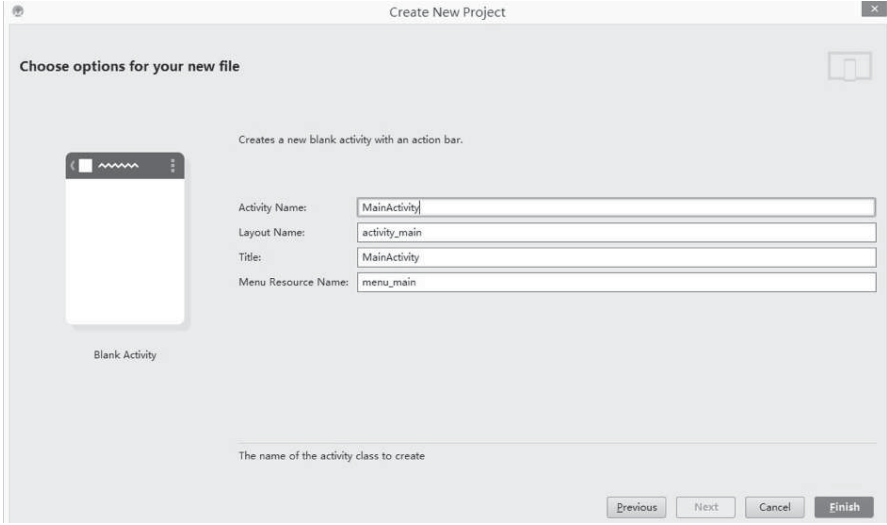


Figure 2.8 Last step of creating an Android application in Android Studio.

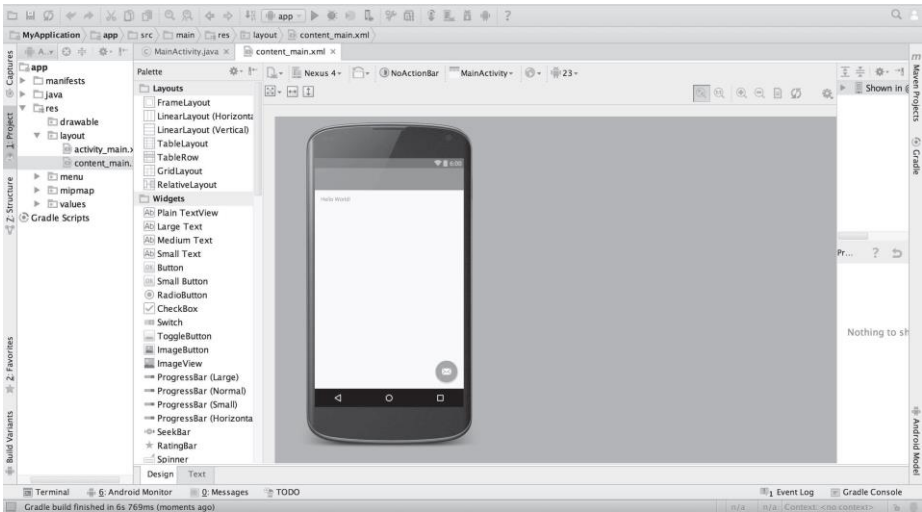


Figure 2.9 Interface of Android Studio with a new Android project.





Figure 2.10 Android Virtual Device Manager in Android Studio.

In the last step of creating an AVD in Android Studio, you can change the name of the AVD you want to create, as shown in Fig. 2.13. Then click “Finish”.

The AVD is created, as shown in Fig. 2.14. Then click the green arrow on the right side to start this virtual device. After waiting a while, the virtual device is started, as shown in Fig. 2.15.

Then run your Android application on this virtual device. Select the application created before, and then click the green arrow to that was run it. At last, the Android application runs on the virtual device, as shown in Fig. 2.16.

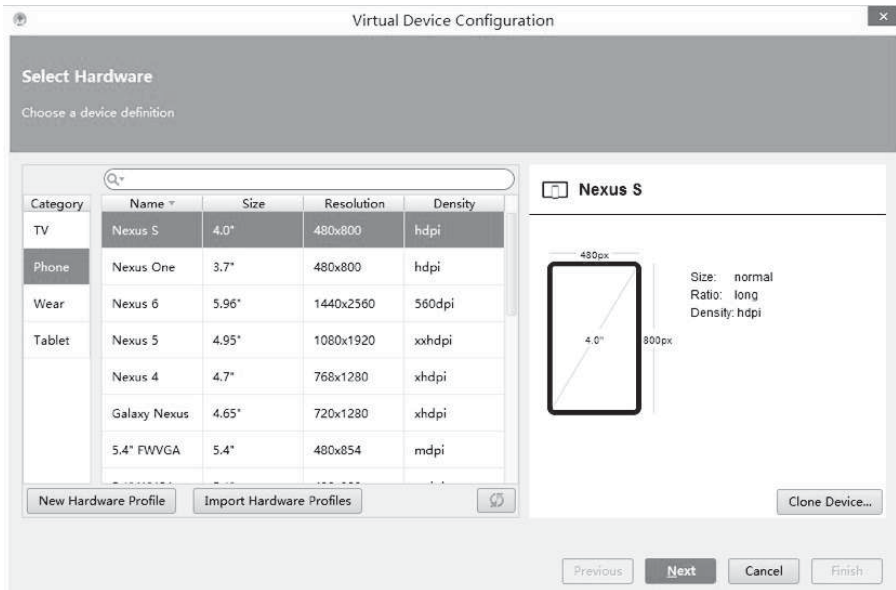


Figure 2.11 First Step of Creating an Android Virtual Device in Android Studio -1.

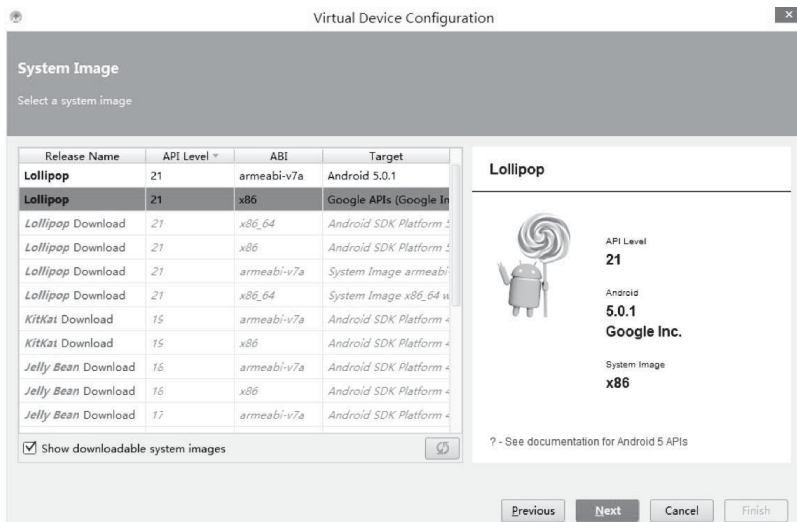


Figure 2.12 Second Step of Creating an Android Virtual Device in Android Studio -2.

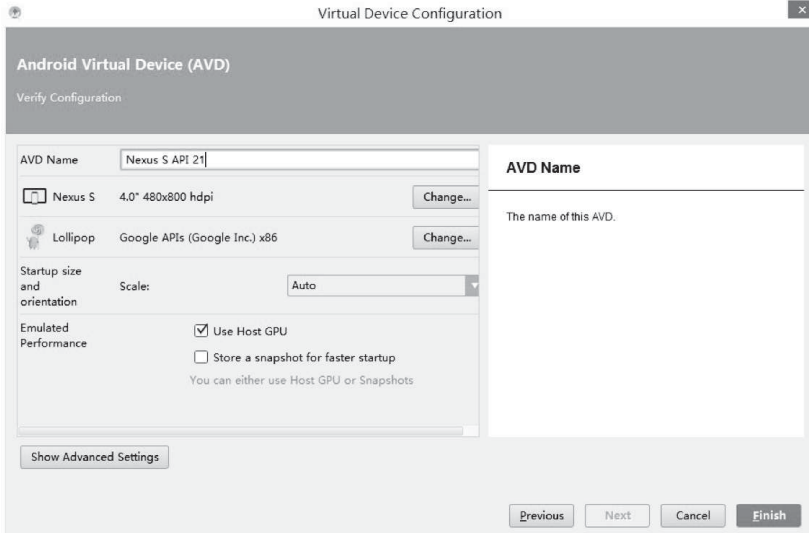


Figure 2.13 Last Step of Creating an Android Virtual Device in Android Studio.



Figure 2.14 New Virtual Device in AVD Manager in Android Studio.



Figure 2.15 Android Virtual Device in Android Studio

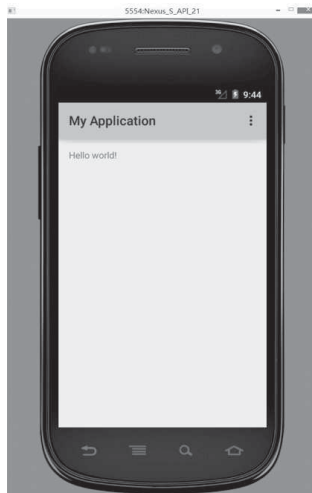


Figure 2.16 Android Application Running on the Android Virtual Device in Android Studio

## Chapter Three

### Introduction of Key Concepts of Android

#### CONTENTS

3.1	App	Components .....
3.1.1	Activities .....	
3.1.2	Services .....	
3.1.3	Content Providers .....	
3.1.4	Intents .....	
3.2	App	Resources .....
3.3	App	Mainfest .....
3.3.1	Elements .....	
3.3.2	Attributes .....	
3.3.3	Declaring Class Names .....	
3.3.4	Multiple Values .....	
3.3.5	Resource Values .....	
3.3.6	Sting Values .....	

Understand key concepts of Android is a basic requirement for designing Android mobile apps. In this chapter, we introduce some basic concepts of Android, including the app components, app resources, and app manifest. Students will able to answer the following questions after reading this chapter.

1. What is an activity in Android?
2. Can we directly save resource files inside the res/directory?
3. What is an *APP MAINFEST*?

## 3.1 APP COMPONENTS

---

App components are the essential building blocks of an Android app. Each component is a different point through which the system can enter your app. Not all components are actual entry points for the user, and some depend on each other, but each one exists as its own entity and plays a specific role. Each one is a unique building block that helps define your app's overall behavior.

The following subsections represent four types of app components, which include activities, services, content providers, and intents.

### 3.1.1 Activities

An activity represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities (if the email app allows it). For example, a camera app can start the activity in

the email app that composes new mail, in order for the user to share a picture. You can find more information about activities at <https://developer.android.com/guide/components/activities.html>.

### **3.1.2 Services**

A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. You can find more information about content providers at <https://developer.android.com/reference/android/app/Service.html>.

### **3.1.3 Content Providers**

A content provider manages a shared set of app data. You can store the data in the file system, a SQLite database, on the web, or any other persistent storage location your app can access. Through the content provider, other apps can query or even modify the data (if the content provider allows it). For example, the Android system provides a content

provider that manages the user's contact information. As such, any app with the proper permissions can query part of the content provider (such as `ContactsContract.Data`) to read and write information about a particular person.

Content providers are also useful for reading and writing data that is private to your app and not shared. For example, the Note Pad sample app uses a content provider to save notes. You can find more information about content provider at: <https://developer.android.com/reference/android/content/ContentProvider.html>

### **3.1.4 Intents**

An intent is a mechanism for describing a specific action, such as “pick a photo,” “phone home,” or “open the pod bay doors.” In Android, just about everything goes through intents, so you have plenty of opportunities to replace or reuse components. For example, there is an intent for “send an email.” If your application needs to send mail, you can invoke that intent. Or, if you are writing a new email application, you can register an activity to handle that intent and replace the standard mail program. The next time somebody tries to send an email, they'll get the option to use your program instead of the standard one.



You can find more information about intents at <https://developer.android.com/guide/components/intents-filters.html>.

A unique aspect of the Android system design is that any app can start another app's component. For example, if you want the user to capture a photo with the device camera, there's probably another app that does that and your app can use it, instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera app. Instead, you can simply start the activity in the camera app that captures a photo. When complete, the photo is even returned to your app so you can use it. To the user, it seems as if the camera is actually a part of your app.

When the system starts a component, it starts the process for that app (if it's not already running) and instantiates the classes needed for the component. For example, if your app starts the activity in the camera app that captures a photo, that activity runs in the process that belongs to the camera app, not in your app's process. Therefore, unlike apps on most other systems, Android apps don't have a single entry point (there's no `main()` function, for example).

Since the system runs each app in a separate process with file permissions that restrict access to other apps, your app cannot directly activate a component from another app. The Android system, however,

can. So, to activate a component in another app, you must deliver a message to the system that specifies your intent to start a particular component. The system then activates the component for you.

Intents can be used to activate activities and services, but content providers are activated when targeted by a request from a *ContentResolver*. There are separate methods for activating each type of component:

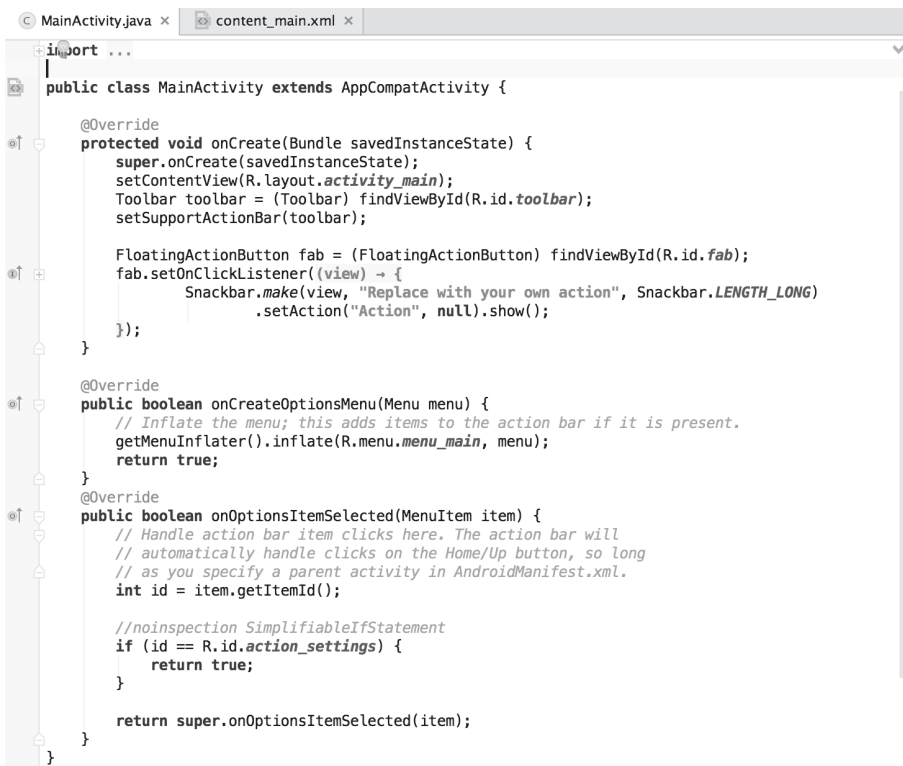
1. You can start an activity (or give it something new to do) by passing an *Intent* to *startActivity()* or *startActivityForResult()* (when you want the activity to return a result).
2. You can start a service (or give new instructions to an ongoing service) by passing an *Intent* to *startService()*. Or you can bind to the service by passing an *Intent* to *bindService()*.
3. You can perform a query to a content provider by calling *query()*

on a *ContentResolver*.

[Fig. 3.1](#) is the default *AndroidManifest.xml* generated by the Integrated Development Environment (IDE) after we create a blank Android application. The *Main Activity* is the only activity in this project, and it is a subclass of *Activity*, as “class *Main Activity* extends *Activity*” shown. In *Main Activity*, we need to implement callback methods that the

system calls when the activity transitions between various states of its life cycle.

The *onCreate()* method is indispensable, and it will be called when the *MainActivity* is created. Within the implementation of the *onCreate()* method, we should initialize the essential components of this activity. We must call *setContentView()* to define the layout for this activity's user interface. Although these processes are implemented by IDE, it is necessary for us to have this knowledge.



```
import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener((view) -> {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

Figure 3.1 Default MainActivity.java.

## 3.2 APP RESOURCES

You should always externalize resources, such as images and strings, from your application code, so that you can maintain them independently. You should place each type of resource in a specific subdirectory of your project's *res/* directory, as shown in Fig. 3.2.

*drawable/* file contains bitmap files, such as *png*, *jpg* and *gif*. *layout/* contains XML files that define a user interface layout. *menu/* contains XML files that define application menus.

*vaules/* contains XML files that contain simple values, such as strings, integers, and colors.

Besides the ones shown in Fig. 3.1, we can add some other resource files into *res/* directory, such as *animator/*, *raw/*, and *xml/* files.

The *animator* file contains Android property animations. The property animation system is a robust framework that allows you to animate almost anything. You can define an animation to change any object property over time, regardless of whether it draws to the screen or not. You can find more information about property animation at <http://developer.android.com/guide/topics/graphics/prop-animation.html>.



Figure 3.2 File hierarchy for a simple project.

The raw file stores any files in their raw form. You must call *Resource.openRawResource()* to open these resources with a raw *InputStream*.

The XML file contains arbitrary XML files that can be read at run- time by calling *Resource.getXML()*. Various XML configuration files must be saved here, such as a searchable configuration.

*HINT: Never save resource files directly inside the res/ directory, because it will cause a compiler error.*

### 3.3 APP MAINFEEST

The manifest file is indispensable in every Android application. The manifest file presents essential information about your app to the Android system, information the system must have before it can run any

of the app's code. Among other things, the manifest does the following:

1. It names the Java package for the application. The package name serves as a unique identifier for the application.
2. It describes the components of the application, the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement each of the components and publishes their capabilities (for example, which Intent messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.
3. It determines which processes will host application components.
4. It declares which permissions the application must have in order to access protected parts of the Application Programming Interface (API) and interact with other applications.
5. It also declares the permissions that others are required to have in order to interact with the application's components.
6. It lists the Instrumentation classes that provide profiling and other information as the application is running. These declarations are

present in the manifest only while the application is being developed and tested; they're removed before the application is published.

7. It declares the minimum level of the Android API that the application requires.
8. It lists the libraries that the application must be linked against.

### **3.3.1 Elements**

Only the <manifest> and <application> elements are required, they each must be present and can occur only once. Most of the others can, occur many times or not at all, although at least some of them must be present for the manifest to accomplish anything meaningful.

If an element contains anything at all, it contains other elements. All values are set through attributes, not as character data within an element. Elements at the same level are generally not ordered. For example,

<activity>, <provider>, and <service> elements can be intermixed in any sequence. (An <activity-alias> element is the exception to this rule:

It must follow the <activity> it is an alias for.)

### **3.3.2 Attributes**

In a formal sense, all attributes are optional. However, there are some that must be specified for an element to accomplish its purpose. Use the documentation as a guide. For truly optional attributes, it mentions a default value or states what happens in the absence of a specification. Except for some attributes of the root <manifest> element, all attribute names begin with an android: prefix, for example, android:alwaysRetainTaskState. Because the prefix is universal, the documentation generally omits it when referring to attributes by name.

### **3.3.3 Declaring Class Names**

Many elements correspond to Java objects, including elements for the application itself (the <application> element) and its principal components, activities (<activity>), services (<service>), broadcast receivers (<receiver>), and content providers (<provider>).

If you define a subclass, as you almost always would for the component classes (Activity, Service, BroadcastReceiver, and Content-Provider), the subclass is declared through a name attribute. The name must include the full package designation. However, as a shorthand, if the first character of the string is a period, the string is appended to the



application's package name (as specified by the <manifest> element's package attribute).

When starting a component, Android creates an instance of the named subclass. If a subclass isn't specified, it creates an instance of the base class.

### **3.3.4 Multiple Values**

If more than one value can be specified, the element is almost always repeated, rather than listing multiple values within a single element.

### **3.3.5 Resource Values**


Some attributes have values that can be displayed to users, for example, a label and an icon for an activity. The values of these attributes should be localized and therefore set from a resource or theme.

The package name can be omitted if the resource is in the same package as the application, type is a type of resource, such as "string" or "drawable," and name is the name that identifies the specific resource.

### **3.3.6 Sting Values**

Where an attribute value is a string, double backslashes ("\\") must be used to escape characters, for example, '\\n' for a newline or '\\uxxxx' for

a Unicode character.



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.csis.pace.edu.mypace" >

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="My Pace "
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="My Pace " >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Figure 3.3 Default AndroidManifest.xml

Fig. 3.3 is the default AndroidManifest.xml file generated by IDE after a blank Android application. In the third creating line, “com.example.csis.pace.edu.mypace,” is the package name of the project, and it exactly as the same as the first line of *MainActivity.java*. Many elements inside the <application> and </application> correspond to Java objects, including activities (<activity>), services (<service>), broadcast receivers (<receiver>), and content providers (<provider>). In our project, we only create an activity, thus, there is

only one `<activity>` in *AnroidManifest.xml* file. In this `<activity>`, the `android:name=".MainActivity"` shows the name of the activity.

The `<intent-filter>` specifies the type of intents that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of its parent component, what an activity or service can do and what types of broadcasts a receiver can handle. It opens the component to receiving intents of the advertised type, while filtering out those that are not meaningful for the component.

When adding an action to an intent filter. An `<intent-filter>` element must contain one or more `<action>` elements. If it doesn't contain any, no Intent objects will get through the filter. Some standard actions are defined in the Intent class as `ACTION_string` constants. To assign one of these actions to this attribute, prepend "android.intent.action." to the string that follows `ACTION_`. In our project, use "android.intent.action.MAIN" for `ACTION_MAIN`.

The `<category>` is used to add a category name to an intent filter. Standard categories are defined in the Intent class as `CATEGORY_name` constants. The name assigned here can be derived from those constants by prefixing "android.intent.category." to the name that follows `CATEGORY_`. In our project, the string value is "android.intent.category.LAUNCHER" for `CATEGORY_LAUNCHER`.

## Chapter Four

### 2 D Graphics and Multimedia in Android

---

#### CONTENTS

---

4.1	Introduction of 2-D Graphics Techniques.....
4.1.1	Color.....
4.1.2	Paint.....
4.1.3	Path.....
4.1.4	Canvas.....
4.1.5	Drawable.....
4.1.6	Button Selector.....
4.2	Advanced UI Design.....
4.2.1	Multiple Screens.....
4.2.2	Action Bar.....
4.2.3	Custom Views.....
4.3	Overview of Multimedia in Android.....
4.3.1	Understanding the MediaPlayer Class.....
4.3.2	Life Cycle of the MediaPlayer State.....
4.4	Audio Implementations in Android.....
4.5	Executing Video in Android.....

2-D Graphics and UI Design are two important aspects in User Interface (UI) design. In this chapter, we will introduce 2-D graphics and some advanced UI design techniques. Main techniques of 2-D graphics include **Color**, **Paint**, **Path**, **Canvas**, **Drawable**, and **Button Selector**. Students will also learn how to create multiple screens, action bars, and custom views on the UI. Moreover, multimedia on Android systems is a functionality increasing your mobile apps' adoptability. In this chapter, we will introduce multimedia in

Android and how to add multimedia to our Android app. Three main aspects in multimedia include *Media*, *Audio*, and *Video*.

## **4.1 INTRODUCTION OF 2 D GRAPHICS TECHNIQUES**

Android implements complete 2-D functions in one package, named `android.graphics`. This package provides various kinds of graphics tools, such as canvas, color filter, point, line, and rectangles. We can use these graphics tools to draw the screen directly. We will introduce some basic knowledge in detail. First of all, we create a new Android application project named ColorTester.

### **4.1.1 Color**

Colors are represented as packed integers, made up of 4 bytes: Alpha, Red, Green, and Blue. Alpha is a measure of transparency, from value 0 to value 255. The value 0 indicates the color is completely transparent. The value 255 indicates the color is completely opaque. Besides alpha, each component ranges between 0 and 255, with 0 meaning no contribution for that component, and 255 meaning 100% contribution.

We can create a half-opaque purple color like: `int color1 = Color.argb(127, 255, 0, 255);`

Or in XML resource file, like:

```
<color name="half_op_purple">#7fff00ff</color>
```

The colors in Android XML resource files must be formulated as “#” + 6 or 8 bit Hexadecimal number.

Furthermore, Android offers some basic colors as constants, as shown in [Fig. 4.1](#). We can use them directly, like:

```
int color2 = Color.Black;
```

In Android Studio, we can preview the color we created in XML file, as shown in [Fig. 4.2](#). There are some small squares with the color created in the same line. We can see that the `#ffffffff` is opaque-white, and the `#ff000000` is opaque-black.









































  BLUE	int
  GRAY	int
  LTGRAY	int
  parseColor (String colorString)	int
  HSVToColor (float[] hsv)	int
  HSVToColor (int alpha, float[] hsv)	int
  alpha (int color)	int
  BLACK	int
  blue (int color)	int
  CYAN	int
  DKGRAY	int
  GREEN	int
  green (int color)	int
  MAGENTA	int
  RED	int
  red (int color)	int
  rgb (int red, int green, int blue)	int
  TRANSPARENT	int
  WHITE	int
  YELLOW	int

Figure 4.1 Colors as constants provided by Android.

We can use these color, created in colors.xml by “color/color\_name”.

For example, android:background=“color/my\_color”.

After we define some colors in the XML file, we can reference them by their names, as we did for strings, or we can use them in Java code like:

```
int color3 = getResources().getColor(R.color.my_color); or
```

```
int color3 = R.color.text_color
```

The *getResources()* method returns the *ResourceManager* class for the current activity, and *getColor()* asks the manager look up a color

given a resource ID.



```
<resources>
  <color name="my_color">#7fff00ff</color>
  <color name="puzzle_background">#ffff0000</color>
  <color name="puzzle_hi_lite">#ffffffff</color>
  <color name="puzzle_light">#64c6d4ef</color>
  <color name="puzzle_dark">#6456648f</color>
  <color name="puzzle_foreground">#ff000000</color>
  <color name="puzzle_hint_0">#64ff0000</color>
  <color name="puzzle_hint_1">#6400ff80</color>
  <color name="puzzle_hint_2">#2000ff80</color>
  <color name="puzzle_selected">#64ff8000</color>
</resources>
```

Figure 4.2 Preview of colors in XML files in Android Studio.

### 4.1.2 Paint

The *Paint* class holds the style and color information on drawing geometries, text, and bitmaps. Before we draw something on the screen, we can set color to a *Paint* via *setColor()* method.

```
Paint cPaint = new Paint();
cPaint.setColor(Color.LTGRAY);
Paint tPaint = new Paint();
tPaint.setColor(Color.BLUE);
tPaint.setTextSize((float) 20.0);
```

Figure 4.3 Paint class in Android.

As shown in Fig. 4.3, we create two *Paints*, which are *cPaint* to draw a circle and *tPaint* to draw text. We set the color of the circle as light gray and the color of text as blue. Beside colors, we also can set other attributes to *Paint* class, such as the *TextSize*.



### 4.1.3 Path

The *Path* class encapsulates multiple contour geometric paths, such as lines, rectangles, circles, and curves. Fig. 4.4 is an example that defines a circular path and a rectangle path.

The second line defines a circle, whose center is at position  $x=300$ ,  $y=200$ , with a radius of 150 pixels. The fourth line defines a rectangle whose left top point is at position  $x=150$ ,  $y=400$ , and right bottom point is at position  $x=400$ ,  $y=650$ . The *Path.Direction.CW* indicates that the shape will be drawn clockwise. The other direction is *CCW*, which indicates counter-clockwise.

```
Path path = new Path();
path.addCircle(300, 200, 150, Path.Direction.CW);

Path path2 = new Path();
path2.addRect(150, 400, 400, 650, Path.Direction.CW);
```

Figure 4.4 Create two *Path* object and add details to them.

### 4.1.4 Canvas

To draw something, we need to prepare four basic components, including a *Bitmap* to hold the pixels, a *Canvas* to host the draw call, a drawing primitive, and a *Paint*. The *Bitmap* is the place where to draw something, and the *Canvas* is used to hold the “draw” calls. A drawing primitive can

be a Rect, a Circle, a Path, a Text, and a Bitmap.

In Android, a display screen is taken up by an Activity, which hosts a *View*, which in turn hosts a Canvas. We can draw on the canvas by overriding the *View.onDraw()* method. A Canvas object is the only parameter to *onDraw()* method. We create a new activity, which contains a view called *GraphicsView*, but not the layout.xml, as shown in [Fig. 4.8](#).

In [Fig. 4.5](#), we comment the original code and set the content view of this activity to some layout.xml, and set it to some new view we created, which is *GraphicsView*.

Let's review the two methods of designing Android apps. There are two methods to design Android apps, which are procedural and declarative. The “`setContentView(R.layout.activity_main)`” is a typical example of declarative, which is described all objects in the activity using XML files. The “`setContentView(new GraphicsView(this))`” is a typical example of a procedural, which means writing Java code to create and manipulate all the user interface objects [56].

This new class, *GraphicsView*, extends the class *View*. The *onDraw()* method is over-rider and used to implement the function of drawing. [Fig. 4.6](#) shows the details of the *onDraw()* method. We use Paint with different colors to draw a Path on the View via calling

*onDraw(Canvas)* method.

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    // super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    setContentView(new GraphicsView(this));
}

static public class GraphicsView extends View {
    public GraphicsView(Context context){
        super(context);
    }

    @Override
    protected void onDraw(Canvas canvas){...}
}

```

Figure 4.5 A new activity whose content view is the view created ourselves but not layout.xml.

Meanwhile, we have another choice to create a Canvas, as shown in Fig. 4.8. In Fig. 4.8, we create a Bitmap that is a square whose size is 100\*100 and will use it as the argument of Canvas. Then we can use this canvas as the same as the one offered in the onDraw() method.

### 4.1.5 Drawable

*Android.graphics.drawable* provides classes to manage a variety of visual elements, which are intended for display, such as bitmaps and gradients. We can combine drawables with other graphics, or we can use them in UI widgets, such as the background for a button. Android offers

following types of drawables:

*Bitmap*: A bitmap graphic file (.png, .jpg, or .gif).

*Nine-Patch*: A PNG file with stretchable regions to allow image resizing based on content (.9.png).

*Layer*: A *Drawable* that manages an array of other *Drawables*. These are drawn in array order, so the element with the largest index is be drawn on top.

```
@Override
protected void onDraw(Canvas canvas){

    String QUOTE = "PACE UNIVERSITY CSIS DEPT.";

    Paint cPaint = new Paint();
    cPaint.setColor(Color.LTGRAY);
    Paint tPaint = new Paint();
    tPaint.setColor(Color.BLUE);
    tPaint.setTextSize((float) 20.0);

    Path path = new Path();
    path.addCircle(300, 200, 150, Path.Direction.CW);

    Path path2 = new Path();
    path2.addRect(150, 400, 400, 650, Path.Direction.CW);

    canvas.drawPath(path, cPaint);
    canvas.drawTextOnPath(QUOTE, path, 0, 20, tPaint);

    canvas.drawPath(path2, tPaint);
}
```

Figure 4.6 The “onDraw()” method that draws a circle and a rectangle.



Figure 4.7 Running result of GraphicsView.

```
//Creating a new Canvas object
Bitmap bitmap = Bitmap.createBitmap(100,100,Bitmap.Config.ARGB_8888);
Canvas canvas = new Canvas(bitmap);
```

Figure 4.8 Use Bitmap to create a new Canvas.

*State:* An XML file that references different bitmap graphics for different states (for example, to use a different image when a button is pressed).

*Level:* An XML file that defines a drawable that manages a number of alternate *Drawables*, each assigned a maximum numerical value. Creates a *LevelListDrawable*.

*Transition:* An XML file that defines a drawable that can cross-fade

between two drawable resources.

*Inset Drawable:* An XML file that defines a drawable that insets another drawable by a specified distance. This is useful when a View needs a background drawable that is smaller than the View’s actual bounds.

*Clip:* An XML file that defines a drawable that clips another *Drawable* based on this *Drawable*’s current level value.

*Scale:* An XML file that defines a drawable that changes the size of another *Drawable* based on its current level value.

*Shape:* An XML file that defines a geometric shape, including colors and gradients.

A drawable resource is a general concept for a graphic that can be drawn to the screen and that can be retrieved with *Application Programming Interface* (API). Now we will add a gradient background to our ColorTester. We create a drawable resource file in `res\drawable`, and then create a *Shape* inside the `background.xml` file, as shown in [Fig. 4.9](#) and [Fig. 4.10](#).



[Figure 4.9](#) The first step of creating a new Drawable resource file.

As shown in [Fig. 4.11](#), we define a gradient from the start color to the end color. The angle indicates the direction of the gradient, and it must

be the extract times 45. When the angle = 0, the sequence is from left to right. When the angle = 90, the sequence is from bottom to top. When the angle = 180, the sequence is from right to left. When the angle = 270, the sequence is from top to bottom.

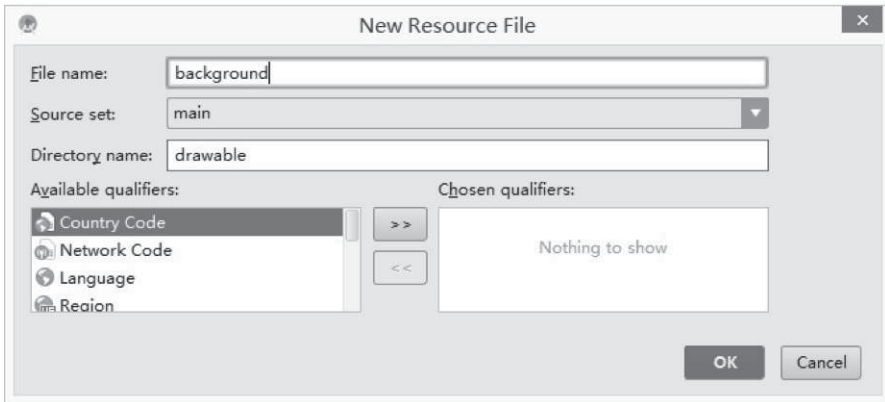


Figure 4.10 The second step of creating a new Drawable resource file.

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
  <gradient
    android:startColor="#FFFFFF"
    android:endColor="#aed130"
    android:angle="90" />
</shape>
```

Figure 4.11 Shape Drawable.

Then add one attribute into the RelativeLayout in `activity_main.xml` as “`android:background:@drawable/background`”. The running result is shown in [Fig. 4.12](#).

Besides gradient, there are some other common attributes that can be

added into a shape, including stroke, corners, and padding. We add them into the shape of background.xml, and set the color of the stroke is red, the width of the dash is 10dp, etc. The attributes and the running result are shown in Fig. 4.13. From Fig. 4.13, we can see that the background is stroked by a red dash, and every corner has a round edge.



Figure 4.12 Gradient background.





Figure 4.13 Stroke, Corners, and Padding Drawables

### 4.1.6 Button Selector

We want to set different colors to buttons when they are at different states. We set the default color of a button as light purple, and the color when it is pressed is light orange. As we introduced in the previous section, we need to create a drawable resource file to implement this function. Thus, we create a new drawable resource file named “**button\_selection**”, and between the `<selector>` and `</selector>` add two items. The first one is the pressed state, which indicates that the button is pressed, as shown in Fig. 4.14. The second one is the default state, as shown in Fig. 4.15.

```

<item android:state_pressed="true" >
  <shape>
    <gradient
      android:startColor="#ffc2b7"
      android:endColor="#FFFFFF"
      android:type="radial"
      android:gradientRadius="50" />
    <stroke
      android:width="2dp"
      android:color="#dcdcdc"
      android:dashWidth="5dp"
      android:dashGap="3dp" />
    <corners
      android:radius="2dp" />
    <padding
      android:left="10dp"
      android:top="10dp"
      android:right="10dp"
      android:bottom="10dp" />
  </shape>
</item>

```

Figure 4.14 Default state of button.

Then, we add one attribute to all the three buttons as follows:

`android:background="@drawable/button_selector"`.

The running result is shown in [Fig. 4.16](#).

## 4.2 ADVANCED UI DESIGN

Android provides a flexible framework for UI design that allows apps to display different layouts for different devices, create custom UI widgets, and control aspects of the system UI beyond the apps' window.

### 4.2.1 Multiple Screens

The goal of this part is to build a UI, which is flexible enough to fit perfectly on any screen and to create different interaction patterns that are optimized for different screen sizes.

To ensure that your layout is flexible and adapts to different screen sizes, you should use “wrap\_content” and “match\_parent” for the width and height of some view components. If you use “wrap\_content”, the width or height of the view is set to the minimum size necessary to fit the content within that view, while “match\_parent” (also known as “fill\_parent” before API level 8) makes the component expand to match the size of its parent view.

```

<item android:state_focused="false">
  <shape>
    <solid android:color="#8f0000f0" />
    <stroke
      android:width="2dp"
      android:color="#fad3cf" />
    <corners
      android:topRightRadius="5dp"
      android:bottomLeftRadius="5dp"
      android:topLeftRadius="0dp"
      android:bottomRightRadius="0dp"
    />
    <padding
      android:left="10dp"
      android:top="10dp"
      android:right="10dp"
      android:bottom="10dp" />
  </shape>
</item>

```

Figure 4.15 Pressed state of the button.

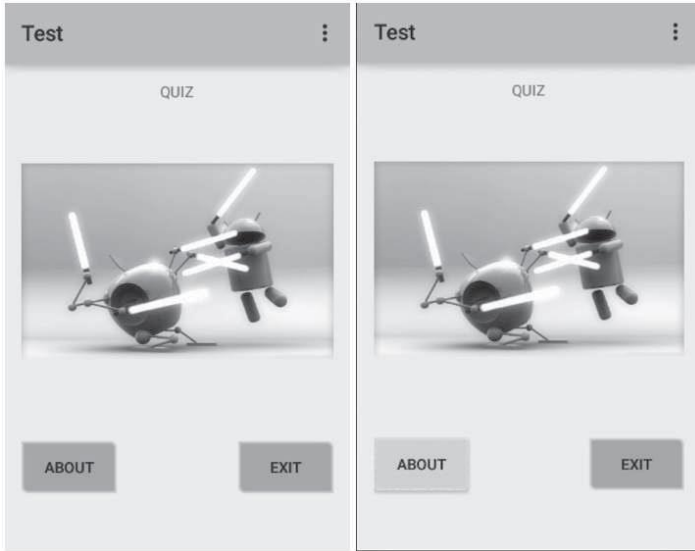


Figure 4.16 Running result of the button selector.

You can construct fairly complex layouts using nested instances of `LinearLayout` and combinations of “`wrap_content`” and “`match_parent`” sizes. However, `LinearLayout` does not allow you to precisely control the spacial relationships of child views; views in a `LinearLayout` simply line up side by side. If you need child views to be oriented in variations other than a straight line, a better solution is often to use a `RelativeLayout`, which allows you to specify your layout in terms of the special relationships between components. For instance, you can align one child view on the left side and another view on the right side of the

screen.

Supporting different screen sizes usually means that your image resources must also be capable of adapting to different sizes. For example, a button background must fit whichever button shape it is applied to. If you use simple images on components that can change size, you will quickly notice that the results are somewhat less than impressive, since the runtime will stretch or shrink your images uniformly. The solution is using nine-patch bitmaps, which are specially formatted PNG files that indicate which areas can and cannot be stretched.

Therefore, when designing bitmaps that will be used on components with variable sizes, always use nine-patches. To convert a bitmap into a nine-patch, you can start with a regular image. Then run it through the `draw9patch` utility of the Software Development Kit (SDK) (which is located in the `tools/` directory), in which you can mark the areas that should be stretched by drawing pixels along the left and top borders. You can also mark the area that should hold the content by drawing pixels along the right and bottom borders. The process is shown from [Fig. 4.17](#) to [Fig. 4.18](#).



Figure 4.17 Original image (.png).

The black pixels are along the borders. The ones on the top and



Figure 4.18 Nine-patch image (.9.png).

left borders indicate the places where the image can be stretched, and the ones on the right and bottom borders indicate where the content should be placed.

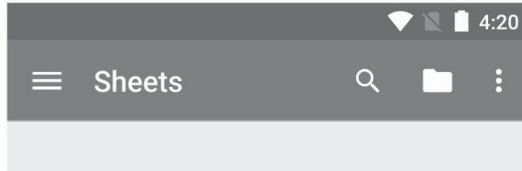


Figure 4.19 A nine-patch image used in various sizes.

### 4.2.2 Action Bar

The action bar, also called app bar, is one of the most important design elements in activities. It provides a visual structure and interactive

elements that are familiar to users. A typical action bar is shown in [Fig. 4.20](#).



[Figure 4.20](#) A typical action bar.

An action bar has some key functions listed as follows:

1. Dedicated space for giving the app an identity and indicating the user’s virtual location in the app.
2. Access to important actions in a predictable way, such as search.
3. Support for navigation and view switching (with tabs or drop-down lists).

In its most basic form, the action bar displays the title for the activity on one side and an overflow menu on the other. Beginning with Android 3.0 (API level 11), all activities that use the default theme have an ActionBar as an app bar. However, app bar features have gradually been added to the native ActionBar over various Android releases. As a result, the native ActionBar behaves differently depending on what version of the Android system a device may be using. By contrast, the most recent

features are added to the support library's version of Toolbar, and they are available on any device that can use the support library.

For this reason, you should use the support library's Toolbar class to implement your activities' app bars. Using the support library's toolbar helps ensure that your app will have consistent behavior across the widest range of devices. For example, the Toolbar widget provides a material design experience on devices running Android 2.1 (API level 7) or later, but the native action bar doesn't support material design unless the device is running Android 5.0 (API level 21) or later.

### **4.2.3 Custom Views**

Android has a large set of view classes for interacting with users and displaying various types of data. However sometimes we have some unique requirements that are not covered by the built-in views. To be a well-designed class, a custom view should:

1. conform to Android standards;
2. provide custom styleable attributes that work with Android XML layouts;
3. send accessibility events;
4. be compatible with multiple Android platforms.

All of the view classes defined in the Android framework extend the View.



The custom view can also extend View directly, or we can extend some existing view subclasses, such as Button. Then we need to define some attributes for the custom view. To define custom attributes, add `<declare-styleable>` resources to our project. It's customary to put these resources into a `res/values/attrs.xml` file.

After a view is created from an XML layout, all of the attributes in the XML tags are read from the resource bundle and passed into the view's constructor as an AttributeSet. Then we will pass the AttributeSet to `obtainStyledAttributes()`. This method passes back a TypedArray array of values that has already been dereferenced and styled.

Then we need to add properties and events to the custom view. To provide dynamic behavior, we need to expose a property getter and setter pair for each custom attribute, for example, showing text and image. After creating and initiating the custom view, we move to the most important part of a custom view, which is its appearance. Furthermore, the most important step in drawing a custom view is to override the `onDraw()` method. The parameter to `onDraw()` is a Canvas object that the view can use to draw itself. The Canvas class defines methods for drawing text, lines, bitmaps, and many other graphics primitives. You can use these methods in `onDraw()` to create your custom UI.

Drawing a UI is only one part of creating a custom view. You also need to make your view respond to user input in a way that closely resembles the real-world action you're mimicking. We need to make the view interactive, including input gestures, physically plausible motion, and making transactions smooth.

### 4.3 OVERVIEW OF MULTIMEDIA IN ANDROID

#### 4.3.1 Understanding the Media Player Class

Android support audio and video output through the Media Player class in the android.media package. The android.media is used to manage various media interfaces. The Media APIs are used to play and record media files, including audio and video . The Media Player class can be used to control playback of audio/video files and streams. The control of audio/video files and streams is managed as a state machine, as shown in [Fig. 4.21](#).

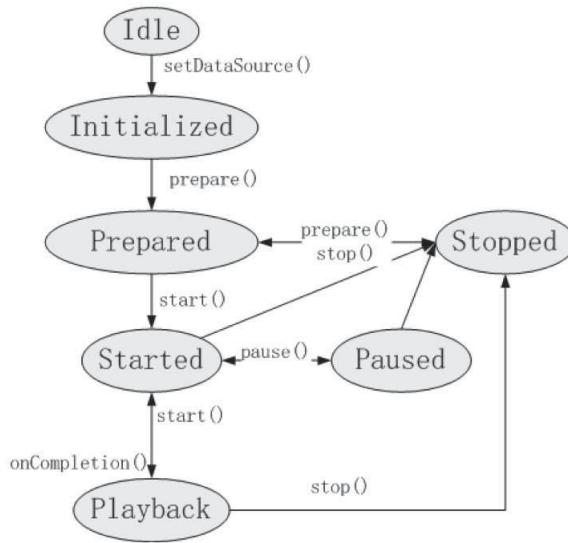


Figure 4.21 State diagram of the MediaPlayer.

### 4.3.2 Life Cycle of the MediaPlayer State

The life cycle and the state of a *MediaPlayer* object are driven by the supported playback control operations. The *setDataSource()* method is called to transfer a *MediaPlayer* object from the idle state to the initialized state. A *MediaPlayer* object must enter the prepared state first before it is started and played back. A *MediaPlayer* can enter the prepared state by call the *prepare()* or *prepareAsync()* method. The *prepare()* method transfers the object to the prepared state once the method call is returned. The *prepareAsync()* method first transfers the object to the preparing state after the call returns while the internal player engine continues working to complete the rest of the preparation work.

The *start()* method must be called to start the playback. The *MediaPlayer* object is in the started state, after *start()* returns. Calling *start()* has no effect on a *MediaPlayer* object that is already in the started state.

#### **4.4 AUDIO IMPLEMENTATIONS IN ANDROID**

---

To learn how to play audio, we create a new project named “MediaTester” and keep other configuration default. Then we copy one song from local directory to “MediaTester\app\src\main\res\raw” directory. Notice that we need to ensure that the file format can be recognized by Android. Fortunately, Android supports most all kinds of audio file formats. However, if Android does not support the file format of your audio, try to transform it to a common format.

First, create two buttons to show the “start” and “pause” functions. As introduced in the previous chapter, we create two buttons in the *activity\_main.xml*, as shown in [Fig. 4.22](#). Meanwhile, we need to add two strings in the *strings.xml* file as:

```
< stringname = “start_button” > Start < /string >
```

```
< stringname = “pause_button” > P ause < /string >
```

```

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/start_button"
    android:id="@+id/button_start"
    android:layout_below="@+id/textView"
    android:layout_alignParentStart="true" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/pause_button"
    android:id="@+id/button_pause"
    android:layout_below="@+id/button_start"
    android:layout_alignParentStart="true" />
    
```

Figure 4.22 Creating two buttons in activity\_main.xml.

Then jump into the MainActivity.java file and add a new Medi- aPlayer object into the MainActivity class as:

```
private MediaPlayer mp
```

Then we modify the MainActivity to implement OnClickListener, as introduced in previous chapter, and create onClick(View v) method to implement the functions of these two buttons. Then set OnClickListener to these two buttons, and now the MainActivity.java is shown as

Fig. 4.23.

```

public class MainActivity extends Activity implements View.OnClickListener {

    private MediaPlayer mp;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        findViewById(R.id.button_1).setOnClickListener(this);
        findViewById(R.id.button_2).setOnClickListener(this);
    }

    public void onClick(View v)
    {...}
}
    
```

[Figure 4.23](#) MediaPlayer object and the OnClickListener.

Then in the *onClick()* method, we implement functions to these two buttons, which are start a song, pause, and resume it. Before starting a song, we need to create a resource to the *MediaPlayer* object. Then we need to tell the computer which audio we want to play. We can use an integer ID of audio resource to identify the audio resource. In our example, we use `resId = R.raw.test`, then we call the *start()* method to play music. Before use pause a song, we need to judge whether it is playing. If it is playing, we call *pause()* method to pause it; if not, we call *start()* method to resume it. The code is shown in [Fig. 4.24](#).

The running result is shown in [Fig. 4.25](#). When we click the “START” button, Android plays the song that we previously put in the raw file. When we click the “PAUSE” button, Android will pause the song if it is playing or resume it if it is paused.

## **4.5 EXECUTING VIDEO IN ANDROID**

The *MediaPlayer* class works with video the same way it does with audio. However, we need to create a surface to play video, and the surface is *VideoView* class. The *VideoView* class can load images from various sources and takes charge of computing its measurement from the video. Furthermore, it provides various display options, such as

scaling.

*HINT: `VideoView` does not retain its full state when going into the background. It does not restore the current play state, position, selected tracks, or any subtitle tracks.*

We will add something about video into the `MediaTester` project.

```
public void onClick(View v) {
    int resId;
    switch (v.getId()) {
        case R.id.button_start:
            resId = R.raw.test;
            if (mp != null) {
                mp.release();
            }
            mp = MediaPlayer.create(this, resId);
            mp.start();
            break;
        case R.id.button_pause:
            if (mp.isPlaying()) {
                mp.pause();
            } else {
                mp.start();
            }
            break;
    }
}
```

Figure 4.24 Implementing the functions of two buttons in `onClick()`

method.



Figure 4.25 Running result of the MediaTester.

First, we create a new `VideoView` below the pause button in `activity_main.xml` as follows:

```
<VideoView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/video"
    android:layout_below="@+id/button_pause"
    android:layout_gravity="center"/>
```

Then, jumping into Java file, we create a `View` object named `video` and connect it to the `VideoView` as follows:

```
VideoView video = (VideoView) findViewById(R.id.video);
```

Then we need to set a path to identify the location of the video.

However, the *Android Virtual Device* (AVD) cannot recognize the local



path in our computer. Android offers several options to store persistent application data as follows:

*Shared Preferences* The `SharedPreferences` class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types.

*Internal Storage* We can save files directly on the device's internal storage. Files saved to the internal storage are private in default.

*External Storage* Android devices support a shared external storage to save files. The external storage can be a removable storage media, such as an SD card, or internal storage.

*SQLite Databases* We can use `SQLite` in Android to create databases that will be accessible by name to any class in the app.

*Network Connection* We can use the network to store and retrieve data in our own services.

Before we play a video using `VideoView`, we need to set a path to locate the video, and this path must be inside the AVD itself.

First of all, run an AVD, and jump into Dalvik Debug Monitor Service (DDMS) after the AVD runs. In Android Studio, select Tools, then

Android, then click Android Device Monitor (Tools → Android → Android Device Monitor), as shown in Fig. 4.26. The Android Device Monitor will be similar to Fig. 4.27.

Then select the AVD we just run, and then in the “File Explorer” tab, we can see many folders and files listed. Find the “data” folder and click “Push a file onto the device” on the right top of the interface, as shown in Fig. 4.28. Then select and push a local video file onto the device.

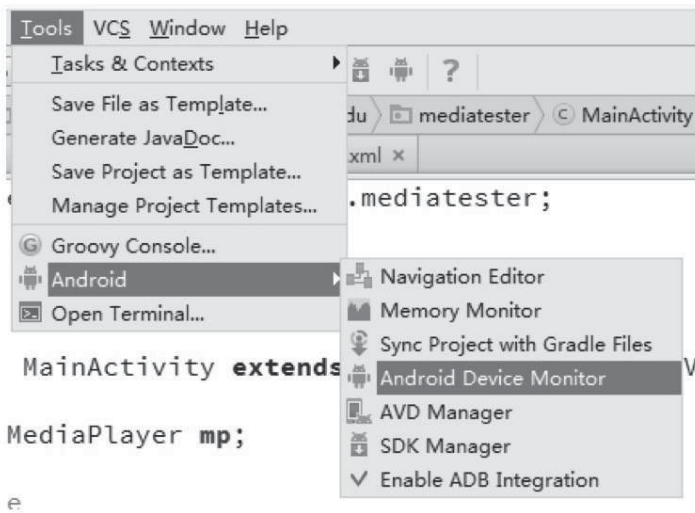


Figure 4.26 Android device monitor.

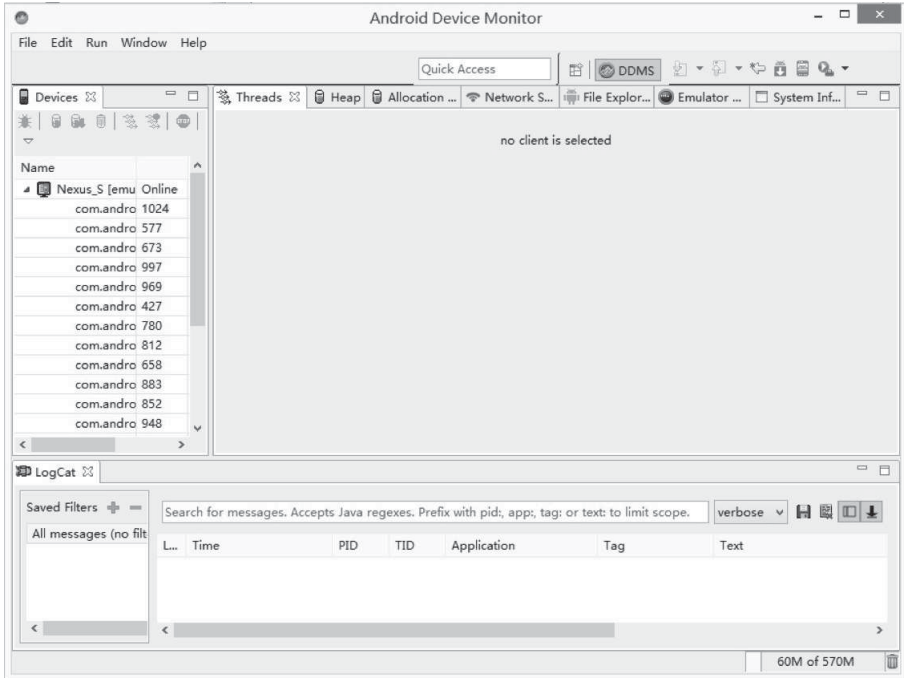


Figure 4.27 Android device Monitor.

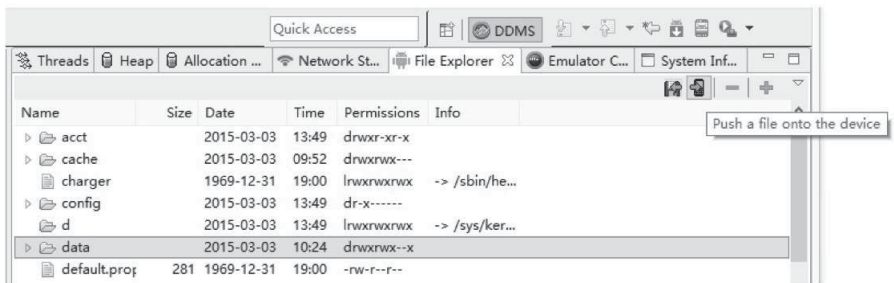


Figure 4.28 Push a file onto the device.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //audio part
    findViewById(R.id.button_start).setOnClickListener(this);
    findViewById(R.id.button_pause).setOnClickListener(this);

    //video part
    //Before we play a video, we need to push the video resource
    //into AVD
    VideoView video = (VideoView) findViewById(R.id.video);
    video.setVideoPath("/data/samplevideo.3gp");
    video.start();
}
```

Figure 4.29 Setting video path and start a video.

The DDMS is used to operate the AVD, not the Android app. If we have pushed some video into a device before, we do not need to push it again in Android Project. Then add two methods to the onCreate() method to set the Video path and play it as follows:

```
video.setVideoPath("/data/samplevideo.3gp");
video.start();
```

Then the current onCreate() method can refer to Fig. 4.29.

In the end, the running result is shown in Fig. 4.30.

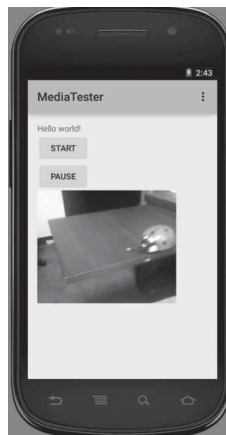


Figure 4.30 The running result of the MediaTester.

# Chapter Five

## Mobile Embedded System Architecture

---

### CONTENTS

---

5.1	Embedded Systems .....
5.1.1	Embedded Systems Overview .....
5.2	Scheduling Algorithms .....
5.2.1	Basic Concepts .....
5.2.2	First-Come, First-Served Scheduling Algorithm ...

5.2.3	Shorted-Job-First Scheduling Algorithm .....
5.2.4	Multiprocessors .....
5.2.5	Priority Scheduling Algorithm .....
5.2.6	ASAP and ALAP Scheduling Algorithm .....
5.2.6.1	ASAP .....
5.2.6.2	ALAP .....
5.3	Memory Technology .....
5.4	Mobile Embedded Systems .....
5.4.1	Embedded Systems in Mobile Devices .....
5.4.2	Embedded Systems in Android .....
5.4.3	Power Management of Android .....
5.4.4	Embedded Systems in Mobile Apps .....
5.5	Messaging and Communication Mechanisms .....
5.5.1	Message Mechanisms .....
5.5.2	Communication Mechanisms .....

---

Mobile device is the indispensable part of a mobile system, and all the chips used in a mobile device are embedded systems. These embedded systems with various functions are controlled by the mobile operating system and collaborate with each other to complete every task mobile apps request.

In this chapter, we introduce the mobile embedded system architecture, including:

Overview of embedded systems

Applications of embedded system.

The processor technology in embedded systems

Basic concepts in processor technology in embedded systems

The scheduling algorithms in processor technology in embedded

systems

Memory technology in embedded systems

Embedded systems in mobile devices

Embedded systems in Android

## **5.1 EMBEDDED SYSTEMS**

---

### **5.1.1 Embedded Systems Overview**

Embedded systems are anything that uses a microprocessor but is not a general-purpose computer. An embedded system is a computer system with a dedicated function, which is embedded as a part of a complete device, including hardware and mechanical parts. These tiny systems can be found everywhere, ranging from commercial electronics, such as cell phones, cameras, portable health monitoring systems, automobile controllers, robots, and smart security devices, to critical infrastructure, such as telecommunication networks, electrical power grids, financial institutions, and nuclear plants. The increasingly complicated embedded systems require extensive design automation and optimization tools.

Modern embedded systems are often based on microcontrollers, such as Central Processing Units (CPU) with integrated memory or

peripheral interfaces, but ordinary microprocessors, which use external chips for memory, and peripheral interface circuits are still common. Embedded systems are commonly used in telecommunication systems, consumer electronics, transportation systems, and medical equipment.

### **Telecommunication Systems**

Telecommunication systems employ numerous embedded systems, from telephone switches to cell phones.

### **Consumer Electronics**

Consumer electronics include personal digital assistants (PDAs), such as audio players, mobile phones, videogame consoles, digital cameras, video players, and printers. Embedded systems are used to provide flexibility, efficiency, and features.

### **Home Automation**

Embedded devices are used for sensing and controlling in-home automation using wired and wireless networks. Embedded devices can be used to control lights, climate, security, audio/visual, and surveillance.

### **Transportation Systems**

Embedded systems are increasingly used from flight to automo-



biles in transportation systems. New airplanes contain advanced avionics, such as Inertial Guidance Systems (IGS) and Global Positioning System (GPS) receivers that also have considerable safety requirements. Various electric motors use electric motor controllers. Automobiles, electric vehicles, and hybrid vehicles increasingly use embedded systems to maximize efficiency and reduce pollution.

### **Medical Equipment**

Medical equipment uses embedded systems for vital signs monitoring, electronic stethoscopes for amplifying sounds, and various medical imaging for non invasive internal inspections. Embedded systems within medical equipment are often powered by industrial computers.

Besides the usages mentioned above, embedded systems are also widely used in a new kind of technology, which is *wireless sensor networking* (WSN). The WSN consists of spatially distributed autonomous sensors to monitor physical or environmental conditions. Commonly monitored parameters are temperature, humidity, pressure, wind direction and speed, illumination intensity, vibration intensity, sound intensity, power-line voltage, chemical concentrations, pollutant

levels, and vital body functions. WSN enables people and companies to measure myriad things in the physical world and acts on this information under the help of embedded Wi-Fi systems. Furthermore, the network wireless sensors, using optimization technologies of embedded systems, are completely self-contained and will typically run off a battery source for years before the batteries need to be changed or charged.

Embedded systems also can be defined as computers purchased as part of some other piece of equipment. They always have a dedicated software in them, and the software may be customizable to users. There are often no “keyboard” and limited display or no general purpose display in an embedded system.

Embedded systems are important for three kinds of reasons:

Engineering reasons. Any device that needs to be controlled can be controlled by a microprocessor. In many situations, it is impossible or unnecessary for the devices to being a complete computer. McDonald’s POS (Point of Sale) terminal is only in charge of recording purchases, calculating and showing price, collecting money, giving change, and printing receipts. This kind of functions is simple, so the POS terminals have little calculating resources. It is unnecessary for the POS terminal to be complete as a general computer, because it is really a

waste.

**Market reasons.** The general-purpose computer market worths billions of dollars; meanwhile the embedded systems market is also worths billions of dollars. Although the price of an embedded system may be much lower than that of a general-purpose computer, the amount of embedded systems are much larger than that of general-purpose computers. In 2009, about 200 embedded systems were used in every new car. There are more than 80 million personal computers were sold every year. While over 3 billion embedded CPUs were sold annually. Furthermore, the personal computer market is mostly saturated, but the embedded market is still growing.

**Pedagogical reasons.** Embedded system designers often need to know hardware, software, and some combination of networking, control theory, and signal processing. This makes the teaching methods of designing embedded systems different from that for designing general-purpose systems.

In this section, we introduce the overview of embedded systems, analyze their usages, explain their importance, list some real applications of embedded systems, and give a high-level of the design of embedded system. We introduce deeper knowledge after introducing the design of embedded system. The first and the most important thing is schedul-

ing.

## 5.2 SCHEDULING ALGORITHMS

---

### 5.2.1 Basic Concepts

First of all, some basic concepts must be introduced and explained. Scheduling is central to operating system design. The success of CPU scheduling depends on two executions. The first one is the process execution consisting of a cycle of CPU execution and Input/ Output (I/O) wait. The second one is the process execution, which begins with a CPU processing, followed by I/O processing, then followed by another CPU processing, then another I/O processing, and so on. The CPU I/O Processing Cycle is the basic concept of processor technology. The processing time is the actual time that is required to complete some job.

The CPU scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. CPU scheduling decisions take place when a process is switching from running to waiting state; switching from running to ready state; switching from waiting to ready and terminating.

Beside the CPU scheduler, dispatcher is also a basic and important

concept in processor technology. The dispatcher module gives control of the CPU to the process selected by the short-term scheduler, and this involves: switching context, switching to user mode, and jumping to the proper location in the user program to restart that program. Most dispatchers have dispatch latency, which is the time they take for the dispatcher to stop one process and start another running.

Then we discuss some criteria of scheduling.

**CPU Utilization.** The CPU utilization refers to a computer's usage of processing resources, or the amount of work handled by a CPU, and it is used to gauge system performance. Actual CPU utilization varies depending on the amount and type of managed computing tasks. The first aim of processor technology is increasing the CPU utilization by keeping the CPU as busy as possible.

**Throughput.** The throughput means the amount of processes that complete their execution per time unit.

**Turnaround Time.** The turnaround time means the amount of time to execute a particular process, and it can be calculated as the sum of the time waiting to get into memory, waiting in the ready queue, and executing on the CPU and the I/O.

**Waiting Time.** The waiting time means the amount of time a process has been waiting in the ready queue.

**Response Time.** The response time means the amount of time it takes from when a request was submitted until the first response is produced.

**Completion Time.** The completion time of one job means the amount of time needed to complete it, if it is never preempted, interrupted, or terminated.

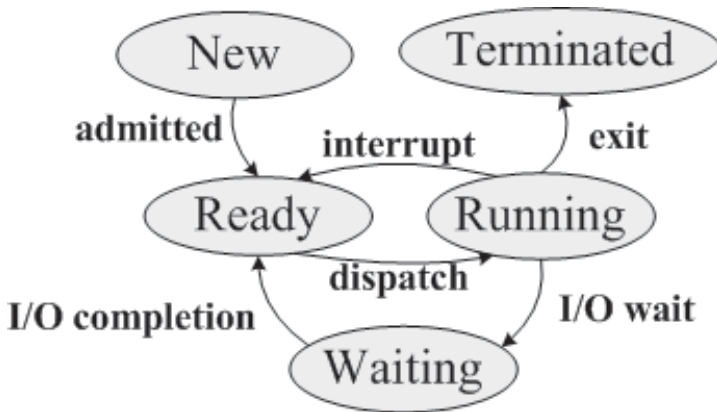


Figure 5.1 The diagram of the process states.

As shown in Fig. 5.1, processes have five types of states. At the *new* state, the process is in the stage of being created. At the *ready* state, the process has all the resources available that it needs to run, but the CPU is not currently working on this process’s instructions. At the *running* state, the CPU is working on this process’s instructions. At the *waiting* state, the process cannot run at the moment, because it is waiting for some resource to become available or for some event to

occur. At the *terminate* state, the process was completed.

### 5.2.2 First Come, First Served Scheduling Algorithm

An important measurable indicator of processor is the average completion time of jobs. Fig. 5.2 represents an example of the schedule for  $k$  jobs. As shown in the figure, there are  $k$  jobs, marked as  $j_k$ , to be completed in the processor. The first job  $j_1$  requires  $t_1$  time units so that the job  $j_1$  can be finished by time  $t_1$ . The second job  $j_2$  starts after the first job  $j_1$  is finished, and the required length of time is  $t_2$ . Therefore, the second job  $j_2$  can be accomplished by the time  $t_1 + t_2$ . Repeat this procedure until the last job  $j_k$  is done.

The total completion time:

$$\begin{aligned}
 A &= t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) + \dots + (t_1 + t_2 + t_3 + \dots + t_k) \\
 &= k * t_1 + (k - 1) * t_2 + (k - 2) * t_3 + \dots + t_k
 \end{aligned}$$

(5.1)



Figure 5.2 A schedule for k jobs.

One of the simplest scheduling algorithm is First Come, First Served (FCFS). The FCFS policy is widely used in daily life. For example, it is the standard policy for the processing of most queues, in which people wait for a service that was not prearranged or preplanned. In the processor technology field, it means the *jobs* are handled in the orders.

For instance, there are four jobs,  $j_1, j_2, j_3$ , and  $j_4$ , with different processing times, which are 7, 4, 3, and 6 respectively. These jobs arrive in the order:  $j_1, j_2, j_3, j_4$ . In FCFS policy, they are handled by the order of  $j_1, j_2, j_3, j_4$ , as shown in Fig. 5.3. The waiting time for  $j_1$  is 0, for  $j_2$  is 7, for  $j_3$  is 11, and for  $j_4$  is 14. The average waiting time is  $(0+7+11+14)/4 = 8$ . The average completion time is  $[7 + (7+4) + (7+4+3) + (7+4+3+6)] / 4 = 13$ .

Suppose that the jobs arrive in the order  $j_2, j_3, j_4, j_1$ ; the result produced by using FCFS is shown in Fig. 5.4. The waiting time for  $j_1$  is 13, for  $j_2$  is 0, for  $j_3$  is 4, and for  $j_4$  is 7. The average waiting time is  $(13+0+4+7)/4 = 6$ . The average completion time is  $[4 +$



$(4+3) + (4+3+6) + (4+3+6+7)] / 4 = 11$ . Both the average waiting time and the average completion time of this scheduling is less than the previous one.

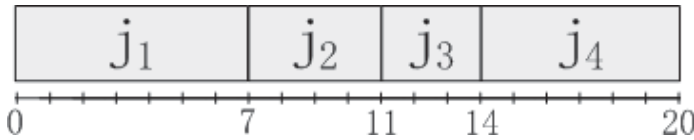


Figure 5.3 An example of FCFS scheduling.

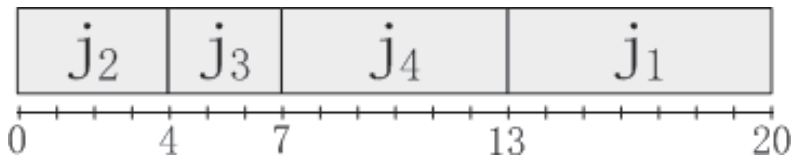


Figure 5.4 Another FCFS result if changing arrival sequence.

### 5.2.3 Shorted Job First Scheduling Algorithm

Then we will introduce another scheduling policy, which is Shortest Job First (SJF). SJF is a scheduling policy that selects the waiting process with the smallest execution time to execute first. SJF is advantageous because of its simplicity, and it minimizes the average completion time. Each process has to wait until its execution is complete.

Using the example mentioned in Section 2.2, while ignoring their arrival time, we first sort these jobs by their processing time, as  $j_3, j_2, j_4, j_1$ . The SJF scheduling result is shown in Fig. 5.5. The

waiting time for  $j_1$  is 13,  $j_2$  is 3,  $j_3$  is 0, and  $j_4$  is 7. The average waiting time is  $(13+3+0+7) = 5.75$ . The completion time for  $j_1$  is  $(13+7)$ ,  $j_2$  is  $(3+4)$ ,  $j_3$  is  $(0+3)$ ,  $j_4$  is  $(7+6)$ . The average completion time is  $(20+7+3+13)/4 = 10.75$ . This scheduling has lower average waiting time and average completion time than the previous two schedules.

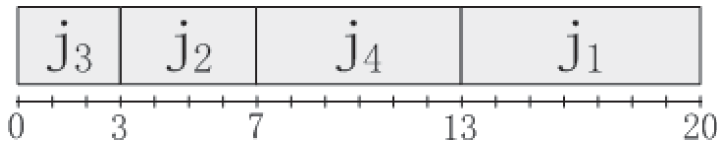


Figure 5.5 An example of SJF scheduling.

*Theorem: SJF scheduling has the lowest total completion time with a single processor.*

**Proof by contradiction:** Assuming that there are a series of jobs that were sorted by their completion time from short to long, as  $j_1, j_2, j_3, \dots, j_i, j_{i+1}, \dots, j_k$ , which also means the completion time of them can be ordered as  $t_1 < t_2 < t_3 < \dots < t_i < t_{i+1} < \dots < t_k$ . Using the SJF scheduling algorithm, the result is exactly the same as the order  $j_1, j_2, j_3, \dots, j_i, j_{i+1}, \dots, j_k$ . Then we suppose that there is another order A that has lower total completion time than the one produced by SJF,  $j_1, j_2, j_3, \dots, j_{i+1}, j_i, \dots, j_k$ . Based on Equation 5.1, the total completions

time is  $T = k * t_1 + (k - 1) * t_2 + (k - 2) * t_3 + \dots + (k - i + 1) * t_i + (k - i) * t_{i+1} + \dots + t_k$ . So, we can get the total completion time of both orders. The SJF one is  $T_s = k * t_1 + (k - 1) * t_2 + (k - 2) * t_3 + \dots + (k - i + 1) * t_i + (k - i) * t_{i+1} + \dots + t_k$ . The A one is  $T_a = k * t_1 + (k - 1) * t_2 + (k - 2) * t_3 + \dots + (k - i + 1) * t_{i+1} + (k - i) * t_i + \dots + t_k$ . From the supposing condition,  $T_s < T_a$ .

$$T_s > T_a;$$

$$k * t_1 + (k - 1) * t_2 + (k - 2) * t_3 + \dots + (k - i + 1) * t_i + (k - i) * t_{i+1} + \dots + t_k > k * t_1 + (k - 1) * t_2 + (k - 2) * t_3 + \dots + (k - i + 1) * t_{i+1} + (k - i) * t_i + \dots + t_k; (k - i + 1) * t_i + (k - i) * t_{i+1} > (k - i + 1) * t_{i+1} + (k - i) * t_i; t_i > t_{i+1}.$$

However,  $t_i > t_{i+1}$  is contradictory to  $t_i < t_{i+1}$ , in the assuming condition. As a result, A does not exist, which means there is no solution that has lower total completion time than the SJF scheduling. In the end, we can conclude that SJF scheduling has the lowest average waiting time with a single processor. However, is SJF still optimal with multiple processors?

### 5.2.4 Multiprocessors

After discussing the single processor, we will expand the topic into multiprocessors. There are nine jobs with different completion times

in three processors, as shown in Fig. 5.6, and we first give an optimal schedule using SJF. The average completion time is  $\{(3+5+6) + [(6+10)+(5+11)+(3+14)] + [(3+14+15)+(5+11+18)+(6+10+20)]\} / 9 = 18.33$ . There is another optimal schedule, as shown in Fig. 5.7.

The average completion time is  $\{(3+5+6) + [(5+10)+(3+11)+(6+14)] + [(5+10+15)+(6+14+18)+(3+11+20)]\} / 9 = 18.33$ .

In multiprocessors, there are three theorems:

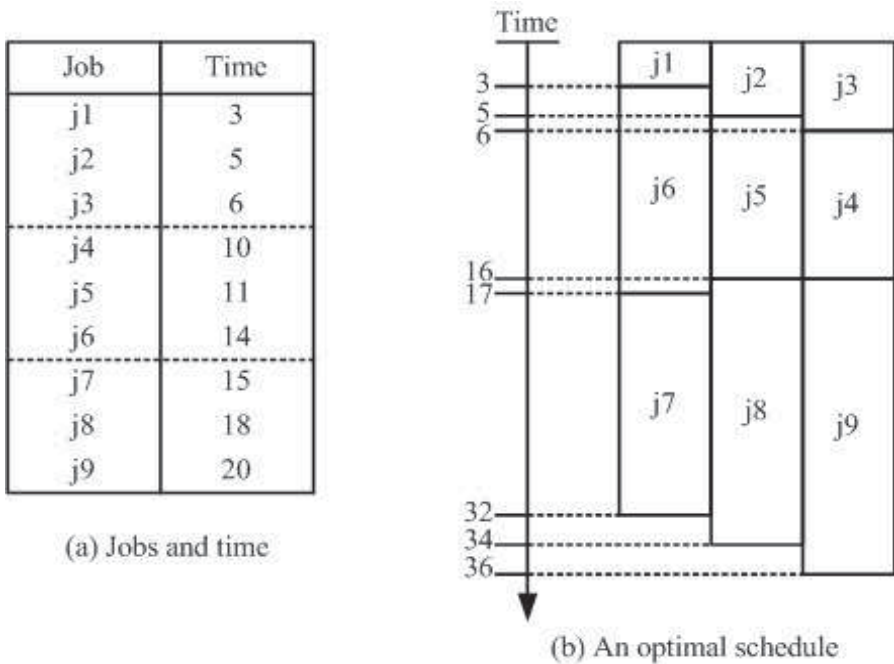


Figure 5.6 An SJF schedule to complete nine jobs in three processors.

Theorem 5.1 *SJF scheduling has the optimal average waiting time and completion time in the multiprocessor.*

Theorem 5.2 *With the same average waiting time, there is more than one schedule with various final completion time.*

Theorem 5.3 *The algorithm to find the optimal final completion time is NP-Hard.*

Assuming that the processing time of  $j_1$  to  $j_k$  is  $t_1$  to  $t_k$ , respectively, the average completion time in three processors calculates as Equation 5.2: The average completion time is

$$\begin{aligned} & \{(t_1 + t_2 + t_3) + (t_1 + t_2 + t_3 + t_4 + t_5 + t_6) + \dots + (t_1 + t_2 + \dots + t_k)\} / 3k \\ & = \{k(t_1 + t_2 + t_3) + (k - 1)(t_4 + t_5 + t_6) + \dots + (t_{k-2} + t_{k-1} + t_k)\} / 3k. \end{aligned}$$

(5.2)

Then we assign that  $T1 = t1 + t2 + t3, T2 = t4 + t5 + t6,$

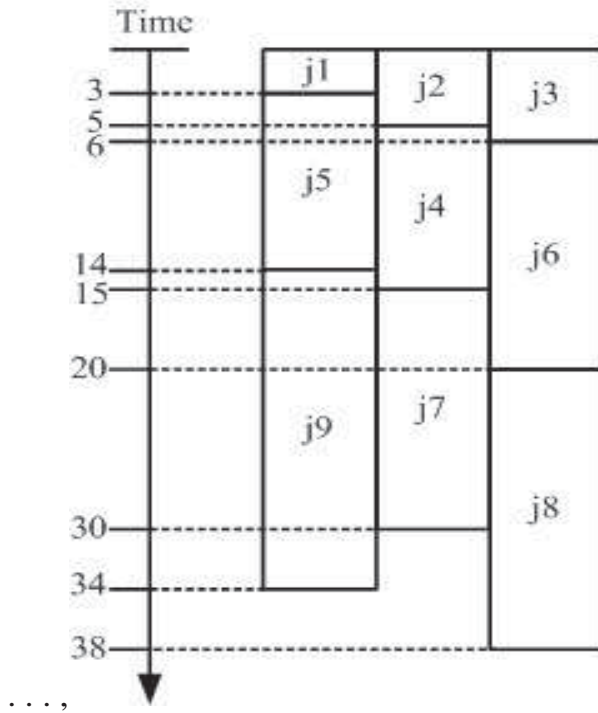


Figure 5.7 Another schedule to complete nine jobs in three processors.

$T_k = t_{3k-2} + t_{3k-1} + t_{3k}$ . The total completion time in three processors can be formulated as  $kT1 + (k-1)T2 + \dots + T_k$ . At last, we can formulate this problem into the one in a single processor. In the end, we can use

the same method as the one in Section 2.3 to prove that the SJF schedule has the optimal average completion time in multiprocessors. From Equation 5.2, we can see that the detailed sequence of  $j1, j2, j3$  does not impact the average waiting time of the whole schedule. As a result, the two schedules in Fig. 5.6 and Fig. 5.7 have the same average

waiting time. However the time when the last job is completed these two schedules are different, which are 36 and 38. If there is a time constraint that is less than 38, the second schedule is not suitable, while the first schedule can be chosen. Furthermore, there are many other schedules having the same average waiting time with these two schedules, because changing the sequence of  $j_{3i+1}$ ,  $j_{3i+2}$ ,  $j_{3i}$  does not change the average waiting time. Nevertheless, the time when the last job is completed is various, and how to find the optimal schedule that has the least time when the last job is completed is too hard to be solved by normal algorithms. This problem is a typical NP-Hard problem, and we will discuss this problem and how to solve it in later chapters.

### 5.2.5 Priority Scheduling Algorithm

The next scheduling algorithm is Priority Scheduling algorithm. In priority scheduling, a priority number, which can be an integer, is associated with each process. The CPU is allocated to the job with the highest priority, and the smallest integer represents the highest priority. The priority scheduling can be used in the preemptive and nonpreemptive schemes. The SJF scheduling is a priority scheduling, where priority is the predicted next CPU processing time. The following is a given example about the implementation of the priority scheduling in

preemptive schemes, as shown in Fig. 5.8. The priority of each job is inverse with its processing time. As a result, the result using the priority scheduling algorithm is the same as the result from SJF scheduling.

The priority scheduling has the potential restrictions deriving from process starvations. The *Process Starvation* is the processes that require a long completion time, while processes requiring shorter completion times are continuously added. A scheme of “Aging” is used to solve this problem. As time progresses, the priority of the process increases. Another disadvantage is that the total execution time of a job must be known before the execution. While it is not possible to exactly predict the execution time, a few methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.

At last, we will introduce the Round Robin (RR) scheduling. In RR scheduling, each job gets a small unit of CPU time, called *time quantum*, usually 10 - 100 milliseconds. After this time has elapsed, the job is preempted and added to the end of the ready queue. If there are  $n$  jobs in the ready queue and the *time quantum* is  $q$ , then each job gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No job waits more than  $(n - 1)$  time units. If the  $q$  is large, the RR scheduling will be the FCFS scheduling. Nevertheless, if the  $q$  is small, the overhead



may be too high because of the too-often context switch.

Jobs	BurstTime	Priority	Arrival Time
j <sub>1</sub>	7	4	0.0
j <sub>2</sub>	4	2	2.0
j <sub>3</sub>	3	1	4.0
j <sub>4</sub>	6	3	5.0

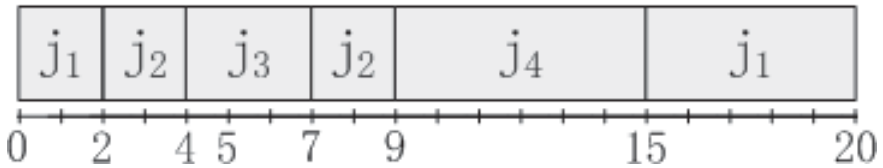


Figure 5.8 An example of the priority scheduling.

Actually, there are two kinds of scheduling schemes that are non-preemptive and preemptive.

***Nonpreemptive.***

The nonpreemptive scheduling means that once the CPU has been allocated to a process, the process keeps the CPU resource until it releases the CPU either by terminating or switching to a waiting state.

***Preemptive.***

In the preemptive schemes, a new job can preempt CPU re-

sources, if its CPU processing length is less than the remaining time of the current executing job. This scheme is known as the Shortest-Remaining-Time-First (SRTF).

In computer science, preemption is the act of temporarily interrupting a job being carried out by a computer. It is normally carried out by a privileged job on the system that allows interruptions. Fig. 5.5 shows SJF scheduling in the situation when all the jobs arrive at the same time, but situation will be complicated when considering their different arrival times, especially in preemptive scheme.

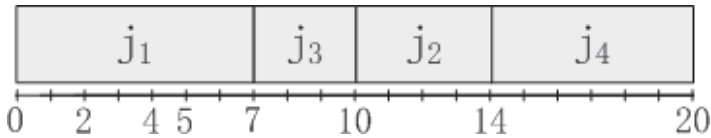


Figure 5.9 An example of the nonpreemptive SJF solution.

Still taking the example mentioned in Section 5.2.2, add arrival times to them,  $j_1$  arriving at time 0.0;  $j_2$  arriving at time 2.0;  $j_3$  arriving at time 4.0;  $j_4$  arriving at time 5.0. The SJF scheduling in a nonpreemptive scheme is shown in Fig. 5.9. At time 0,  $j_1$  arrives, and there are no other jobs competing with it, so  $j_1$  is in the running list. At time 2, 4, and 5,  $j_2$ ,  $j_3$ , and  $j_4$  arrive, respectively.

However, they cannot interrupt  $j_1$  and grab the resource  $j_1$  is using, so they are all in the waiting list. At time 7,  $j_1$  is finished, and now there are three jobs in the waiting list. Among these three jobs,  $j_3$  needs the shortest processing time, so it gets the resource and turns into the running list. At time 10,  $j_3$  is finished, and now there are two jobs in the waiting list, which are  $j_2$  and  $j_4$ . Since  $j_2$  needs a shorter processing time than  $j_4$  does,  $j_2$  gets the resource and turns into the running list. At time 14,  $j_2$  is finished, and now there is only one job in the waiting list, which is  $j_4$ . So  $j_4$  gets the resource and turns into the running list. Finally,  $j_4$  is finished at time 20. In this scheduling, the waiting time for  $j_1$  is 0,  $j_2$  is (10-2),  $j_3$  is (7-4), and  $j_4$  is (14-5). The average waiting time is  $(0+8+3+9)/4 = 5$ . The completion time for  $j_1$  is 7,  $j_2$  is (14-2),  $j_3$  is (10-4), and  $j_4$  is (20-5). The average completion time is  $(7+12+6+15)/4 = 10$ .

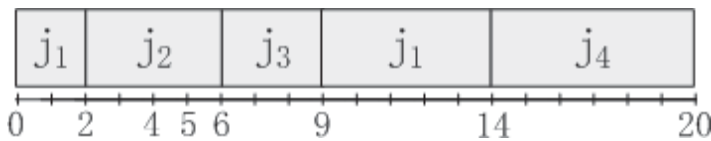


Figure 5.10 Example of the preemptive SJF solution.

The SJF scheduling in a preemptive scheme is shown in Fig. 5.10. At time 0,  $j_1$  arrives, and there are no other jobs competing with it, so  $j_1$  is in the running list. At time 2,  $j_2$  arrives, and  $j_2$  has

shorter processing time than  $j_1$ , so it preempts  $j_1$ .  $j_1$  goes to the waiting list, while  $j_2$  is in the running list. At time 4,  $j_3$  arrives.  $j_3$  needs 3 time to be completed, while  $j_2$  needs 2 time. So  $j_3$  cannot preempt  $j_2$  and stays in the waiting list. At current stage,  $j_1$  and  $j_3$  are both in the waiting list.

Next, at time 5,  $j_4$  arrives, but it has longer processing time than  $j_2$ , so it cannot preempt  $j_2$ .  $j_4$  joins in the waiting list. At time 6,  $j_2$  is finished, and now there are three jobs in the waiting list. Among them,  $j_3$  needs the shortest processing time, so  $j_3$  get the resource, while others are still waiting. At time 9,  $j_3$  is finished, and now there are two jobs in the waiting list. Since  $j_1$  needs a shorter processing time, which is 5, than  $j_4$  does, which is 6.  $j_1$  gets the resource and turns into the running list. At time 14,  $j_1$  is finished, and now there is only one job in the waiting list, which is  $j_4$ . As a result,  $j_4$  get the resource and is finally finished at time 20. In this scheduling, the waiting time for  $j_1$  is  $9-2$ ,  $j_2$  is  $(0)$ ,  $j_3$  is  $(6-4)$ , and  $j_4$  is  $(14-5)$ . The average waiting time is  $(7+0+2+9)/4 = 4.5$ . The completion time for  $j_1$  is 14,  $j_2$  is  $(6-2)$ ,  $j_3$  is  $(9-6)$ , and  $j_4$  is  $(20-5)$ . The average completion time is  $(14+4+3+15)/4 = 9$ .

### 5.2.6 ASAP and ALAP Scheduling Algorithm

First, we will introduce the Directed Acyclic Graphs (DAG) to model the scheduling problem about the delay in processors. A DAG is a directed graph with no directed cycles. It is formed by a collection of vertices and directed edges, each edge connecting one vertex to another. There is no way to start at some vertex and follow a sequence of edges that eventually loop back to this vertex. We create a DAG with a source node and a sink node, as shown in Fig. 5.11. The source node is  $v_0$ , and the sink node is  $v_n$ . The solid lines refer to the execution delay between nodes. Broken lines mean there is no execution delay between nodes. For example, neither source node nor sink node has the execution time.

Moreover, students need to understand two concepts before introducing the algorithm, including *Predecessor* and *Successor*. A *Predecessor* refers to the node that needs to be finished before the current node. For example, in Fig. 5.11,  $v_2$  and  $v_3$  are the predecessors of  $v_5$ .

A *Successor* refers to the node that succeeds the current node. In Fig. 5.11,  $v_4$  is  $v_1$ 's successor.

As exhibited in Fig. 5.11, we define  $V = \{v_0, v_1, \dots, v_n\}$  in which  $v_0$  and  $v_n$  are pseudo nodes denoting the source node and sink node, respectively.  $D = \{d_0, d_1, \dots, d_n\}$  where  $d_i$  denotes the execution

delay of  $v_i$ ;

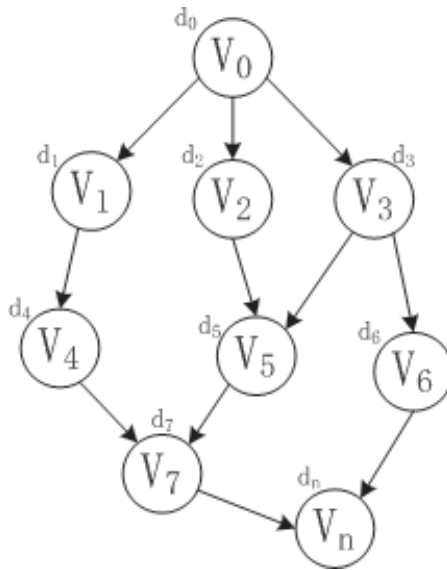


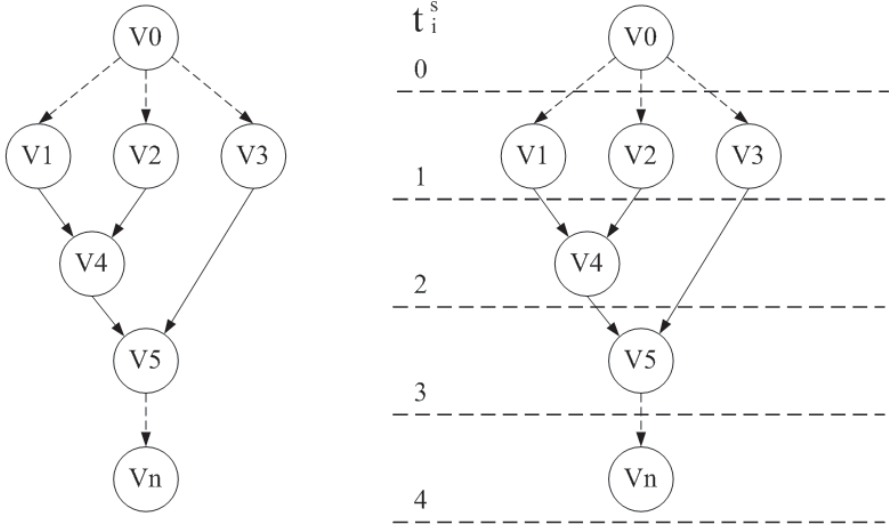
Figure 5.11 A sample of the directed acyclic graph.

Then we use a topological sorting algorithm to produce a legal sequence, which is scheduling for uniprocessor. A topological sorting of a directed acyclic graph is a linear ordering of its vertices, such that for every directed edge  $\{u, v\}$  from vertex  $u$  to vertex  $v$ ,  $u$  comes before  $v$  in the ordering. First, finding a list of nodes whose indegree = 0, which means they have no incoming edges, inserting them into a set  $S$ , and removing them from  $V$ . Then starting the loop that keeps removing the nodes without incoming edges until  $V$  is empty. The output is the result of topological sorting and the scheduling for the uniprocessor. Referring to Fig. 5.11, we can get three results:  $\{v_0, v_1, v_4, v_7, v_n\}$ ,

$\{ v_0, v_2, v_5, v_7, v_n \}$ , and  $\{ v_0, v_3, v_6, v_n \}$ .

To eliminate the latency, we assign values to  $d_i$  and simplify the problem. We set  $d_1, d_2, d_3, d_4$ , and  $d_5$  as 1. We use two scheduling algorithms, which are As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) Scheduling Algorithms.

5.2.6.1 ASAP



(a) A DAG  $G(V, E, D)$  with  $d_1=d_2=d_3=d_4=d_5=1, d_0=d_n=0$

(b) The schedule generated by ASAP

Figure 5.12 A simple ASAP for minimum latency scheduling.

As shown in Fig. 5.12, first, set  $t^s = 1$ , and  $v_0$  has no predecessors, and  $d_0$  is 0. Thus,  $v_0$  has the same latency as its successors,  $v_1, v_2$ , and  $v_3$ . In this step,  $v_0$  is scheduled. Then because  $v_1$ 's predecessor  $v_0$  is scheduled, it can be selected at the 1 latency

time. The same operations can be implemented with  $v_2$  and  $v_3$  at the first latency time unit. In this step,  $v_1$ ,  $v_2$ , and  $v_3$  are scheduled. Then  $v_4$  can be selected at the 2 latency, because its predecessors,  $v_1$  and  $v_2$ , are scheduled. However,  $v_5$  cannot be selected at the 2 latency, because one of its predecessors,  $v_4$ , is not scheduled before the 2 latency. Then after  $v_4$  is scheduled,  $v_5$  can be selected at the 3 latency, because its predecessors,  $v_3$  and  $v_4$ , are scheduled. At last,  $v_n$  is selected at 4 latency, because its predecessor,  $v_5$  is scheduled.

In ASAP for minimum latency scheduling algorithm:

Step 1: schedule  $v_0$  by setting  $t_0^s = 1$ . This step is for launching the calculation of the algorithm.

Step 2: select a node  $v_i$  whose predecessors are all scheduled. This process will be repeated until the sink node  $V_n$  is selected.

Step 3: schedule  $v_i$  by setting  $t^s = \max_{j: v_j \rightarrow v_i \in E} t^s + d_j$ . The equation represents the current node status at the exact timing unit. It represents the latency time at the current node is summing up the maximum latency time of the predecessors' nodes.

$$t^s = \max_{j: v_j \rightarrow v_i \in E} t^s + d_j$$

Step 4: repeat Step 2 until  $v_n$  is scheduled.



5.2.6.2 ALAP

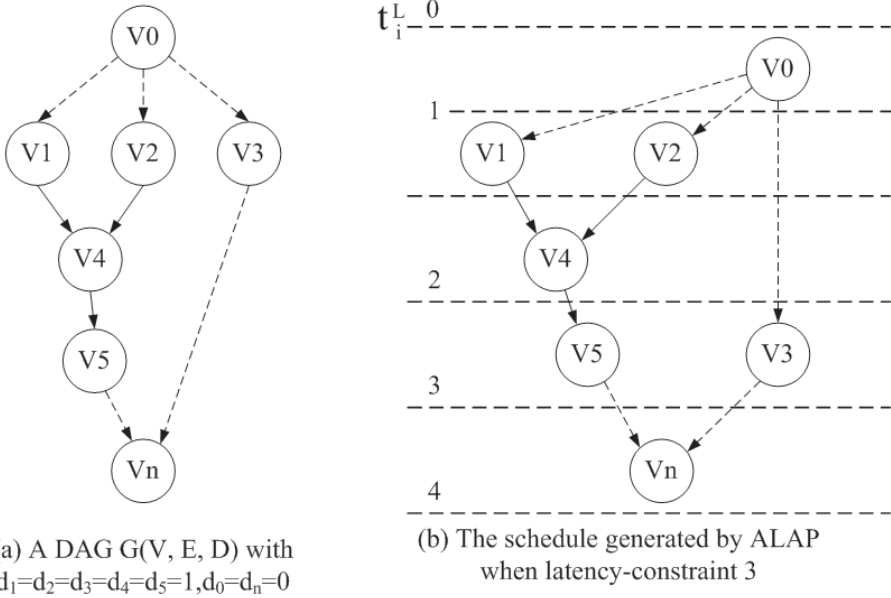


Figure 5.13 ALAP scheduling for latency-constraint scheduling.

As shown in Fig. 5.13, first, schedule the button node  $v_n$  at the time latency  $3+1$ , and set  $t_i^L = 4$ . In this step,  $v_n$  is scheduled. Then  $v_3$  and  $v_5$  can be selected at the 3 time latency, because their successor,  $v_n$ , is scheduled. In this step,  $v_3$  and  $v_5$  are scheduled. Then  $v_4$  can be selected at the 2 time latency, because its successor,  $v_5$ , is scheduled. In this step,  $v_4$  is scheduled. In this time latency, although  $v_0$  is the

predecessor of  $v_3$ , it cannot be selected at the 2 time latency, because  $v_0$ 's other successors,  $v_1$  and  $v_2$ , are not scheduled. Then  $v_1$  and  $v_2$  can be selected at the 1 time latency, because their successor,  $v_4$ , is scheduled. In this step,  $v_1$  and  $v_2$  are scheduled. At last,  $v_0$  can be selected at 1 time latency, because its successor,  $v_1$ ,  $v_2$ , and  $v_3$ , are scheduled, and  $d_0$  is 0.

In ALAP for latency-constraint ( $\lambda$ ) scheduling algorithm:

Step 1: schedule  $v_n$  by setting  $t_n^L = \lambda + 1$ . This step means the first scheduled node is  $v_n$ .

Step 2: select a node  $v_i$  whose successors are all scheduled. It means the selected node must be a node whose successors must be scheduled. This process will be repeated until the source node  $v_0$  is selected.

Step 3: schedule  $v_i$  by setting  $t_i^L = \min_{j: v_j \rightarrow v_i \in E} t_j^L + d_j$ . The equation represents the current node status at the exact timing unit. It represents that the latency time at the current node is subtracting the sum of minimum latency times from the sink node's latency- constraint.

$$t_i^L = \min_{j: v_j \rightarrow v_i \in E} t_j^L + d_j$$

Step 4: repeat Step 2 until  $v_0$  is scheduled. Fig. 5.13 exhibits an

ALAP scheduling for latency-constraint scheduling.

Comparing ASAP and ALAP scheduling as shown in Fig. 5.12 and Fig. 5.13, we can find that  $v_3$  can be completed at several time latencies. It can be completed at 1 time latency as soon as possible, and 3 time latency as late as possible.

In this section, we introduce some basic concepts, such as CPU utilization, waiting time, response time, and completion time. Then we introduce some scheduling algorithms, including *First-Come, First Server*, *Shortest-Job-First*, *priority scheduling*, *Round Robin*, *As-Soon-As-Possible*, and *As-Late-As-Possible*. In the next section, we introduce the processor technology about scheduling algorithm in single processor and multi-processor.

### 5.3 MEMORY TECHNOLOGY

---

Memory is one of the fastest evolving technologies in embedded systems over the recent decade. No matter how fast processors can run, there is one unchanged fact so that every embedded system needs memory to store data. Furthermore, with the rapid development of the processor, more and more data pass back and forth between the processor and the memory. The bandwidth of a memory, which is the speed of the

memory, becomes the major constraint impacting the system's performance.

When building an embedded system, the designers should consider the overall performance of the memory in the system. There are two key metrics for memory performance: *write ability* and *storage permanence*. Writing in memory can be various in different memory technologies. Some kinds of memories, such as *Random-Access Memory* (RAM), require special devices or techniques for writing. A RAM device allows data items to be read and written in roughly the same amount of time regardless of the order in which data items are accessed. The two main forms of modern RAM are *Static RAM* (SRAM) and *Dynamic RAM* (DRAM). In SRAM, a bit of data is stored using the state of a six transistor memory cells. This form of RAM is more expensive to produce, but it is generally faster and requires less power than DRAM and, in modern computers, is often used as cache memory for the CPU. DRAM stores a bit of data using a transistor and capacitor pair, which together comprise a DRAM memory cell. The capacitor holds a high or low charge (1 or 0, respectively), and the transistor acts as a switch that lets the control circuitry on the chip read the capacitor's state of charge or change it. As this form of memory is less expensive to produce than static RAM, it is the predominant

form of computer memory used in modern computers.

At the high end of the memory technology, we can select the memory that the processor can write to simply in a short time. There are some kinds of memories that can be accessed by setting address lines, or data bits, or control lines appropriately. At the middle of the range of memory technology, some slow written memory can be chosen. At the low end are the types of memory that require special equipment for writing.

Besides the write ability, we also need to take storage permanence into consideration. How long the memory can hold the written bits in themselves can have a key impact on the reliability of the system. In the aspect of storage permanence, there are two kinds of memory technologies: *nonvolatile* and *volatile*. The major difference is that the nonvolatile memory can hold the written bits after power is no longer supplied, but volatile cannot. The nonvolatile memory is typically used for the task of secondary storage, or long-term persistent storage. Meanwhile, the most widely used form of primary storage today is volatile memory. When the computer is shut down, anything contained in the volatile memory is lost. The advanced memory technology needs to attach to the operating system. Dynamic programming is an option for heterogeneous memories' optimizations,

which will be discussed later.

## 5.4 MOBILE EMBEDDED SYSTEMS

---

### 5.4.1 Embedded Systems in Mobile Devices

A mobile device is a typical embedded system, which is formed by a group of electronic components, such as mobile processors, storage, memory, graphics, sensors, camera, battery, and other chips. Integrating these electronic parts is to achieve a variety of desired functions for different purposes. In this section, we will use the smartphone to represent an example of a mobile embedded system. A smart phone is one of the most adopted mobile devices in contemporary people's lives. Currently, the hardware structures of most smartphones are two-processor frameworks. The two processors are the application processor and the baseband processor, which are shown in [Fig. 5.14](#). The *Application Processor* is in charge of running a mobile operating system and various kinds of mobile apps. It is the one that controls the whole system. Most functions provided by chips, such as the keyboard, screen, camera, and sensors, are controlled by the application processor.

Meanwhile, the *Baseband Processor* is responsible for wireless communication. This wireless communication is not the cellular or Wi-Fi

network, and it is the telephone network with *Radio Frequency* (RF). The radio frequency is a rate of oscillation, which corresponds to the frequency of radio waves, and the alternating currents that carry radio signals. The radio frequency module is used to send signals to the telephone network. There are two other basic modules in the *base-band processor*, which are the *Digital Baseband* (DBB) and the *Analog Baseband* (ABB). They modulate and demodulate the voice signal and the digital signal, encode and decode the communication channel,

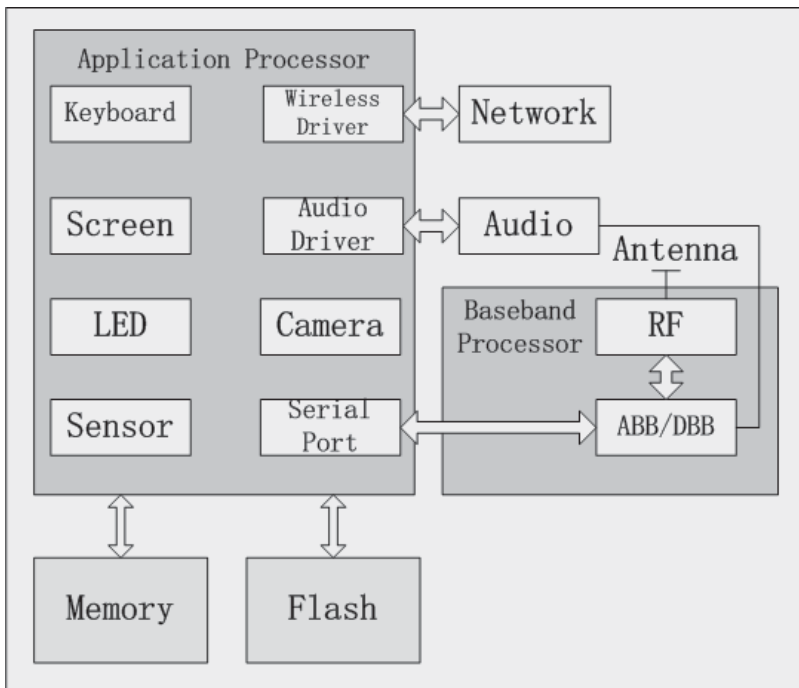


Figure 5.14 Hardware structure of a smartphone.

and control the wireless modem (modulator-demodulator). The application processor communicates with the baseband processor via the

serial port, USB, and others.

### 5.4.2 Embedded Systems in Android

After introducing the hardware structure of the smartphone, we will take Android as an example to explain the Kernel inside Android and show how the Kernel works. As discussed in [Chapter 1](#), Android is based on the Linux Kernel, and the Linux Kernel is an abstract layer between the hardware and the software. The basic functions of Android are provided by the Linux Kernel core system service, such as file management, memory management, process management, network stack, and drivers. The Linux Kernel also provides drivers to support all the hardware related to the mobile embedded system. As shown in [Fig. 5.15](#), there are display driver, keyboard driver, audio driver, power management, Wi-Fi driver, camera driver, and other sensor drivers. We will list some of them and explain what they do.

**Display Driver.** It is based on the framebuffer driver in Linux. The framebuffer offers a mechanism that allows the application to directly control the change of the screen.

**Keyboard Driver.** It is the driver for buttons on the mobile device, such as the Home button, the Menu button, the Return



button, and the Power button.

**Wi-Fi Driver.** It is the driver for Wi-Fi connection based on IEEE 802.11.

**Sensor Driver.** Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device.

For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

Above the Linux Kernel is the hardware abstraction layer, which provides an easy way for applications to discover the hardware on the system. The “abstract” of the hardware abstraction layer does not

mean the real operations of the hardware, and the operations of the hardware are still achieved by drivers. However, the interfaces offered by the hardware abstraction layer make it simple for developers to “use” the hardware.

*Hardware Abstraction Layer Hardware abstraction layer is a software subsystem for UNIX-based operating systems providing hardware abstraction. The purpose of the hardware abstraction layer is to allow application to discover and use the hardware of the host system through a simple, portable, and abstract Application Programming Interface (API), regardless of the type of the underlying hardware.*

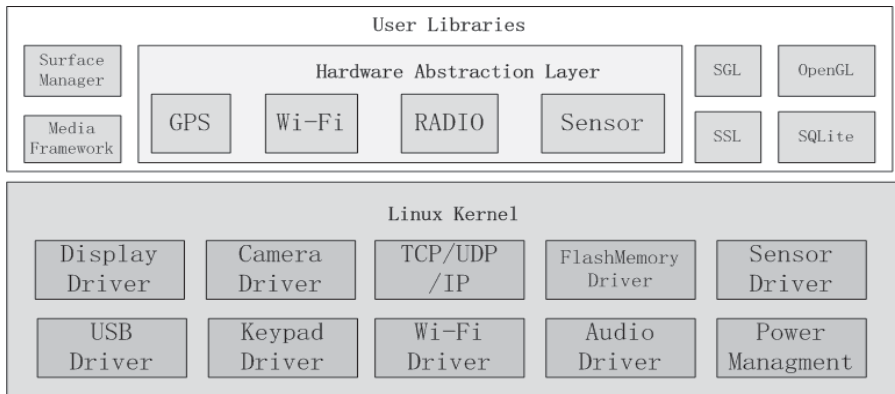


Figure 5.15 Linux Kernel of Android.

### 5.4.3 Power Management of Android

Android supports its own power management (on top of the standard Linux power management) designed with the premise that the CPU should not consume power if no applications or services require power.

As shown in [Fig. 5.16](#), Android requires that applications and services request CPU resources with wake locks through the Android application framework and native Linux libraries. If there are no active wake locks, Android will shut down the CPU. The wake locks are used by applications and services to request CPU resources. The power management uses wake locks and time-out mechanism to switch the state of system power, so that system power consumption decreases.

Currently, Android only supports screen, keyboard, buttons backlight, and the brightness of the screen. As shown in [Fig. 5.17](#), when a user application acquires full wake lock or a screen/keyboard touch activity event occurs, the machine will enter “awake” state. If timeout happens or the power key is pressed, the machine will enter the “notification” state. If partial wake locks are acquired, it will remain in “notification”. If all partial locks are released, the machine will go into “sleep.

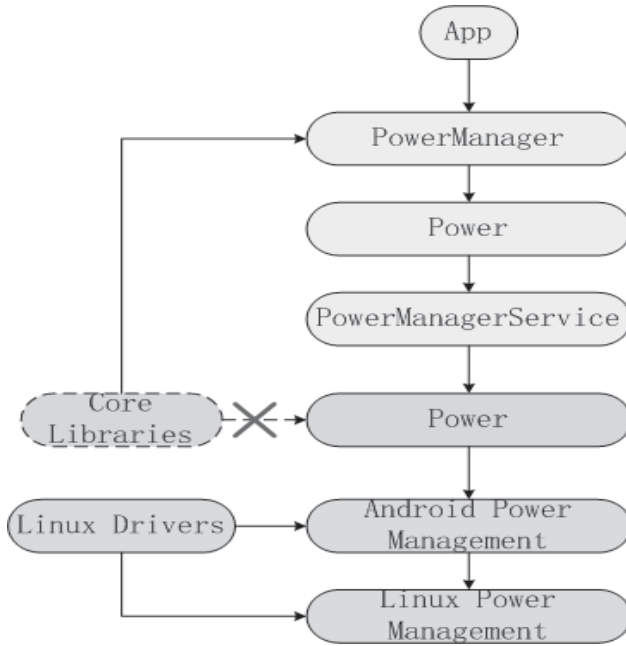


Figure 5.16 Power management of Android.

### 5.4.4 Embedded Systems in Mobile Apps

The mobile embedded systems are under the layer of mobile operating systems. The mobile embedded systems cannot directly used by mobile apps, and they only can be used through mobile operating systems, such as iOS and Android. We will take Android as an example.

Android is already an embedded operating system, and its roots are derived from embedded Linux. The main hardware platform for Android is the Acorn RISC Machine (ARM) architecture. ARM is a family of instruction set architectures for computer processors based on a reduced instruction set computing architecture. An approach that is based on reduced instruction set reduces costs, heat, and power

consumption. Such reductions are desirable traits for light, portable, battery-powered devices, and other embedded systems. Android devices incorporate many optional hardware components, including cameras, GPS, orientation sensors, dedicated gaming controls, accelerometers, gyroscopes, barometers, magnetometers, proximity sensors, thermometers, and touch-screens.

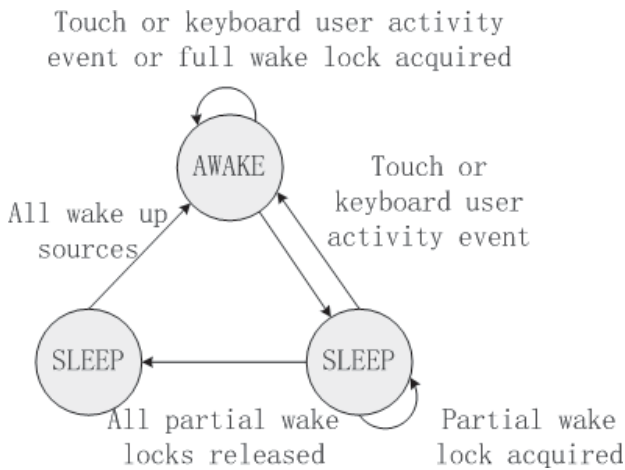


Figure 5.17 A finite-state machine of the Android power management.

We can use Android *Software Development Kit* (SDK) to develop our own mobile apps, and via the methods that are already implemented to use the embedded systems inside a mobile device. For example, developers only can use the camera of a mobile device through calling methods encapsulated in the Android SDK. This design method makes the process of developing mobile apps much simpler than old methods. The developers do not need to spend time designing the in-

teraction with embedded systems inside an Android device, and they only need to know what functions Android SDK can provide. We will introduce more knowledge about Android SDK and developing technologies in the next chapter.

## **5.5 MESSAGING AND COMMUNICATION MECHANISMS**

---

In this section, we will introduce two mechanisms used in Android, including message and communication mechanisms.

### **5.5.1 Message Mechanisms**

Android provides message mechanisms in three core classes: `Looper`, `Handler`, and `Message`. Similar to some other operating system, there is a `Message Queue` in Android. However, this `Message Queue` is packaged in `Looper` class. This class is mainly used to run a message loop for a thread. Threads by default do not have a message loop associated with them. We can call the `prepare()` method in the thread that is to run the loop, and then call `loop()` to process the message queue. The main function of `prepare()` method is defining the `Looper` object as a `ThreadLocal` object. After calling the `loop()` method, the `Looper` thread begins to work, and it continually processes the first message

in the message queue. The working mechanism of the prepare() and loop() method, are shown in Fig. 5.18.

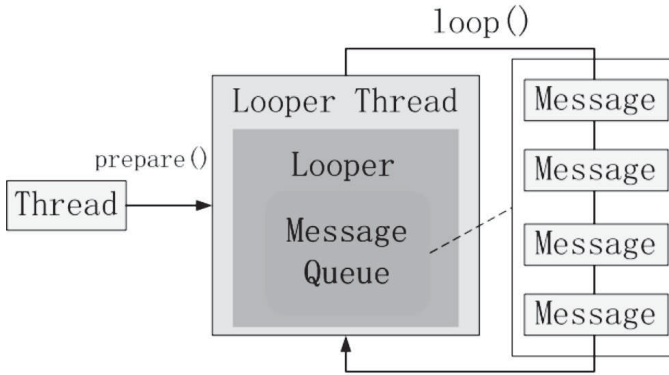
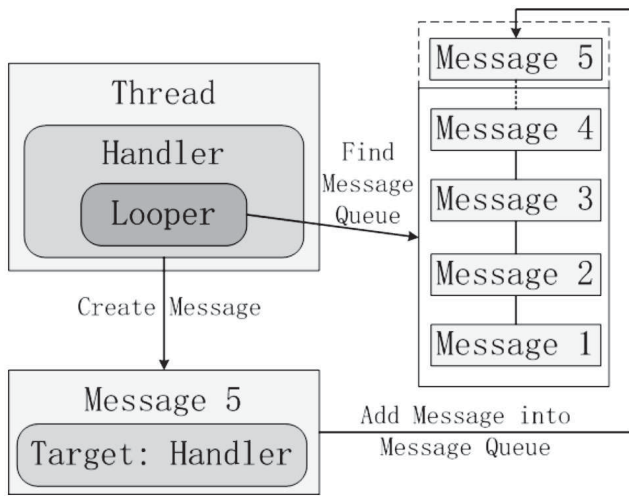


Figure 5.18 prepare() and loop() methods.

Furthermore, we will introduce how to add a message into the message queue. In Android, a Handler allows us to send and process Message and Runnable objects associated with the thread’s Message-Queue. Each Handler instance is associated with a single thread and that thread’s message queue. A Handler has two main functions: (1) to schedule messages and runnable to be executed as some point in the future, and (2) to enqueue an action to be performed on a different thread than our own.

Fig. 5.19 shows the process of adding a message into the message queue using Handler. First, Handler creates a message. Then, find the



**Figure 5.19** Use handler to add message into Message Queue.

related message queue based on the looper. Then add the new message to the end of the message queue.

Message class in Android defines a message containing a description and arbitrary data object that can be sent to a Handler. Although the constructor of Message is public, the best way to get one of these is to call *obtain()* method or *obtainMessage()* method of Handler to save resource costs.

### 5.5.2 Communication Mechanisms

Android provides the process-unit component model. All Android operations are expressed as Linux processes eventually. Android runs based on the Linux Kernel, and the memory, process, and file management



are controlled by the Linux Kernel. The system service is isolated by Linux processes for protection. To support mobile devices, all the default system functions of Android are provided as the server processes. Meanwhile, the functions realized by apps belong to application processes.

In Android, the server process and the application process are implemented by the class *Binder*. Binder is the most important part in Android, and it is the core part of a lightweight Remote Procedure Call (RPC) mechanism. We can derive directly from Binder to implement our own custom RPC protocol or simply instantiate a raw Binder object directly to use a token that can be shared across processes. RPC is a form of Inter Process Communication (IPC), which is a set of techniques for the exchange of data among multiple threads in one or more processes.

Android Interface Definition Language (AIDL) allows us to define the programming interface that both the client and service agree upon in order to communicate with each other using IPC. Normally, one process cannot access the memory of other processes. Processes need to decompose their objects into primitives that the operating system can understand and configure the objects across that boundary.

All system functions of Android are provided as a server process,

which makes the optimized communication method between processes extremely important. *Binder* refers to Kernel memory that is shared between all processes to minimize the overhead caused by memory copy. Furthermore, the RPC framework provided by Binder is written in C++, which is more efficient than Java.

In the RPC framework, the kernel space is a place where all processes can share and let each process refer to the memory address. In the Kernel space, a *Binder Driver* is implemented to use the kernel space to convert the memory address that each process has mapped with the memory address of the kernel space for reference. The Binder Driver supports the system call Input/Output Control (ioctl) and the file operations, including open, map, release, and poll. In computer science, ioctl is a system call for device-specific input/output operation and other operations which cannot be expressed by regular system calls.

Fig. 5.20 shows an example of the process of transmitting data from process A to process B. The first thing process A must do is to open the Binder kernel module, and this module uses the descriptor to identify the initiators and recipients of Binder IPCs. After defining the transmission (process A) and the reception (process B) of this operation, process A transmits the data to the Binder Driver first.

Then, the Binder Driver converts the memory address of the data to allow process B to access it.

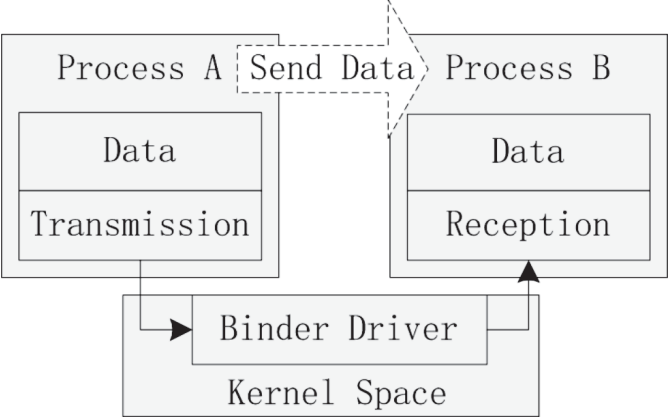


Figure 5.20 Transmit data via Binder Driver in RPC framework.