Servisi

Mobilne aplikacije

Stevan Gostojić

Fakultet tehničkih nauka, Novi Sad

29. oktobar 2024.

Pregled sadržaja

Process

- Proces je jedna instanca nekog programa koji se izvršava
- Karakterišu ga:
  - angažovanje procesora na izvršavanju programa
  - upotreba dela operativne memorije koji sadrži naredbe u mašinskom jeziku i podatke na stack-u i heap-u
  - atributi kao što su: ID, stanje, prioritet, itd.

Thread

- Thread odn. nit je redosled izvršavanja naredbi u procesu
- Jedan proces može da sadrži više niti (onda svaka nit sadrži stack, stanje i prioritet i izvršava relativno nezavisnu sekvencu naredbi)

Raspoređivanje niti

- Različite niti mogu da se izvršavaju na jednom procesoru (konkurentno) ili na više procesora (paralelno)
- Kako jedan procesor ne može istovremeno da izvršava više niti, one se moraju izvršavati naizmenično
- S obzirom da različite niti mogu da pristupaju istom resursu, potrebno je voditi računa o sinhronizaciji niti

Razlika između procesa i niti

- Niti se koriste za "male" zadatke, a procesi za "velike" zadatke (izvršavanje aplikacije)
- Niti koje pripadaju istom procesu dele isti adresni prostor (to znači da mogu da komuniciraju direktno preko operativne memorije)
- Procesi ne dele isti adresni prostor (to znači da je komunikacija između procesa složenija i sporija od komunikacije između niti)

Android i procesi

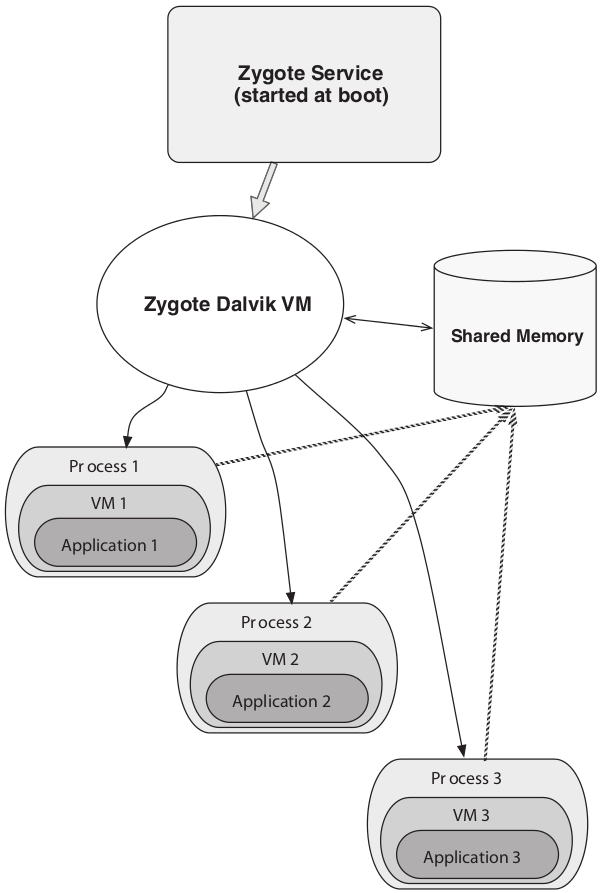


Figure 1: Android i procesi.

- Kada Android startuje prvu komponentu neke aplikacije, startuje je u novom procesu sa jednom niti
- Svaka sledeća komponenta iste aplikacije startuje se u istom procesu i u istoj niti kao i prva komponenta
- Moguće je startovati različite komponente iste aplikacije u različitim procesima ili različite komponente različitih aplikacija u istom procesu (mada to nije preporučljivo)

Android i procesi

- Android zadržava procese u operativnoj memoriji što je duže moguće
- Da bi se oslobodila memorija za procese višeg prioriteta, nekada je potrebno "ubiti" proces nižeg prioriteta
- Prioritet procesa se određuje na osnovu vrste i stanja komponenti koje sadrži kao i prioriteta drugih procesa koji od njega zavise
- Zato bi aktivnosti i prijemnici poruka koji izvršavaju dugačke operacije trebalo da startuju servis umesto niti

Prioritet procesa (u opadajućem poretku)

- `foreground` (proces sadrži aktivnost koja se nalazi u prvom planu)
- `visible` (proces sadrži vidljivu ali pauziranu aktivnost)
- `service` (proces sadrži servis)
- `cached` (proces je zaustavljen i može biti ili "ubijen" ili obnovljen povratkom sa neke druge aplikacije)

Android i niti

- Android izvršava aplikaciju (tj. njene komponente) u glavnoj niti
- Ova nit je, između ostalog, zadužena za slanje i primanje poruka od komponenti korisničkog interfejsa (zato se zove i UI nit)
- Stoga nije preporučljivo blokirati UI nit ("application isn't responding" dijalog) i pristupati komponentama korisničkog interfejsa iz drugih niti (nisu thread-safe)
- Metode životnog ciklusa servisa i dobavljača sadržaja moraju biti thread-safe

Android i niti

```
1 // Pogresno (blokiranje UI niti)
2 public void onClick(View v) {
3   Bitmap b = loadImageFromNetwork(imageUrl);
4   imageView.setImageBitmap(b);
5 }
6
```

Android i niti

```
1  // Pogresno (GUI komponente nisu thread-safe)
2  public void onClick(View v) {
3    new Thread(new Runnable() {
4      public void run() {
5        Bitmap b = loadImageFromNetwork(imageUrl);
6        imageView.setImageBitmap(b);
7      }
8    }).start();
9  }
10
```

Android i niti

```
1  // Ispravno
2  public void onClick(View v) {
3    new Thread(new Runnable() {
4      public void run() {
5        Bitmap b = loadImageFromNetwork(imageUrl);
6        imageView.post(new Runnable() {
7          public void run() {
8            imageView.setImageBitmap(b);
9          }
10       });
11     }
12   }).start();
13 }
14
```

Pregled sadržaja

Rukovaoci

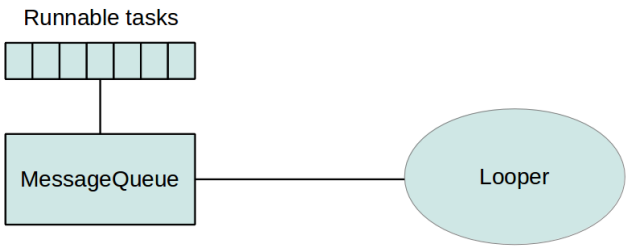


Figure 2: Rad sa nitima.

- Red poruka (`MessageQueue`) je red koji sadrži poruke koje je potrebno obraditi (zadatke koje je potrebno izvršiti)
- Rukovaoc (`Handler`) obrađuje poruke (izvršava zadatke) koje se nalaze u redu poruka
- Looper održava nit "u životu" i prosleđuje poruke (zadatke) iz reda poruka rukovaocu na obradu

Rukovaoci

```
1  class MyLooperThread extends Thread {
2    public Handler handler;
3
4    public void run() {
5      Looper.prepare();
6
7      handler = new Handler(Looper.myLooper()) {
8        public void handleMessage(Message msg) {
9          // process incoming messages here
10         // this will run in non-ui/background thread
11       }
12     };
13
14     Looper.loop();
15   }
16 }
17
```

Rukovaoci

```
1 MyLooperThread myLooperThread = new MyLooperThread();
2 myLooperThread.start();
3 // ...
4 Message message = new Message();
5 message.obj = "Hello world";
6 myLooperThread.handler.sendMessage(message);
7
```

Rukovaoci

```
new Handler(Looper.getMainLooper()).post(
  new Runnable() {
    @Override
    public void run() {
      // this will run in the main thread
    }
});

```

Rukovaoci

Za obradu poruka potrebno je implementirati `void handleMessage(Message msg)` i pozvati:

- `boolean sendMessage(Message)`
- `boolean sendMessageAtTime(Message, long)`
- `boolean sendMessageDelayed(Message, long)`

Za izvršavanje proizvoljnog koda potrebno je pozvati:

- `boolean post(Runnable)`
- `boolean postAtTime(Runnable, long)`
- `boolean postDelayed(Runnable, long)`

ExecutorService i Executors

- Za asinhrono izvršavanje operacija i raspolaganje nitima mogu se koristiti `ExecutorService` i `Executors` iz paketa `java.util.concurrent`.
- Interfejs `ExecutorService` definiše mehanizme za izvršavanje zasebnih niti, a klasa `Executors` kreira instance ovih servisa.
- Za potrebe komunikacije sa UI niti koristi se klasa `Handler`.
- Objekat tipa `ExecutorService` koji koristi jednu nit za izvršavanje pozadinskih zadataka:

```
1 ExecutorService executorService = Executors.newSingleThreadExecutor();
```

- Objekat tipa `ExecutorService` koji koristi više niti za izvršavanje pozadinskih zadataka:

```
1 int n = 10; // number of threads in the pool
2 ExecutorService executorService = Executors.newFixedThreadPool(n);
```

ExecutorService i Executors

```
1  public void backgroundImageDownload(String url) {
2
3      ExecutorService executorService = Executors.newSingleThreadExecutor();
4      Handler handler = new Handler(Looper.getMainLooper());
5
6      executorService.execute(new Runnable() {
7          @Override
8          public void run() {
9              Bitmap bitmap = loadImageFromNetwork(url);
10
11             handler.post(new Runnable() {
12                 @Override
13                 public void run() {
14                     ImageView iv = findViewById(R.id.imageView);
15                     iv.setImageBitmap(bitmap);
16                 }
17             });
18         }
19     });
20 }
21
```

Pregled sadržaja

WorkManager

- WorkManager omogućava izvršavanje pozadinskih zadataka kojima nije potrebna interakcija sa korisnikom.
- WorkManager poseduje podršku za širi skup verzija Android operativnog sistema i time prevazilazi izvesne razlike u upravljanju pozadinskim zadacima.
- Omogućava jednokratno i periodično izvršavanje zadataka.

build.gradle

```
1
2 implementation "androidx.work:work-runtime:2.9.0"
3
4
```

ExampleWorker.java

```
1  public class ExampleWorker extends Worker {
2
3    public ExampleWorker(@NonNull Context context,
4                         @NonNull WorkerParameters params){
5      super(context, params);
6    }
7
8    @Override
9    public Result doWork() {
10     // Do the work here
11     // ...
12
13     // Indicate whether the work finished successfully
14     return Result.success();
15   }
16
17 }
18
```

Result

Vrednost	Opis
Result.success()	The work finished successfully
Result.failure()	The work failed
Result.retry()	The work failed and should be tried

Table 1: Povratne vrednosti doWork metode.

ExampleActivity.java (jednokratno izvršavanje)

```
1 WorkRequest request =
2     new OneTimeWorkRequest.Builder(ExampleWorker.class)
3         .build();
4
5 WorkManager.getInstance(this).enqueue(request);
6
```

ExampleActivity.java (jednokratno izvršavanje sa ograničenjima)

```
1  Constraints constraints = new Constraints.Builder()
2      .setRequiredNetworkType(NetworkType.UNMETERED)
3      .setRequiresCharging(true)
4      .setRequiresBatteryNotLow(true)
5      .setRequiresStorageNotLow(true)
6      .setRequiresDeviceIdle(true)
7      .build();
8
9  WorkRequest request =
10     new OneTimeWorkRequest.Builder(ExampleWorker.class)
11       .setConstraints(constraints)
12       .build();
13
14 WorkManager.getInstance(this).enqueue(request);
15
```

ExampleActivity.java (periodično izvršavanje)

```
1 PeriodicWorkRequest request =
2   new PeriodicWorkRequest.Builder(
3     ExampleWorker.class, 1, TimeUnit.HOURS).build();
4
5 WorkManager.getInstance(this).enqueue(request);
6
```

Pregled sadržaja

Servisi

- Servis je komponenta koja izvršava "duge" operacije.
- Postoje tri vrste servisa: servis u prvom planu (foreground service), servis u pozadini (background service) i vezan servis (bound service).
  - Servis u prvom planu izvršava uočljivu operaciju (npr. reprodukcija muzike)
  - Servisu u pozadini nije potrebna interakcija sa korisnikom (npr. file download)
  - Vezan servis služi za implemantaciju klijent-server arhitekture
- Servis u prvom planu i servis u pozadini se nazivaju i startovani servisi (started service)

Servisi

- Servis se izvršava u istoj niti u kojoj se izvršavala komponenta koja ga je startovala (čak i ako ta komponenta više nije aktivna)
- Druga komponenta može da se veže za servis i da sa njime komunicira (čak i ako se nalazi u drugom procesu)

Servisi

- Servis može biti startovan ili vezan (može istovremeno biti i startovan i vezan, ali se to retko koristi)
- Startovan servis se izvršava neodređeno vreme (servis treba da se sam zaustavi kada izvrši operaciju)
- Vezan servis se izvršava samo dok je neka komponenta vezana za njega (nudi interfejs koji omogućava komponentama da komuniciraju sa njim šaljući zahteve i dobijajući odgovore)

Životni ciklus servisa

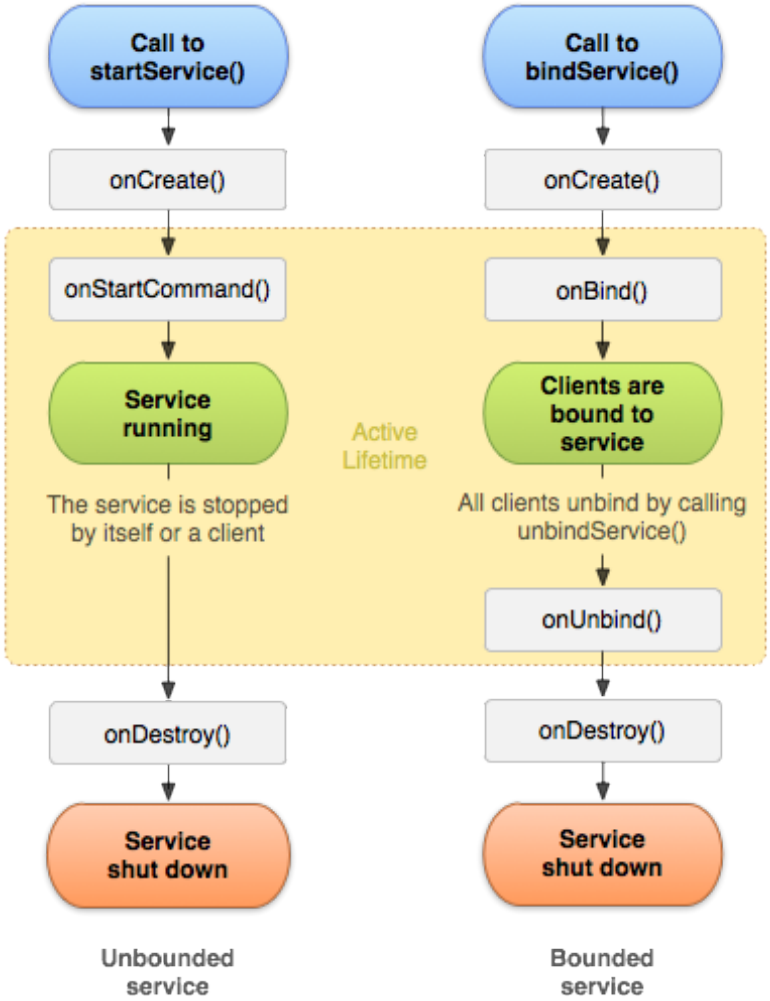


Figure 3: Životni ciklus servisa.

Životni ciklus servisa

Servisi, poput aktivnosti, sadrže metode koje se pozivaju prilikom prelaska iz jednog u drugo stanje:

- `onCreate` (poziva se prilikom stvaranja servisa)
- `onStartCommand` (poziva se posle poziva startService metode)
- `onBind` (poziva se posle poziva bindService metode)
- `onUnbind` (poziva se posle poziva unbindService metode)
- `onRebind` (poziva se posle poziva bindService ako je prethodno izvršena onUnbind metoda)
- `onDestroy` (poziva se prilikom uništavanja servisa)

Životni ciklus servisa

- Razlikuje se ceo životni vek servisa (između poziva `onCreate` i `onDestroy` metoda) i
- aktivni životni vek (počinje pozivom `onStartCommand` ili `onBind` metode, a završava se pozivom `onDestroy` ili `onUnbind` metode)

Pravljenje servisa

- Servis može da se implementira nasleđivanjem klase `Service` uz dodavanje odgovarajuće deklaracije u `AndroidManifest`
- Važno je koristiti pozadinsku nit u kojoj će se izvršiti operacije i voditi računa u sinhronizaciji ukoliko više komponenti istovremeno koriste isti servis

AndroidManifest.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest ... >
3   <application ... >
4     <service android:name=".ExampleService" />
5   </application>
6 </manifest>
7
```

ExampleService.java

```
1  public class ExampleService extends Service {
2    @Override
3    public void onCreate() {
4      // ...
5    }
6
7    @Override
8    public int onStartCommand(Intent intent, int flags, int startId) {
9      // ...
10     // If we get killed, after returning from here, restart
11     return START_STICKY;
12   }
13
14   @Override
15   public IBinder onBind(Intent intent) {
16     // We don't provide binding, so return null
17     return null;
18   }
19
20   public void onDestroy() {
21     // ...
22   }
23 }
24
```

Servisi

Constant	Meaning
`START_NOT_STICKY`	If the system kills the service after `onStartCommand()` returns, do not recreate the service.
`START_STICKY`	If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()`.
`START_REDELIVER_INTENT`	If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()` with the last intent delivered.

Table 2: Vrednosti `flags` parametra.

Pokretanje startovanog servisa

```
1 Intent intent = new Intent(this, ExampleService.class);
2 startService(intent);
3
```

Pristup vezanom servisu

```
1  ServiceConnection connection = new ServiceConnection() {
2    @Override
3    public void onServiceConnected(ComponentName className,IBinder service) {
4      // ...
5    }
6
7    @Override
8    public void onServiceDisconnected(ComponentName arg0) {
9      // ...
10   }
11 };
12
13
14 // binding to the service
15 Intent intent = new Intent(this, ExampleService.class);
16 bindService(intent, connection, Context.BIND_AUTO_CREATE);
17
18
19 // unbinding
20 unbindService(connection);
21
```

Zaustavljanje servisa

- Servis se može zaustaviti sam pozivom stopSelf metode, može ga zaustaviti druga komponenta pozivom stopService metode ili ga može zaustaviti Android platforma (da bi oslobodila memoriju)
- Aplikacije bi trebalo da zaustave svoje servise čim izvrše operaciju da se ne bi trošili resursi (npr. baterija)

Pokretanje servisa u prvom planu

- Servis se može pokrenuti u prvom planu pozivom startForeground metode, a ukloniti iz prvog plana pozivom stopForeground metode
- Trebalo bi da se nalazi u prvom planu ukoliko je korisnik svestan servisa (što znači da ne treba da se "ubije" u nedostatku memorije)
- Servis u prvom planu mora obezbediti obaveštenje u statusnoj liniji

ExampleService.java

```
1  public class ExampleService extends Service {
2    @Override
3    public int onStartCommand(Intent intent, int flags, int startId) {
4      // ...
5      Notification notification = ...;
6      startForeground(ONGOING_NOTIFICATION_ID, notification);
7      // ...
8      stopForeground(true);
9      // ...
10 }
11
```

Pregled sadržaja

Prijemnici poruka

- Prijemnici poruka (`BroadcastReceiver`) obrađuju događaje (opisane objektima klase Intent)
- Te događaje može da izazove Android platforma ili druga komponenta (koja može da se nalazi u drugoj aplikaciji)

AndroidManifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest ... >
3    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
4    <!-- dynamic permission request should also be made -->
5
6    <application ... >
7      <receiver android:name=".SMSReceiver">
8        <intent-filter>
9          <action name="android.provider.Telephony.SMS_RECEIVED" />
10       </intent-filter>
11     </receiver>
12   </application>
13 </manifest>
14
```

SMSReceiver.java

```
1 public class SMSReceiver extends BroadcastReceiver {
2   @Override
3     public void onReceive(Context context, Intent intent) {
4       // ...
5   }
6 }
7
```

Prijemnici poruka

Parametri `onReceive` metode su:

- `context` (kontekst u kome se izvršava onReceive metoda)
- `intent` (namera koja opisuje događaj koji treba obraditi)

Namera prosleđena `startActivity` ili `startService` metodi neće prouzrokovati događaj koji će obraditi `onReceive` metoda (važi i obrnuto)

Prijemnici poruka

Konstanta	Značenje
`ACTION_BATTERY_LOW`	A warning that the battery is low.
`ACTION_POWER_CONNECTED`	External power has been connected to the device.
`ACTION_HEADSET_PLUG`	A headset has been plugged into the device, or unplugged from it.
`ACTION_SCREEN_ON`	The screen has been turned on.
`ACTION_SHUTDOWN`	Device is shutting down.

Table 3: Neke od akcija (događaja) koje se mogu obraditi.

Prijemnici poruka

- Metoda `onReceive` se poziva iz glavne niti (to znači da "dugačke" operacije treba izvršavati u posebnom servisu koji startuje posebnu nit)
- Prijemnik poruka postoji samo u toku izvršavanja `onReceive` metode (zato se u ovoj metodi ne mogu izvršiti asinhrone operacije kao što su prikazivanje dijaloga ili vezivanje za servis)
- Proces u kome se izvršava `onReceive` metoda ima foreground prioritet (nakon toga, prioritet procesa određuju ostale komponente koje se u njemu nalaze)

Prijemnici poruka

- Informacije o događajima se do prijemnika poruka prenose putem namera.
- Kao što namere mogu biti eksplicitne i implicitne tako se i broadcast poruke mogu podeliti na eksplicitne (namenjene konkretnom prijemniku poruka) i implicitne (prijemnik poruka se pronalazi na osnovu filtera namera).

Registrovanje prijemnika poruka

- Prijemnik poruka je moguće registrovati statički (putem manifest fajla) ili dinamički (iz programkog koda).
- Primer statički registrovanog prijemnika poruka:

```
1  <receiver android:name=".SMSReceiver">
2    <intent-filter>
3      <action name="android.provider.Telephony.SMS_RECEIVED" />
4    </intent-filter>
5  </receiver>
6
```

Registrovanje prijemnika poruka

- Primer dinamički registrovanog prijemnika poruka:

```
1 MyBroadcastReceiver myBroadcastReceiver = new MyBroadcastReceiver();
2 IntentFilter filter = new IntentFilter("android.provider.Telephony.
      SMS_RECEIVED");
3 registerReceiver(myBroadcastReceiver, filter);
4
```

- Pri tome je prijemnik poruka implementiran na uobičajen način:

```
1 MyBroadcastReceiver extends BroadcastReceiver {
2
3     @Override
4     public void onReceive(Context context, Intent intent) {
5       // ...
6     }
7 }
8
```

Prijemnici poruka

- Sistemski događaji koriste implicitne namere kako bi različite komponente mogle primiti poruke o tim događajima.
- Prijavljivanje prijemnika poruka na određenu vrstu implicitno definisanih poruka može dovesti do preopterećenja resursa uređaja ukoliko postoji veći broj prijemnika poruka koji očekuju istu vrstu poruke.
- Tako je za neke implicitne broadcast poruke koje generiše sistem dovoljno koristiti <intent-filter>. To su one poruke koje se ne dešavaju često ili se ređe javlja potreba za obradom te vrste događaja. Primeri takvih broadcast poruka su opisani akcijama: ACTION_BOOT_COMPLETED, ACTION_TIMEZONE_CHANGED, ACTION_USB_DEVICE_ATTACHED, ACTION_USB_DEVICE_DETACHED, ACTION_MEDIA_MOUNTED, ACTION_MEDIA_UNMOUNTED.

Prijemnici poruka

- Za ostale sistemske događaje je potrebno koristi dinamičku registraciju prijemnika poruka. Na primer, to je slučaj sa događajima opisanim akcijom: CONNECTIVITY_ACTION
- Pored dinamičke registracije prijemnika poruka, može biti neophodan i dinamički (runtime) zahtev za odgovarajućim permisijama potrebnim za obradu te vrste događaja.
- Osim toga, za neke sistemske događaje u novijim verzijama Androida se ne emituju broadcast poruke. To je slučaj sa: ACTION_NEW_PICTURE, ACTION_NEW_VIDEO.

Prijemnici poruka

Postoje dva vrste događaja koje prijemnici poruka mogu obraditi:

- normalni događaji (asinhroni su, prijemnici ih obrađuju nedefinisanim redosledom i njihova obrada je efikasnija)
- uređeni događaji (obrađuje ih više prijemnika redom, a svaki prijemnik može da prosledi događaj sledećem prijemniku ili da potpuno obustavi njegovu obradu)

ExampleActivity.java

```
1 public class ExampleActivity extends Activity {
2   @Override
3   public void onCreate(Bundle bundle) {
4     Intent intent = new Intent();
5     intent.setAction(SMS_RECEIVED);
6     sendBroadcast(intent);
7   }
8 }
9
```

SMSReceiver.java

```
1 public class SMSReceiver extends BroadcastReceiver {
2   @Override
3   public void onReceive(Context context, Intent intent) {
4     // ...
5   }
6 }
7
```

ExampleActivity.java

```
1  public class ExampleActivity extends Activity {
2    @Override
3    public void onCreate(Bundle bundle) {
4      Intent intent = new Intent();
5      intent.setAction(SMS_RECEIVED);
6      sendOrderedBroadcast(intent, null);
7    }
8  }
9
10
11 // 2nd parameter is null because no special permissions
       are needed
12
```

AndroidManifest.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest ... >
3    <application ... >
4      <receiver android:name=".SMSReceiver">
5        <intent-filter android:priority="100">
6          <action name="android.provider.Telephony.SMS_RECEIVED" />
7        </intent-filter>
8      </receiver>
9    </application>
10 </manifest>
11
```

SMSReceiver.java

```
1  public class SMSReceiver extends BroadcastReceiver {
2    @Override
3    public void onReceive(Context context, Intent intent) {
4      // ...
5      if (condition) {
6          abortBroadcast();
7      }
8    }
9  }
10
```

Prijemnici poruka

Metoda	Značenje
`abortBroadcast()`	Sets the flag indicating that this receiver should abort the current broadcast.
`getResultCode()`	Retrieve the current result code, as set by the previous receiver.
`getResultData()`	Retrieve the current result data, as set by the previous receiver.
`setResultCode(int code)`	Change the current result code of this broadcast.
`setResultData(String data)`	Change the current result data of this broadcast.

Table 4: Metode koje se koriste za spregu između prijemnika poruka.

Prijemnici poruka

- Prijemnici poruka omogućavaju razmenu poruka između komponenti različitih aplikacija
- To znači da treba obratiti pažnju na moguće zloupotrebe
- Moguće je primeniti prava pristupa prilikom slanja poruke (prosleđujući permission parametar `sendBroadcast` metodi) ili prilikom prijema poruke (postavljajući permission atribut receiver elementa)
- Komponenta koja salje/prima poruku mora imati odgovarajuće pravo pristupa (zatraženo `<uses-permission>` elementom u `AndroidManifest.xml`)

Prijemnici poruka i permisije - prijem poruke

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>

    <!-- We define custom permission -->
    <permission android:name="my.app.PERMISSION" />
  <!-- Additional protection can be set using the protection level:
   android:protectionLevel="signature" will require the sender app to be signed
     with the same certificate,
   android:protectionLevel="dangerous" will asks user to grant this permission
    -->

    <!-- The receiver demands this permission -->
    <receiver
        android:name="my.app.BroadcastReceiver"
        android:permission="my.app.PERMISSION">
        <intent-filter>
            <action android:name="my.app.Action" />
        </intent-filter>
    </receiver>
     ...
</manifest>
```

Prijemnici poruka i permisije - slanje poruke

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>

    <!-- we want to use our custom permission -->
    <uses-permission android:name="my.app.PERMISSION" />
    <!--  now this app is allowed to send broadcasts
          to the application that declared this permission -->
    <!--  depending on the protection level it may be necessary
          to sign the app or the user to grant this permission -->
     ...
</manifest>

```

Agenda

Zakazivanje

- Tajmer (`Timer`) zakazuje izvršavanje jednokratnih zadataka (u apsolutnom trenutku ili posle relativnog kašnjenja) ili zadataka koji se ponavljaju (sa fiksnim periodom ili fiksnom frekvencijom)
- Svaki tajmer ima jednu nit koja zadatke izvršava sekvencijalno (to znači da može da dođe do kašnjenja u izvršavanju zadatka ukoliko je ta nit zauzeta)
- Komponenta koja je zakazala izvršavanje zadatka ne mora biti aktivna u trenutku u kome zadatak treba da se izvrši (to znači da zadatak može da se ne izvrši)

Zakazivanje

Metoda	Značenje
`schedule(TimerTask task, Date when)`	Schedule a task for single execution when a specific time has been reached.
`schedule(TimerTask task, long delay)`	Schedule a task for single execution after a specified delay.
`cancel()`	Cancels the Timer and all scheduled tasks.

Table 5: Metode klase `Timer`.

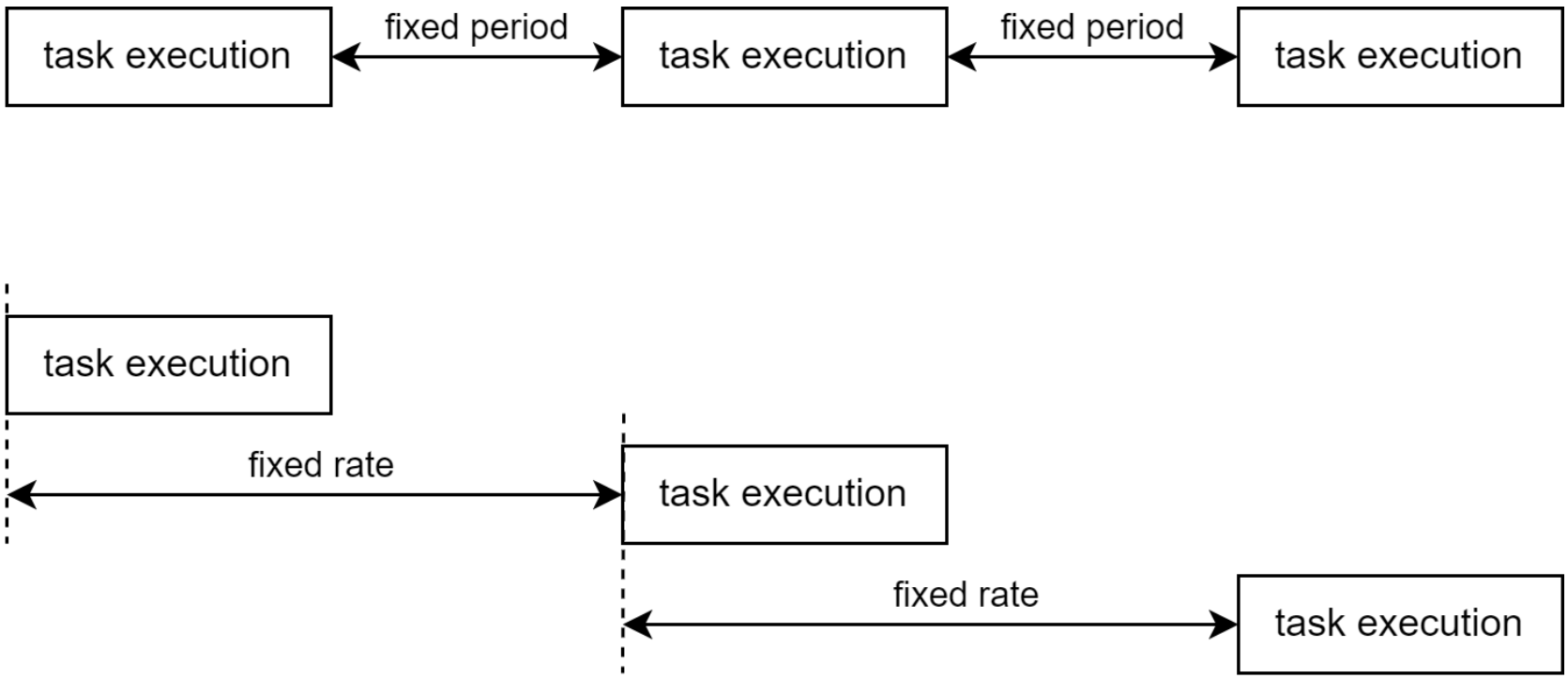
Zakazivanje

Metoda	Značenje
`schedule(TimerTask task, Date when, long period)`	Schedule a task for repeated fixed-delay execution after a specific time has been reached.
`schedule(TimerTask task, long delay, long period)`	Schedule a task for repeated fixed-delay execution after a specific delay.
`scheduleAtFixedRate (TimerTask task, long delay, long period)`	Schedule a task for repeated fixed-rate execution after a specific delay has passed.
`scheduleAtFixedRate (TimerTask task, Date when, long period)`	Schedule a task for repeated fixed-rate execution after a specific time has been reached.

Table 6: Metode klase `Timer`.

schedule vs. scheduleAtFixedRate

- izvršavanje taskova pomoću schedule metode (gornji primer) i scheduleAtFixedRate metode (donji primer)



SplashScreen.java

```
1  public class SplashScreen extends Activity {
2
3    public static final int SPLASH_TIMEOUT = 1500;
4
5    @Override
6    protected void onCreate(Bundle bundle) {
7      super.onCreate(bundle);
8      setContentView(R.layout.splash_screen);
9      new Timer().schedule(new TimerTask() {
10         @Override
11         public void run() {
12           startActivity(new Intent(SplashScreen.this, MyMovies.class));
13           finish();
14         }
15       }, SPLASH_TIMEOUT);
16   }
17 }
18
```

Zakazivanje

- Klasa `AlarmManager` omogućuje pristup sistemskom alarmu i startovanje aplikacije u nekom trenutku u budućnosti
- Kada se alarm aktivira, sistem emituje objekat klase `Intent`, što kao posledicu ima automatsko startovanje aplikacije (ukoliko već nije startovana)

AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application ... >

    <receiver android:name=".PortfolioStartupReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
      </intent-filter>
    </receiver>

    <receiver android:name=".AlarmReceiver">
      <!-- -->
    </receiver>

    <service android:name=".PortfolioManagerService">
      <!-- -->
    </service>

  </application>
</manifest>

```

PortfolioStartupReceiver.java

```
1  public class PortfolioStartupReceiver extends BroadcastReceiver {
2
3    private static final int FIFTEEN_MINUTES = 15*60*1000;
4
5    @Override
6    public void onReceive(Context context, Intent intent) {
7      AlarmManager mgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
8      Intent i = new Intent(context, AlarmReceiver.class);
9      PendingIntent pi = PendingIntent.getBroadcast(
10         context, 0, i, PendingIntent.FLAG_CANCEL_CURRENT);
11     Calendar now = Calendar.getInstance();
12     now.add(Calendar.MINUTE, 2);
13     mgr.setRepeating(
14         AlarmManager.RTC_WAKEUP, now.getTimeInMillis(),FIFTEEN_MINUTES, pi);
15   }
16 }
17
```

AlarmReceiver.java

```
1 public class AlarmReceiver extends BroadcastReceiver {
2   @Override
3   public void onReceive(Context context, Intent intent) {
4     Intent stockService = new Intent(context, PortfolioManagerService.class);
5     context.startService(stockService);
6   }
7 }
8
```

PortfolioManagerService.java

```
1 public class PortfolioManagerService extends Service {
2   @Override
3   public int onStartCommand(Intent intent, int flags, int startId) {
4     updateStockData();
5     return Service.START_NOT_STICKY;
6   }
7 }
8
```