Servisi

Mobilne aplikacije

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

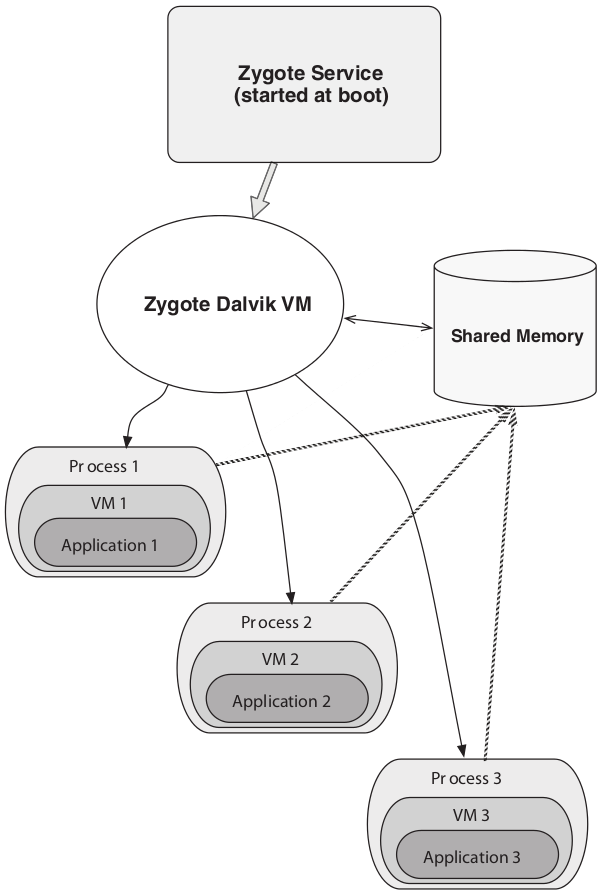

Figure 1: Android i procesi.

- 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)
- `background` (proces sadrži zaustavljenu aktivnost koja se nalazi u pozadini)
- `empty` (proces ne sadrži komponente)

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

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

```

Android i niti

```
// Pogresno (GUI komponente nisu thread-safe)
public void onClick(View v) {
  new Thread(new Runnable() {
    public void run() {
      Bitmap b = loadImageFromNetwork(imageUrl);
      imageView.setImageBitmap(b);
    }
  }).start();
}

```

Android i niti

```
// Ispravno
public void onClick(View v) {
  new Thread(new Runnable() {
    public void run() {
      Bitmap b = loadImageFromNetwork(imageUrl);
      imageView.post(new Runnable() {
        public void run() {
          imageView.setImageBitmap(b);
        }
      });
    }
  }).start();
}

```

Pregled sadržaja

Rukovaoci

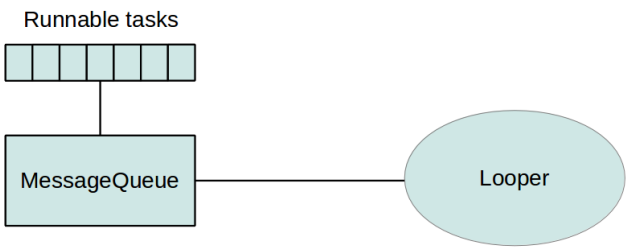

Figure 2: Rad sa nitima.

- 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

```
class MyLooperThread extends Thread {
  public Handler handler;

  public void run() {
    Looper.prepare();

    handler = new Handler(Looper.myLooper()) {
      public void handleMessage(Message msg) {
        // process incoming messages here
        // this will run in non-ui/background thread
      }
    };

    Looper.loop();
  }
}
```

Rukovaoci

```
MyLooperThread myLooperThread = new MyLooperThread();
myLooperThread.start();
// ...
Message message = new Message();
message.obj = "Hello world";
myLooperThread.handler.sendMessage(message);
```

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 sendEmptyMessage(int)`
- `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)`

Pregled sadržaja

Asinhroni zadatak

- Do API level 30 za ainhrono izvršavanje operacija se koristio `AsyncTask`.
- Na taj način je automatizovano izvršavanje blokirajuće operacije u pozadinskoj niti, vraćanje rezultata u UI nit i neke dodatne funkcije (kao što je obaveštavanje o progresu operacije).
- Svi asinhroni zadaci jedne aplikacije izvršavaju se u jednoj niti (oni se serijalizuju)
- `AsyncTask` je generička klasa koja koristi tri tipa:
  - `params` (tip parametara koji se prosleđuju pozadinskoj niti)
  - `progress` (tip jedinice u kojoj se meri progres operacije)
  - `result` (tip povratne vrednosti koju vraća pozadinska nit)

Asinhroni zadatak

- Poziv asinhronog zadatka

```
public void onClick(View v) {
  ImageView imageView = (ImageView)findVideById(R.id.imageView);
  new DownloadImageTask(imageView).execute(imageUrl);
}
```

- Implementacija asinhronog zadatka

```
public class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
  private ImageView imageView;

  public DownloadImageTask(ImageView imageView) {
    this.imageView = imageView;
  }
  // The system calls this to perform work in a worker
  // thread and delivers it the parameters given to execute()
  protected Bitmap doInBackground(String url) {
    return loadImageFromNetwork(url);
  }

  // The system calls this to perform work in the UI
  // thread and delivers the result from doInBackground()
  protected void onPostExecute(Bitmap result) {
    imageView.setImageBitmap(result);
  }
}
```

ExecutorService i Executors

- Od API level 30 za ainhrono izvršavanje operacija se koriste `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:

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

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

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

ExecutorService i Executors

```
public void backgroundImageDownload(String url) {

    ExecutorService executorService = Executors.newSingleThreadExecutor();
    Handler handler = new Handler(Looper.getMainLooper());

    executorService.execute(new Runnable() {
        @Override
        public void run() {
            Bitmap bitmap = loadImageFromNetwork(url);

            handler.post(new Runnable() {
                @Override
                public void run() {
                    ImageView iv = findViewById(R.id.imageView);
                    iv.setImageBitmap(bitmap);
                }
            });
        }
    });
}
```

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

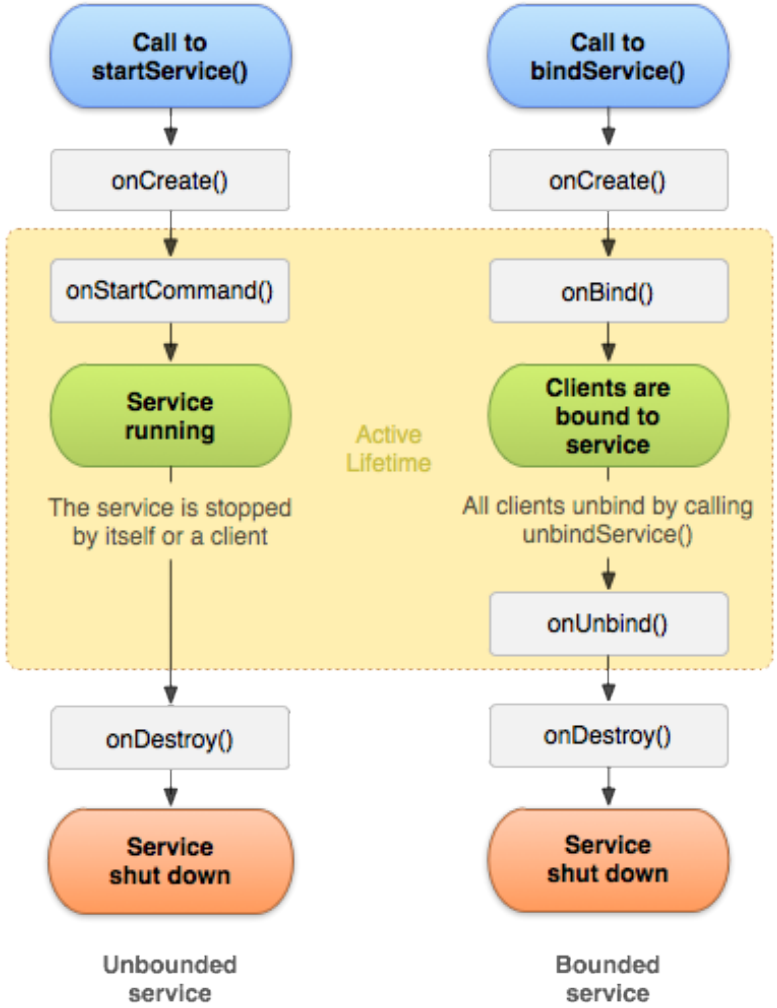

Figure 3: Životni ciklus servisa.

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 napravi nasleđivanjem klasa:

- `Service` (u ovom slučaju je važno startovati pozadinsku nit u kojoj će se izvršiti operacije i voditi računa u sinhronizaciji ukoliko više komponenti istovremeno koriste isti servis)
- `IntentService` (u ovom slučaju će se operacije automatski izvršiti u pozadinskoj niti i pozivi metoda servisa će se automatski sinhronizovati). Od verzije Android 11 (API level 30) koristi se `JobIntentService`

Usled različitosti u načinu izvršavanja servisa na starijim i novijim platformama, preporučuje se `WorkManager` koji poseduje podršku za uređaje sve do API level 14

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application ... >
    <service android:name=".ExampleService" />
  </application>
</manifest>
```

ExampleService.java

```
public class ExampleService extends Service {
  @Override
  public void onCreate() {
    // ...
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    // ...

    stopSelfResult(startId);

    // If we get killed, after returning from here, restart
    return START_STICKY;
  }

  @Override
  public IBinder onBind(Intent intent) {
    // We don't provide binding, so return null
    return null;
  }

  public void onDestroy() {
    // ...
  }
}
```

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 1: Vrednosti `flags` parametra.

ExampleService.java

```
public class ExampleService extends IntentService {
  // A constructor is required, and must call the super
  // IntentService(String) constructor with a name for
  // the worker thread
  public ExampleService() {
    super("ExampleService");
  }

  // The IntentService calls this method from the default
  // worker thread with the intent that started the service.
  // When this method returns, IntentService stops the
  // service, as appropriate.
  @Override
  protected void onHandleIntent(Intent intent) {
    // ...
  }
}

```

Pokretanje servisa

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

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

```
public class ExampleService extends Service {
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    // ...
    Notification notification = ...;
    startForeground(ONGOING_NOTIFICATION_ID, notification);
    // ...
    stopForeground(true);
    // ...
}
```