Projektovanje namenskih računarskih struktura – praktikum

Verzija 3.00

Vezija	Datum	Opis
v1.05		• [Košutić, Vranić] Dodat CallStack primer. Navedeno da se dump koristi za JB , a log za novije verzije • [Vranic] Dodat ashmem primer. • [Vranic] Alpha player: ffmpeg media player za Jelly Bean • [Vranic] Cult player: primer za media codec. Tekst nije formatiran, treba da se sredi.
v1.06	25. Mart 2016.	• [Vranic] Sređen tekst i primer za media codec
v1.07	29. Mart 2016.	• [Košutić] Android svojstva (properties) • [Košutić] Native Servis • [Košutić] Init
v1.08	1. April 2016.	• [Košutić] Android svojstva (pristup iz jave) • [Košutić] Android svojstva (definicija iz init)
v1.09	1. April 2016.	• [Košutić] Android svojstva (prisup iz jave) PRIVREMENO IZBACENO
v1.10	10. Maj 2016.	• [Košutić] Binder primer: native service - native client
v1.11	14. Maj 2016.	• [Košutić] JNI primer
v2.11	20. Maj 2016.	Osnova je 1.11 korišćena za predavanja na Matematičkom fakultetu u Beograd-u. Izmene: • [Vranic] Svako poglavlje je prošireno sa kratkim opisom šta se sve očekuje da polaznika kursa savlada u toj vežbi. Takođe, navedeno je šta sve i gde polaznik treba da podigne rešenje ve žbe. • [Vranic] Dodat je u uvod podešavanje Java VM mašine kao i instalacija dodatnih paketa neophodnih za rad sa AOSP kod-om
v2.12	25. Maj 2016.	• [Vranic] Pretraga koda: lokalno pretraživanje vs pretraživanje koda na server-u • [Košutić] Binder C/C++ - C/C++: primer binder komunikacije između dva native procesa • [Košutić] Binder C/C++ - Java: primer binder komunikacije između native servisa i Java aplikacije • [Vranic] Beta player: ffmpeg media player koji se dodaje u okviru media player servisa radi reprodukcije video sadržaja iz lokalne datoteke na novijim verzijama Android-a
v2.13	26. Maj 2016.	• [Košutić] Java Android svojstva
v2.14	26. Maj 2016.	• [Vranić] Ashmem primer C/C++ - Java • [Vranić] Binder C/C++ - Java, složeni tipovi podataka
v2.15	30. Maj 16	• [Jakovljević] HAL vežba • [Vranić] Ashmem, JNI, Binder opcioni zadatak
v2.16	1. Jun 2016.	• [Vranić] Delta player, NDK media player
v.2.17	27. Avgust 2016.	• [Vranić] Ispravka sitnih grešaka u tekstu.

V2.18	8. Avgust 2016.	• [Vranić] Binder callback native servis-java klijent opcioni zadatak. • [Vranić] Osvežen Beta player • [Jakovljević] Osvežen HAL zadatak • [Jakovljevič] Opcioji HAL zadatak
V3.0	27. Oktobar	• Verzija za predmet PNRS2 2016/2017.

[1] Radno okruženje

Očekivani rezultati ove vežbe su:

- Podešavanje Java programski paket za rad sa AOSP kodom.
- Instalacija dodatnih alata potrebnih za rad sa AOSP kodom.

Android Open Source Project (AOSP) je operativni sistem otvorenog programskog koda. Celokupno programsko rešenje nalazi se na javnim repozitorijumima, sa kojih svako može da preuzme postojeće rešenje i pruži svoj doprinos u vidu izmena. Detaljno uputstvo za preuzimanje programskog koda nalazi se na sledećoj adresi https://source.android.com/source/downloading.html.

Za prevođenje AOSP programskog rešenja potreban je sledeći skup alata:

1. Ubuntu 14.04 LTS
2. Java Development Kit (JDK)
3. Dodatni alati

Usled AOSP izmena ili promene aktuelne verzije operativnog sistema za razvoj,skup alata podložan je čestim promenama. Uvek aktuelno uputstvo za podešavanje radnog okruženja nalazi se na sledećoj adresi https://source.android.com/source/initializing.html.

Za potrebe ovog kursa AOSP kod je već preuzet i nalazi se na vašim lokalnim računarima na putanji: /home/rtrk/android-5.1.1_r9. Pošto je za potrebe kursa potrebno prevesti AOSP kod koje u zavisnosti od procesorske snage i količine operativne memorije može da potraje između 1 i 2 sata, prevođenje je urađeno unapred. Za potrebne kursa vežbe će se izvoditi na AOSP kodu 5.1.1 r9.

Preuzmite Java razvojno okruženje x64 za Linux sa sledeće stranice: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Kopirajte sadržaj arhive na putanju /usr/lib/jvm/oracle_jdk8. Pomoću update-alternatives alata i install opcije dodajte novu verziju Jave u sistem

```
rtrk@rtrk:~/android-5.1.1_r9$ sudo update-alternatives --install
"/usr/bin/java" "java" "/usr/lib/jvm/oracle_jdk8/bin/java" 1
rtrk@rtrk:~/android-5.1.1_r9$ sudo update-alternatives --install
"/usr/bin/javac" "javac" "/usr/lib/jvm/oracle_jdk8/bin/javac" 1
```

Ukoliko radite sa Android Jelly Bean ili Lollipop verzijama podesite pomoću config opcije da verzija Jave bude 1.7. Ukoliko koristi noviju verziju Android-a podesite da verzija Jave bude ova koju je upravo instalirana.

```
rtrk@rtrk:~/android-5.1.1_r9$ sudo update-alternatives --config
java
rtrk@rtrk:~/android-5.1.1_r9$ sudo update-alternatives --config
javac
```

Sa komandom java –version proverite da je ste dobro podesili Java razvojno okruženje.

Poslednji korak u podešavanju radnog okruženja je instalacija dodatnih alata.

Sa apt-get komandom instalirajte dodatne alate:

```
rtrk@rtrk:~/android-5.1.1_r9$ sudo apt-get install git-core gnupg
flex bison gperf build-essential zip curl zlib1g-dev gcc-multilib
g++-multilib libc6-dev-i386 lib32ncurses5-dev x11proto-core-dev
libx11-dev lib32z-dev ccache libgl1-mesa-dev libxml2-utils
xsltproc unzip
```

[2] Prevođenje

Očekivani rezultati ove vežbe su:

- Inicijalizacija terminala za rad sa AOSP kod-om.
- Pokretanje sistema za prevođenje koda.
- Pokretanje emulatora.

Prevođenje AOSP programskog rešenja izvršava se isključivo posredstvom terminala iz AOSP direktorijuma (/home/rtrk/android-5.1.1_r9). Pre nego što se pokrene prevođenje AOSP paketa potrebno je izvršiti inicijalizaciju terminala koja se izvodi u dva koraka.

Inicijalizacija

Prvi korak prevođenja predstavlja inicijalizacija sistema za prevođenje i izvršava se na sledeći način:

```
rtrk@rtrk:~/android-5.1.1_r9$ source build/envsetup.sh
including device/asus/deb/vendorsetup.sh
including device/asus/flo/vendorsetup.sh
including device/asus/fugu/vendorsetup.sh
including device/asus/grouper/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
including device/generic/mini-emulator-arm64/vendorsetup.sh
including device/generic/mini-emulator-armv7-a-neon/vendorsetup.sh
including device/generic/mini-emulator-mips/vendorsetup.sh
including device/generic/mini-emulator-x86_64/vendorsetup.sh
including device/generic/mini-emulator-x86/vendorsetup.sh
including device/htc/flounder/vendorsetup.sh
including device/lge/hammerhead/vendorsetup.sh
including device/lge/mako/vendorsetup.sh
including device/moto/shamu/vendorsetup.sh
including device/samsung/manta/vendorsetup.sh
including vendor/rtrk/vendorsetup.sh
including sdk/bash_completion/adb.bash
```

Inicijalizacijom se postavlja okruženje za prevođenje koje važi za trenutnu seansu terminala, iz čega sledi da je inicijalizaciju potrebno izvršiti sa svakim ponovnim pokretanjem terminala.

Izbor konfiguracije

AOSP je programsko rešenje predviđeno za rad na velikom broju različitih uređaja. Za svaki uređaj moguće je kreirati različitu softversku konfiguraciju. Pre prevođenja potrebno je odabrati željenu konfiguraciju uz pomoć lunch komande.

```
rtrk@rtrk:~/android-5.1.1_r9$ lunch
```

```
You're building on Linux

Lunch menu... pick a combo:
     1. aosp_arm-eng
     2. aosp_arm64-eng
     3. aosp_mips-eng
     4. aosp_mips64-eng
     5. aosp_x86-eng
     6. aosp_x86_64-eng
     7. aosp_deb-userdebug
     8. aosp_flo-userdebug
     9. full_fugu-userdebug
     10. aosp_fugu-userdebug
     11. aosp_grouper-userdebug
     12. aosp_tilapia-userdebug
     13. mini_emulator_arm64-userdebug
     14. m_e_arm-userdebug
     15. mini_emulator_mips-userdebug
     16. mini_emulator_x86_64-userdebug
     17. mini_emulator_x86-userdebug
     18. aosp_flounder-userdebug
     19. aosp_hammerhead-userdebug
     20. aosp_mako-userdebug
     21. aosp_shamu-userdebug
     22. aosp_manta-userdebug
     23. example_product-eng
     24. example_product-userdebug
     25. example_product-user

Which would you like? [aosp_arm-eng]]
```

Izlaz lunch komande je lista svih definisanih konfiguracija, i u sledećem koraku treba navesti redni broj željene konfiguracije. Drugi način izbora konfiguracije je poziv lunch komande sa nazivom željene konfiguracije.

```
rtrk@rtrk:~/android-5.1.1_r9$ lunch example_product-eng

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=5.1.1
TARGET_PRODUCT=example_product
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=x86
TARGET_ARCH_VARIANT=x86
TARGET_CPU_VARIANT=
TARGET_2ND_ARCH=
TARGET_2ND_ARCH_VARIANT=
TARGET_2ND_CPU_VARIANT=
```

```
HOST_ARCH=x86_64
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.5.0-37-generic-x86_64-with-Ubuntu-12.10-quantal
HOST_BUILD_TYPE=release
BUILD_ID=LMY48Z
OUT_DIR=out
============================================
```

Nakon uspešnog izbora konfiguracije prikazuje se kratak opis izabrane konfiguracije. Izabrana konfiguracija aktuelna je samo u trenutnoj seansi terminala, svakim pokretanjem terminala treba ponovo izvršiti odabir željene konfiguracije.

Izaberite example_product-eng konfiguraciju i analizirajte izlaz lunch komande. Uporedite skup i vrednosti sistemskih promenljivih pre i nakon izvršenja lunch komande. Skup sistemskih promenljivih dobija se komandom printenv.

Prevođenje

Sistem za prevođenje AOSP programskog rešenja oslanja se na GNU Make alat. Za prevođenje koristi se make komanda koja se izvršava iz AOSP direktorijuma.

```
rtrk@rtrk:~/android-5.1.1_r9$ make
```

GNU Make alat podržava paralelizaciju, odnosno može da prevodi nekoliko različitih modula istovremeno. Uz pomoć paralelizacije značajno se skraćuje vreme potrebno za prevođenje. Prevođenje sa paralelizacijom pokreće se na sledeći način:

```
rtrk@rtrk:~/android-5.1.1_r9$ make -jN
```

N predstavlja faktor paralelizacije, odnosno broj modula koji će se istovremeno prevoditi. Optimalno vreme prevođenja dobija se kada je C < N < 2C, gde je C broj jezgara centralne procesorske jedinice. Tačnu vrednost faktora optimizacije potrebno je eksperimentalno utvrditi, jer pored centralne procesorske jedinice zavisi od drugih parametara, npr. količina i brzina radne memorije. Broj jezgara centralne procesorske jedinice može se utvrditi sledećom komandom:

```
rtrk@rtrk:~/android-5.1.1_r9$ cat /proc/cpuinfo
```

Izvršite prevođenje AOSP programskog rešenja za example_product-eng konfiguraciju. U ovom slučaju kao rezultat prevođenja dobićete sledeću poruku "make: Nothing to be done for `droid'". Sistem za prevođenje vas obaveštava da nema novih izmena koje treba prevesti, jer je programsko rešenje već prevedeno u celosti.

Rezultat prevođenja smešten je u out direktorijum. Sistem za prevođenje razvrstava rezultate prevođenja različitih konfiguracija po direktorijumima, tako da je moguće raditi na nekoliko konfiguracija istovremeno. Datoteke dobijene prevođenjem example_product-eng konfiguracije nalaze se u out/target/product/example_product/ direktorijumu.

Pogledajte i analizirajte šta je dobijeno kao rezultat prevođenja example_product-eng konfiguracije.

Emulator

Sistemska slika dobijena prevođenjem (out/target/product/example_product/system.img) može biti pokrenuta na emulatoru. Emulator predstavlja okruženje izuzetno pogodno za učenje i zbog toga će se u daljem radu koristiti emulator kao odredišni uređaj. Emulator se pokreće sledećom komandom:

```
rtrk@rtrk:~/android-5.1.1_r9$ emulator
```

Uz komandu emulator može se upotrebiti mnoštvo parametara, za dobijanje kompletne liste parametara izvršiti komandu sa -help.

Pokrenite emulator za example_product-eng konfiguraciju. Treba da dobijete prozor sa funkcionalnim Android operativnim sistemom.

Pristup emulatoru ostvaruje se uz pomoć **adb** (Android Debug Bridge) alata, na sledeći način:

```
rtrk@rtrk:~/android-5.1.1_r9$ adb shell
root@android:/ #
```

Otvorite novi terminal i pristupite prethodno pokrenutom emulatoru. Primetite da adb alat omogućava izvršenje komandi na emulatoru.

[3] Pretraga koda

Očekivani rezultati ove vežbe su:

- Upoznavanja i upoređivanje vremena pretrage na lokalnom računaru i xref server-u

Tokom rada se AOSP kodom često je potrebno pronaći određene datoteke ili sadržaj u njima. Pretrage se mogu podeliti na one koje se izvršavaju na lokalnom računaru i one koje se izvršavaju na udaljenom server-u.

Ukoliko se pretraga izvršava na lokalnom računaru tada je moguće koristiti cgrep, jgrep i rgrep funkcije za pregracu c/c++ koda, java koda i xml datoteka. Takođe moguće je koristi standardnu komande u linux-u: find i grep.

Pokušajte da sa find, grep i cgrep komandama pronađete gde se nalazi implementacija MediaPlayerFactory klase:

```
rtrk@rtrk:~/android-5.1.1_r9$ find –iname “*.cpp” | xargs grep
–s “MediaPlayerFactory”
...

rtrk@rtrk:~/android-5.1.1_r9$ grep –r “MediaPlayerFactory” .
...

rtrk@rtrk:~/android-5.1.1_r9$ cgrep “MediaPlayerFactory”
...
```

Pokrenite komande bar 2 puta. Koja komanda najbrže da rezultat?

Kao što se verovatno primetili pretraga na lokalnom računaru, čak i kada se koriste optimizovane funkcije iz AOSP koda, zahteva mnogo vremena. Drugi način je pretraga koda preko servera kao što su:

- http://xref.opersys.com/
- http://androidxref.com/

Ovi serveri, pored toga što izbacuju rezultate u ms, pružaju mogućnost pretrage različitih verzija AOSP koda. Šta više, pomoću http://aosp.opersys.com/changelog/ moguće je uporediti dve različite verzije AOSP koda i na taj način pogledati koje izmene se nalaze u novijoj verziji. Ovakav način pretrage je karakterističan za Linux sistem, odakle je i preuzet.

Pronađite MediaPlayerFactory klasu pomoću http://xref.opersys.com/ stranice na verziji Android-a 5.1.1 r9

Uporedite vreme pretrage na lokalnom računaru i na serveru. Nikada više nemojte pretraživati AOSP kod preko lokalnog računara. ☺

[4] Softverska konfiguracija

Očekivani rezultati ove vežbe su:

- Izvorni kod koji će izvršiti zamenu podrazumevane pozadine na emulator-u kao i potvrdu da je to uspešno urađeno (snimak ekrana).

Iz prethodnog poglavlja moglo se videti da prevođenju AOSP programskog rešenja obavezno prethodi izbor konfiguracije. Uz AOSP dolazi nekoliko konfiguracija koje se mogu upotrebiti, ali moguće je kreirati novu konfiguraciju. Za potrebe ovog kursa kreirana je example_product-eng konfiguracija, čiji se opis nalazi u vendor/rtrk/example_product/ direktorijumu. Uobičajena praksa je da programski moduli specifični za datu konfiguraciju budu smešteni u direktorijum konfiguracije. Od sada vendor/rtrk/example_product/ će se smatrati radnim direktorijumom, odnosno sve nove module treba smestiti u direktorijum example_product-eng konfiguracije.

Analizirajte sadržaj vendor/rtrk/example_product/ direktorijuma.

Osnovni opis example_product-eng konfiguracije nalazi se u example_product.mk datoteci. Konfiguraciona datoteka može da sadrži naziv konfiguracije, skup programskih modula koji ulaze u sastav konfiguracije, opis lokalizacije, putanju do resursnih datoteka, itd.

Analizirajte sadržaj vendor/rtrk/example_product/example_product.mk datoteke. Obratite pažnju da je postojeću konfiguraciju moguće "naslediti", odnosno iskoristiti za kreiranje nove konfiguracije.

U fazi inicijalizacije sistem za prevođenje pokušava da pronađe sve vendorsetup.sh datoteke, koje sadrže deklaracije softverskih konfiguracija. Deklaracija example_product-eng konfiguracije nalazi se u vendor/rtrk/vendorsetup.sh datoteci. Pomenuto pretraživanje odvija se u device i vendor direktorijumima, odnosno samo u dva navedena direktorijuma moguće je deklarisati konfiguraciju.

Analizirajte sadržaj vendor/rtrk/vendorsetup.sh datoteke.

U fazi izbora konfiguracije sistem za prevođenje traži konfiguracionu datoteku željene konfiguracije. Lokacija konfiguracione datoteke example_product-eng konfiguracije definisana je vendor/rtrk/AndroidProducts.mk datotekom.

Analizirajte sadržaj vendor/rtrk/AndroidProducts.mk datoteke.

AOSP sistem za prevođenje pruža mogućnost da se izvrši izmena resursnih datoteka na nivou konfiguracije, odnosno moguće je obezbediti skup resursnih datoteka koji će biti primenjen samo na zadatu konfiguraciju. Putanja do resursnih datoteka za konfiguraciju definiše se PRODUCT_PACKAGE_OVERLAYS promenljivom, za example_product-eng konfiguraciju ima sledeću vrednost vendor/rtrk/example_product/overlay. Novi resursi moraju imati istu putanju u sistemu direktorijuma kao originalni

resursi, u odnosu na zadati direktorijum. Na primer, zamena za konfiguracionu datoteku frameworks/base/core/res/res/values/config.xml je vendor/rtrk/example_product/overlay/frameworks/base/core/res/res/values/config.xml.

Zamenite podrazumevanu pozadinu frameworks/base/core/res/res/drawable-nodpi/-default_wallpaper.jpg sa example_default_wallpaper.jpg za example_product-eng konfiguraciju. Napravite novu sliku emulatora sa make komandom i pokretanjem emulatora proverite da li je pozadina promenjena.

[5] AOSP modul

Očekivani rezultati ove vežbe su:

- Izvorni kod izvršnog programa kao i potvrdu da ovaj program ispravno radi (snimak ekrana).
- Izvorni kod izvršnog programa i statičke biblioteke kao i potvrdu da ovaj program ispravno radi (snimak ekrana).
- Izvorni kod izvršnog programa i dinamičke biblioteke kao i potvrdu da ovaj program ispravno radi (snimak ekrana).

AOSP modul predstavlja minimalnu jedinicu prevođenja i definiše se uz pomoć Android.mk datoteke. Jedna Android.mk datoteka može sadržati više modula. Modul je ograničen instrukcijom za brisanje okruženja i instrukcijom za prevođenje, na primer:

```
include $(CLEAR_VARS)
      .
      .
      .
include $(BUILD_EXECUTABLE)
```

Napravite novi modul za generisanje izvršne datoteke pod nazivom helloworld. Modul treba da prevodi sledeći programski kod:

```
#include <stdio.h>

int main()
{
    printf("Hello world\n");
    return 0;
}
```

PODSETNIK: LOCAL_MODULE - naziv modula, LOCAL_SRC_FILES - datoteke koje se prevode, include $(BUILD_EXECUTABLE) - instrukcija za generisanje izvršne datoteke. Za prevođenje novog modula možete upotrebiti mm ili mmm komande. Za generisanje sistemske slike koja sadrži izvršnu datoteku upotrebiti make snod komandu.

Između modula može postojati zavisnost, dobar primer je odnos izvršne datoteke i programske biblioteke. Zavisnost od statičkih biblioteka definiše se LOCAL_STATIC_LIBRARIES promenljivom, a zavisnost od dinamičkih biblioteka LOCAL_SHARED_LIBRARIES promenljivom u opisu modula.

Napravite novi modul za generisanje statičke programske biblioteke "libhello.a", koja se koristi od strane helloworld izvršne datoteke. Modul treba da prevodi sledeći programski kod:

```
#include "libhello.h"
#include <stdio.h>

void hello_static()
{
    printf("Hello static library\n");
}
```

PODSETNIK:LOCAL_C_INCLUDES - zavisnost od zaglavlja, include $(BUILD_STATIC_LIBRARY) - instrukcija za generisanje statičke biblioteke.

Napravite novi modul za generisanje dinamičke programske biblioteke "libhello.so", koja se koristi od strane helloworld izvršne datoteke. Modul treba da prevodi sledeći programski kod:

```
#include "libhello.h"
#include <stdio.h>

void hello_shared
{
printf("Hello shared library\n");
}
```

PODSETNIK: include $(BUILD_SHARED_LIBRARY) - instrukcija za generisanje dinamičke biblioteke.

[6] Uslovno prevođenje

Očekivani rezultati ove vežbe su:

- [opciono] Izvorni kod za uslovno prevođenje izvršnog programa preko pretprocesorske direktive definisane u Android.mk datoteci kao i potvrdu da uslovno prevođenje uspešno funkcioniše (snimak ekrana)
- [opciono] Izvorni kod za uslovno prevođenje izvršnog programa preko promenljivih kao i potvrdu da uslovno prevođenje uspešno funcioniše (snimak ekrana).

Vrlo često pojedine delove programskog rešenja nije potrebno prevoditi u zavisnosti od željene konfiguracije. Da bi programsko rešenje bilo konfigurabilno programski jezik mora podržavati uslovno prevođenje. Programski jezik C/C++ pruža mogućnost uslovnog prevođenja posredstvom pretprocesorskih direktiva #if, #ifdef, #ifndef, itd. Sledeći primer pokazuje slučaj kada prevođenje programske funckije hello_world zavisi od CONF_HELLO_WORLD makroa. Prevođenje date funkcije biće izvršeno samo u slučaju kada je definisan CONF_HELLO_WORLD makro.

```
#define CONF_HELLO_WORLD

#ifdef CONF_HELLO_WORLD
void hello_world()
{
printf("Hello world\n");
}
#endif
```

Napravite AOSP modul koji generiše izvršnu datoteku exampleandroid. Modul treba da prevodi sledeći programski kod:

```
#include <stdio.h>

int main()
{

#ifdef EXAMPLE_ANDROID
printf("EXAMPLE_ANDROID is defined\n");
#else
printf("EXAMPLE_ANDROID is not defined\n");
#endif

return 0;
}
```

Pokretanje exampleandroid izvršne datoteke treba da rezultira "EXAMPLE_ANDROID is not defined" ispisom, jer EXAMPLE_ANDROID makro nije definisan.

Pored #define pretprocesorske direktive makro je moguće definisati parametrima programskog prevodioca. U slučaju AOSP modula parametri C programskog prevodioca definišu se LOCAL_CFLAGS promenljivom. Uz pomoć parametara prevodioca ostvaruje se konfigurabilnost programskog rešenja na nivou modula.

Proširite prethodno definisani AOSP modul za generisanje exampleandroid izvršne datoteke sa parametrom prevodioca koji definiše EXAMPLE_ANDROID makro.

```
LOCAL_CFLAGS += -DEXAMPLE_ANDROID
```

Ako ste ispravno definisali EXAMPLE_ANDROID makro dobićete "EXAMPLE_ANDROID is defined" ispis kao rezultat rada exampleandroid izvršne datoteke.

AOSP sistem za prevođenje programskog rešenja baziran je na GNU Make alatu, koji dozvoljava pisanje uslovnog recepta. Uslovni recepti realizuju se uz pomoć ifeq, ifneq, ifdef, ifndef, else i endif direktiva. Uslovnim receptima moguće je uticati na konfiguraciju modula, isključiti pojedine datoteke ili cele module iz procesa prevođenja. Sledeći primer pokazuje slučaj kada se foo.c datoteka prevodi u zavisnosti od USE_FOO promenljive.

```
include $(CLEAR_VARS)
    .
    .
ifeq ($(USE_FOO), YES)
LOCAL_SRC_FILES += foo.c
endif
    .
    .
include $(BUILD_EXECUTABLE)
```

Pisanje uslovnog recepta za prevođenje pruža mogućnost konfigurabilnosti programskog rešenja na nivou AOSP konfiguracije. Ako je USE_FOO promenljiva definisana AOSP konfiguracijom, svi uslovni recepti koji koriste datu promenljivu biće pogođeni promenom njene vrednosti.

Izmenite prethodno definisani AOSP modul za generisanje exampleandroid izvršne datoteke, tako da EXAMPLE_ANDROID makro bude definisan samo kada je vrednost USE_EXAMPLE_ANDROID promenljive jednaka YES. Promenljivu USE_EXAMPLE_ANDROID definisati u okviru example_product-eng konfiguracije.

[7] Otklanjanje programskih grešaka

Očekivani rezultati ove vežbe su:

- Izvorni kod izvršnog programa koja ispisuje poruke sa HelloLog oznakom kao i potvrdu da ovaj program ispravno funkcioniše (logcat).
- Izvorni kod callstack izvršnog programa koji ispisuje adrese callstack-a u logcat kao i potvrdu to program ispravno funkcioniše (logcat). Potvrdu da je addr2line alat uspešno preveo adrese iz callstack izvršnog programa (snimak ekrana).
- Potvrdu da je GDB server i klijent uspešno povezan, odnosno da je GDB alat uspešno iskorišćen za otkrivanje greške u izvršnog programu (snimci ekrana).
- [opciono] Tekstualni opis uzroka ANR greške.

Otklanjanje programskih grešaka je proces koji uključuje jasnu identifikaciju problema, pronalaženje uzroka za dati problem i odgovarajuće korekcije programskog rešenja. Ovim poglavljem opisane su tehnike i alati za određivanje uzroka problema. Date tehnike i alati mogu poslužiti za bolje upoznavanje i analizu implementacije postojećih AOSP modula.

Android Logcat

Jedna od najčešće korišćenih tehnika u otklanjanju grešaka je analiza programskih zapisa (eng. *log*). Zapisi daju stalan uvid u stanje programskog rešenja, redosled izvršavanja, greške i razloge grešaka. AOSP poseduje sistem za prikupljanje i analizu programskih zapisa. Zapisi se skladište u kružnim baferima, koje je naknadno moguće pregledati i filtrirati uz pomoć adb alata i logcat instrukcije.

```
rtrk@rtrk:~/android-5.1.1_r9$ adb logcat
I/SystemServer(  154): Wi-Fi P2pService
W/MountService(  154): getVolumeState(/mnt/sdcard): Unknown volume
W/MountService(  154): getSecureContainerList() called when storage not
mounted
```

Podrazumevani format zapisa sadrži prioritet (I, W, D, ...), oznaku (eng. *tag*), identifikator procesa operativnog sistema (PID) i tekst zapisa, gledano sa leva na desno. Oznaka (*SystemServer, MountService*) i prioritet zapisa koriste se za filtriranje, odnosno izdvajanje zapisa za koje postoji interesovanje. Na primer za analizu svih zapisa prioriteta većeg ili jednakog prioritetu greške upotrebljava se sledeća instrukcija:

```
rtrk@rtrk:~/android-5.1.1_r9$ adb logcat *:E
```

Za analizu zapisa sa oznakom "Trace", prioriteta većeg ili jednakog prioritetu greške koristi se sledeća intrukcija:

```
rtrk@rtrk:~/android-5.1.1_r9$ adb logcat -s Trace:E
```

Analizirajte sve zapise sa oznakom "ActivityManager".

Kreiranje zapisa omogućeno je na nivou C/C++ i Java programskih jezika. Za Java programski jezik definisana je klasa Log,a u slučaju programskog jezika C/C++ koriste se makroi za kreiranje zapisa (ALOGD, ALOGW, ALOGE, ...). Makroi za kreiranje zapisa definisani su cutils/log.h zaglavljem, a oznak u zapisa definiše LOG_TAG makro. Za kreiranje zapisa neophodno je uključiti libcutils programsku biblioteku. Sledeći primer ilustruje kreiranje zapisa na nivou C/C++ programskog jezika "Debug" prioriteta sa oznakom "Example". Obratiti pažnju da "#define LOG_TAG" direktiva mora prethoditi "#include <cutils/log.h>" direktivi.

```
#define LOG_TAG "Example"
#include <cutils/log.h>

ALOGD("Example log");
```

Makroi za kreiranje zapisa koriste formatiranje "printf" programske funkcije, odnosno kreiranje zapisa je slično pisanju na standardni izlaz u programskom jeziku C.

Napravite AOSP modul koji generiše izvršnu datoteku hellolog. Modul treba da prevodi sledeći programski kod:

```
#define LOG_TAG "HelloLog"
#include <cutils/log.h>

int main()
{
    ALOGD("Hello world");
    return 0;
}
```

Analizirajte zapise za oznakom HelloLog.

CallStack

Vrlo često postoji potreba za analizom toka programskog rešenja. Analiza toka može da se izvrši uz pomoć prethodno pomenutih zapisa. AOSP nudi gotovo rešenje za analizu toka u vidu programske klase CallStack i addr2line alata. CallStack klasom mogu da se kreiraju zapisi sa programskim adresama funkcija poređani po hronološkom redosledu poziva. Na osnovu dobijenih adresa dobijaju se lokacije u programskom kodu (datoteka:linija koda) uz pomoć addr2line alata.

Sledećim primerom definisan je način upotrebe CallStack programske klase, koja je deklarisana utils/CallStack.h zaglavljem.

Ukoliko se CallStack klasa poziva iz Jelly Bean verzije Andorid-a koristiti dump funkciju za ispis poruka u logcat. Filter u ovom slušaju treba postaviti na oznaku "CallStack".

Ukoliko se CallStack klase poziva iz novije verzije Android-a koristi log("CallStackExample") funkciju za ispis poruka. "CallStackExample" predstavlja oznaku zapisa koji se dobijaju pozivom log metode.

```
#include <utils/CallStack.h>

CallStack stack;
stack.update();
//! If you are using Jelly Bean use dump function. For higher versions use log
stack.dump();
//stack.log("CallStackExample");
```

Analizom zapisa dobija se uvid u redosled adresa poziva programskih funkcija.

```
rtrk@rtrk:~/android-5.1.1_r9$ adb logcat -s CallStackExample CallStack
D/CallStackExample(  133): #00 pc 0000063f /system/bin/callstack_example
D/CallStackExample(  133): #01 pc 00000605 /system/bin/callstack_example
D/CallStackExample(  133): #02 pc 000005f8 /system/bin/callstack_example
D/CallStackExample(  133): #03 pc 000005eb /system/bin/callstack_example
D/CallStackExample(  133): #04 pc 000005de /system/bin/callstack_example
D/CallStackExample(  133): #05 pc 000005cc /system/bin/callstack_example
D/CallStackExample(  133): #06 pc 0001704c /system/lib/libc.so(__libc_init+99)
D/CallStackExample(  133): #07 pc 00000512 /system/bin/callstack_example
```

Odgovarajuće lokacije u programskom kodu dobijaju se sledećom komandom.

```
rtrk@rtrk:~/android-5.1.1_r9$ addr2line -psa -e
out/target/product/example_product/symbols/system/bin/callstack_example
0000063f 00000605 000005f8 000005eb 000005de 000005cc

0x0000063f: callstack_example.cpp:38
0x00000605: callstack_example.cpp:33
0x000005f8: callstack_example.cpp:29
0x000005eb: callstack_example.cpp:25
0x000005de: callstack_example.cpp:21
0x000005cc: callstack_example.cpp:16
```

Analizirati tok izvršenja callstack_example primera (resursi/callstack_example) za funkciju function5(), uz pomoć CallStack klase i addr2line alata.

GDB (The GNU Project Debugger)

Napredni alat za otklanjanje programskih grešaka je debugger. Debugger pruža mogućnost kontrolisanog izvršenja programskog rešenja, zaustavljanje u željenoj tački, ispitivanje stanja u trenu zaustavljanja, eksperimentalna izmena stanja programskog rešenja itd. Uz AOSP dolazi GDB, debugger rešenje otvorenog programskog koda, sa podrškom za rad na udaljenom uređaju.

Upotrebi GDB debuggera prethodi sledećih 6 koraka:

1. Prevođenje programskog rešenja sa simbolima i isključenim optimizacijama programskog prevodioca

```
LOCAL_CFLAGS += -g -O0 -fno-inline    // programski jezik C
LOCAL_CPPFLAGS += -g -O0 -fno-inline  // programski jezik C++
```

2. Pokretanje GDB servera na razvojnom uređaju (emulator)

```
root@android:/ # gdbserver :5039 /system/bin/executable
```

3. Preusmeravanje sadržaja sa lokalnog porta 5039 na port 5039 razvojnog uređaja (emulator)

```
rtrk@rtrk:~/android-5.1.1_r9$ adb forward tcp:5039 tcp:5039
```

4. Pokretanje GDB klijenta

```
rtrk@rtrk:~/android-5.1.1_r9$ x86_64-linux-android-gdb
out/target/product/example_product/symbols/system/bin/executable
GNU gdb (GDB) 7.1-android-gg2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-linux-gnu --target=arm-elf-
linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

5. Podešavanje putanja do ostalih simbola

```
(gdb) set solib-absolute-prefix /home/rtrk/android-
5.1.1_r9/out/target/product/example_product/symbols
(gdb) set solib-search-path /home/rtrk/android-
5.1.1_r9/out/target/product/example_product/symbols/system/lib
```

6. Povezivanje sa GDB serverom

```
(gdb) target remote:5039
```

Uz pomoć GDB alata otklonite programske greške u okviru resursi/gdbexample primera.

[opciono] ANR (Application Not Responding)

Osnovni problem implementacije korisničke sprege je zadovoljenje kriterijuma odgovarajućeg odziva na korisničke unose. AOSP programsko rešenje definiše 5 sekundi kao maksimalno vreme odziva pri korisničkoj interakciji. U slučaju kada se ne zadovolji gore pomenuti kriterijum generiše se ANR greška, koja rezultira prekidom izvršenja trenutno aktivne aplikacije. Otkrivanje uzroka za ANR grešku može biti problematično uz prethodno opisane tehnike, zbog čega je uveden am monitor alat specijalizovan za otkrivanje uzroka ANR greške.

```
root@android:/ # am monitor
Monitoring activity manager... available commands:
(q)uit: finish monitoring
** Activity starging: rtrk.pnrs.anr
** ERROR: EARLY PROCESS NOT RESPONDING
processName: rtrk.pnrs.anr
```

Rezultat rada am monitor alata je datoteka sa stanjem aktivinih procesa operativnog sistema u trenutku generisanja ANR greške. Pomenuta datoteka nalazi se na sledećoj lokaciji /data/anr/traces.txt.

Uz pomoć am monitor alata pronađite uzrok nastanka ANR greške u primeru resursi/anr_example.

[8] Sistemski koncepti

Očekivani rezultati ove vežbe su:

- Izvorni kod izvršnog programa koji dobavlja i postavlja rtrk.android svojstvo kao i potvrdu da program ispravno radi (snimci ekrana).
- [opciono] Izvorni kod Android aplikacije koja dobavlja i postavlja rtrk.android svojstvo kao i potvrdu da program ispravno radi (snimci ekrana).
- Izvorni kod exd izvršnog programa kao i potvrdu da se on ispravno prevodi.
- Izvorni kod init.goldfish.rc datoteke kao i potvrdu da se exd izvršno program uspešno pokreće iz init procesa (logcat ili snimci ekrana).
- Izvorni kod binder servera i klijenta pisanih u C/C++ programskom jeziku kao i potvrdu da primer ispravno funkcioniše (logcat ili snimci ekrana)
- Izvorni kod binder servera pisanog u C/C++ i binder klijenta pisanog u Java programskom jeziku koji razmenjuju osnovne tipove podatak kao i potvrdu da primer ispravno funkcioniše (logcat ili snimci ekrana)
- [opciono] Izvorni kod binder servera pisanog u C/C++ i binder klijenta pisanog u Java programskom jeziku koji razmenjuju složene tipove podatak kao i potvrdu da primer ispravno funkcioniše (logcat ili snimci ekrana)
- [opciono] Izvorni kod binder servera pisanog u C/C++ i binder klijenta pisanog u Java programskom jeziku koji razmenjuju callback pozive kao i potvrdu da primer ispravno funkcioniše (logcat ili snimci ekrana)
- Izvorni kod ashmem C/C++ server i C/C++ klijent izvršnog programa kao i potvrdu da program ispravno funkcioniše (logcat ili snimci ekrana).
- [opciono] Izvorni kod ashmem C/C++ server i Java klijent izvršnog programa kao i potvrdu da program ispravno funkcioniše (logcat ili snimci ekrana).
- Izvorni kod dinamičke biblioteke i Android JNI aplikacije kao i potvrdu da aplikacija ispravno funkcioniše (logcat ili snimci ekrana).
- [opciono] Izvorni kod Android java aplikacije, JNI sprege i native servisa koji se koriste za zauzimanje ashmem regiona, dobavljanje file descriptora preko refleksije u JNI-u i slanje istog preko binder sprege u native servis kao i potvrdu da sve navedeno ispravno funkcioniše (logcat ili snimci ekrana).

Android svojstva

Android svojstva predstavljaju skup vrednosti koje opisuju konfiguraciju i trenutno stanje Android operativnog sistema. Svojstva su definisana nazivom i vrednošću svojstva, npr:

```
[ro.product.brand]: [Android]
[ro.product.cpu.abi2]: [armeabi]
[ro.product.cpu.abi]: [armeabi-v7a]
[ro.product.device]: [example_product]
```

Uvid u Android svojstva ostvaruje se getprop komandom, koja u osnovnom obliku ispisuje vrednosti svih svojstava.

```
root@android:/ # getprop
[ARGH]: [ARGH]
[dalvik.vm.heapsize]: [48m]
[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
[init.svc.adbd]: [running]
[init.svc.bootanim]: [running]
[init.svc.console]: [running]
...
```

Pored uvida u sva svojstva moguće je analizirati vrednosti pojedinačnih svojstava navodeći naziv svojstva pri pozivu getprop komande.

```
root@android:/ # getprop ro.product.name
```

example_productVrednosti svojstava mogu se menjati ili dodavati setprop komandom, koja kao parametre prihvata naziv i vrednost parametra.

```
root@android:/ # setprop rtrk.android RTRK
```

Definišite rtrk.android Android svojstvo sa vrednošću RTRK i proverite vrednost novog svojstva.

Pristup Android svojstvima moguć je iz programskih rešenja. Za programski jezik C/C++ definisana je programska sprega cutils/properties.h zaglavljem i libcutils programskom bibliotekom. Programska sprega za pristup svojstvima definisana je na sledeći način:

```
int property_set(const char *key, const char *value)
int property_get(const char *key, char *value, const char *default_value)
```

Napravite program koji proverava vrednost rtrk.android svojstva. Program treba da postavi vrednost rtrk.android svojstva na RTRK, kada se postojeća vrednost razlikuje od date vrednosti.

Za potrebe programskog jezika Java definisana je android.os.SystemProperties klasa koja pruža funkcionalnosti za pristup Android svojstvima. Pomenuta klasa poseduje metode za pisanje vrednosti i čitanje sa odgovarajućom konverzijom u osnovne tipove podataka.

[opciono] Uz pomoć resursi/PropertyExample Android aplikacije i android.os.SystemProperties programske klase ispitati vrednost rtrk.android svojstva i podesiti vrednost na RTRK, ako se trenutna vrednost razlikuje od date vrednosti. **Napomena:** PropertyExample je Android aplikacija bez funkcionalnosti. Sva proširenja treba dodati u onCreate() metodu MainActivity klase, jer onCreate() metodu možete smatrati ulaznom tačkom programa.

Prilikom pokretanja emulatora isključite selinux ili u saradnji sa asistentom definišite selinux prava za PropertyExample Android aplikaciju.

```
emulator  -selinux permissive
```

Servis

Servis je vid programske podrške kojim se ne ostvaruje direktna interakcija sa korisnikom. U opštem slučaju servis pruža usluge drugim procesima operativnog sistema. Osnovna odlika servisa je aktivnost tokom celokupnog vremena rada operativnog sistema (*daemon*). Naziv servisa po konvenciji završava se slovom *d* (npr. vold - Volume Daemon, dhcpcd - DHCP Client Daemon). Implementacija servisa zasniva se na beskonačnoj programskoj petlji u okviru koje je definisana odgovarajuća obrada.

Implementirajte Android modul koji generiše exd (Example Daemon) izvršnu datoteku na osnovu sledećeg programskog koda.

```
#define LOG_TAG "EXD"
#include <cutils/log.h>
#include <unistd.h>

int main()
{
    while(1)
    {
        ALOGD("TICK");
        sleep(5); //seconds
    }
    return 0;
}
```

Example Daemon predstavlja trivijalnu implementaciju servisa, čiji je jedini zadatak kreiranje zapisa u svrhu provere aktivnosti servisa.

Init

Init proces je prvi proces koji se pokreće sa Android operativnim sistemom i svi ostali procesi se pokreću iz Init procesa. Init proces je zadužen za inicijalizaciju sistema i pokretanje servisa. Rad Init procesa definisan je *.rc skriptama, gde je init.rc primarna skripta. Pored init.rc skripte koja je definisana na nivou AOSP programskog rešenja, moguće je definisati skriptu specifičnu za odabranu programsku konfiguraciju (init.${ro.hardware}.rc), gde je ro.hardware Android svojstvo definisano konfiguracijom. U slučaju Android emulatora ro.hardware ima vrednost goldfish, a skripta se nalazi na sledećoj lokaciji system/core/rootdir/etc/init.goldfish.rc. Dobra praksa je da se izmene init skripte vrše u skripti specifičnoj za konfiguraciju, da bi se očuvala kompatibilnost na nivou različitih konfiguracija.

Sadržaj init skripte opisan je Android init skript jezikom, koji je baziran na događajima ili okidačima. Detaljan skup pravila za pisanje init skripte nalazi se u system/core/init/readme.txt datoteci. U nastavku teksta dat je primer definicije Android svojstva init skriptom nakon boot događaja (generiše se neposredno nakon dostizanja aktivnog stanja operativnog sistema).

```
on boot
```

```
    setprop rtrk.android RTRK
```

Proširite init.goldfish.rc skriptu tako da se definiše rtrk.android Android svojstvo sa vrednošću RTRK. Proveriti vrednost svojstva nakon pokretanja emulatora. **Napomena:** nakon izmene init.goldfish.rc potrebno je pokrenuti prevođenje celokupnog AOSP programskog rešenja, da bi izmena bila uvrštena u sistemsku sliku.

Sledeći primer ilustruje definiciju exd servisa i njegovo pokretanje na boot događaj. Pri definiciji servisa potrebno je navesti naziv servisa i apsolutnu putanju do izvršne datoteke u sistemu direktorijuma. Servis je moguće kreirati sa različitim modifikatorima. U datom primeru koristi se oneshot čime se definiše da servis ne treba pokretati nakon prestanka rada servisa.

```
service exd /system/bin/exd
    oneshot

on boot
    start exd
```

Proširite init.goldfish.rc skriptu tako da se exd servis pokreće sa boot događajem. Proveriti da li je exd servis uspešno pokrenut.

Binder

Binder je mehanizam za međuprocesnu komunikaciju Android operativnog sistema (IPC - eng. *Inter Process Communication*). Implementacija Binder programske sprege bazirana je na razmeni poruka, organizovana u vidu klijent-server modela. Pomenuta organizacija Binder programske sprege pruža mogućnost komunikacije procesa operativnog sistema realizovanih na različitim nivoima apstrakcije, različitim programskim jezicima: C/C++ - C/C++, C/C++ - Java, Java - C/C++ i Java - Java.

U praksi detalji implementacije razmene poruka skriveni su od korisnika, odnosno implementacija Binder komunikacije podrazumeva definiciju programske sprege, uz poštovanje pravila već definisanog radnog okruženja. Komunikacija se zasniva na dobavljanju instance programske klase kojom je definisana programska sprege (Binder objekat) i pozivanju odgovarajućih metoda. AOSP definiše Service menadžer kao centralnu tačku sistema, odnosno servis koristi Service menadžer da registruje svoj Binder objekat, odakle će ga klijentski procesi preuzeti i koristiti za komunikaciju sa servisom. U nastavku vežbe biće prikazana dva primera upotrebe binder-a:

- Primer komunikacije između dva nezavisna programa (procesa) napisana u C/C++ jeziku
- Primer komunikacije između dva nezavisna programa (procesa) od kojih je servis napisan u C/C++ u klijent u Java jezika

Binder C/C++ - C/C++

Programska sprega za implementaciju Binder komunikacije na ovom nivou apstrakcije definisana je libbinder programskom bibliotekom i programskim zaglavljima "binder/IInterface.h" i "binder/Parcel.h".

Početni korak implementacije Binder komunikacije je deklaracija programske sprege servisa. Programska sprega servisa deklariše se nasleđivanjem IInterface klase i navođenjem apstraktnih metoda. U primeru dole deklarisana je programska sprega za metodu getExample. Pored navedene medote obavezna je enumeracija sa jedinstvenim identifikatorima svih metoda. Data enumeracija koristiće se kasnije za identifikaciju metoda pri Binder komunikaciji.

```
class IExample : public IInterface
{
public:
    enum
    {
        GET_EXAMPLE = IBinder::FIRST_CALL_TRANSACTION
    };

    virtual int32_t getExample() = 0;

    DECLARE_META_INTERFACE(Example);
};
```

Nakon deklaracije programske sprege servisa implementira se komunikacija na strani servisa nasleđivanjem BnInterface klase. Servis prima poruke uz pomoć onTransact metode, gde su code identifikator metode koja se poziva, data ulazni podaci i reply izlazni podaci jedne transakcije. Treba imati na umu da jedan poziv onTransact metode predstavlja jednu Binder sesiju, odnosno sastoji se od upita klijenta i odgovora servisa.

```
status_t BnExample::onTransact(uint32_t code, const Parcel& data,
        Parcel* reply, uint32_t flags __attribute__((unused)))
{
    data.checkInterface(this);

    switch(code)
    {
        case GET_EXAMPLE:
        {
            int32_t example = getExample();
            reply->writeInt32(example);
            break;
        }
    }

    return NO_ERROR;
}
```

Komunikacija na strani klijentskog procesa implementira se nasleđivanjem BpInterface klase i implementacije svih ranije navedenih metoda. Implementacija u BpInterface klasi podrazumeva samo

komunikaciju ne i stvarnu implementaciju metoda servisa. Sledeći primer ilustruje implementaciju getExample metode BpInterface klase za dati primer. U okviru getExample od najvećeg interesa je poziv transact metode kojim se inicira jedna binder sesija. Prvi parametar transact metode je identifikator metode servisa koju treba izvršiti, drugi i treći parametar su ulazni odnosno izlazni podaci sesije respektivno.

```
int32_t BpExample::getExample()
{
    Parcel data, reply;
    int32_t example;

    data.writeInterfaceToken(IExample::getInterfaceDescriptor());
    remote()->transact(GET_EXAMPLE, data, &reply);
    reply.readInt32(&example);

    return example;
}
```

Implementacija metoda u okviru servisa realizuje se nasleđivanjem klase koja je zadužena za komunikaciju na strani servisa, u datom primeru u pitanju je BnExample klasa.

```
class Example : BnExample
{
public:
    int32_t getExample();
};
```

Nakon implementacije programske sprege servisa sledi deljenje Binder objekta servisa sa ostatkom sistema uz pomoć Service menadžera. Sledeći primer ilustruje registraciju servisa.

```
int main(int argc, char** argv) {
    defaultServiceManager()->addService(String16("Example"), Example());
    android::ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
    return 0;
}
```

Pristup već registrovanom servisu podrazumeva dobavljanje Binder objekta datog servisa i pozivanje njegovih metoda, kao u sledećem primeru.

```
int main() {
      sp<IServiceManager> smanager = defaultServiceManager();
      sp<IBinder> binder = smanager->getService(String16("Example"));
      sp<IExample> example = interface_cast<IExample>(binder);
      int testValue = example->getExample();
      ALOGI("Test value: %d", testValue);
}
```

Proširite implementaciju exd servisa, iz prethodnog zadatka, IBinder spregom uz pomoć resursi/libbinderexample programske biblioteke. Data biblioteka sadrži kompletnu implementaciju

Binder komunikacije, odnosno ovim zadatkom treba implementirati serversku i klijentsku aplikaciju koje komuniciraju uz pomoć Binder mehanizma.

Proširite IExample programsku spregu servisa iz prethodnog zadatka setExample metodom "void setExample(int32_t value)". Data metoda treba da sačuva dobijenu vrednost u okviru servisa, tako da naredni pozivi getExample metode vraćaju sačuvanu vrednost.

Binder C/C++ - Java, osnovni tipovi

Na nivou Java programskog jezika Binder komunikacija realizuje se na sličan način. Radi lakše implementacije za Javu postoji alat za generisanje celokupne komunikacija, a na korisniku je samo da upotrebi dobijeno rešenje. Definisan je poseban programski jezik (AIDL - Androd Interface Definition Language) kojim se definiše programska sprega servisa. Datoteke koje sadrže AIDL programski kod imaju ekstenziju .aidl. Sledeći primer ilustruje jedan AIDL primer opisa programske sprege servisa.

```
package com.course.android;

interface IExample {
    int getExample();
    void setExample(int value);
}
```

Zbog toga što se Binder komunikacija odigrava razmenom poruka moguće je realizovati komunikaciju između procesa pisanih različitim programskim jezicima (npr. Java - C/C++). Da bi funkcionisala Java - C/C++ komunikacija potrebno je implementirati istu Binder programsku spregu sa oba programska jezika. U nastavku je dat primer pristupa servisu iz programskog jezika Java.

```
IBinder binder = (IBinder) ServiceManager.getService("Example");
IExample example = IExample.Stub.asInterface(binder);
try {
    int value = example.getExample();
    Log.d(TAG, "value = " + value);
} catch (RemoteException e) {
    Log.d(TAG, "getExample FAILED");
}
```

Preuzmite BinderExampleJavaClient iz resursa i Implementirajte testNativeService metodu MainActivity programske klase, tako da se u okviru nje pozove getExample metoda IExample servisa iz prethodnog zadatka.

Binder C/C++ - Java, složeni tipovi

Proširite prethodni primer tako da get funkcija umesto celobrojne vrednosti vrati složeni podatak koji sadrži u sebi jednu celobrojnu vrednost, jednu long vrednost i jedno tekstualno polje.

Obratite pažnju da native kod mora biti poravnat sa kodom koji je generisan u javi.

Pronađite na predavanjima putanju do AIDL među koda i proveriti kako se prenose vrednosti iz jave. Za prenos String-a u native-u koristite read-writeString16. Za prenos long vrednosti u native koristite read-writeInt64.

[opciono] Binder C/C++ - Java, callback

Proširite prethodni native servis – java klijent primer tako da se iz Java aplikacije registruje callback funkcija u native servisu i zatim da se periodično poziva.

Napraviti novu AIDL datoteku ICallback sa funkcijom

```
void callbackFunction();
```

Dodajte novu registerCallback funkciju u postojeću AIDL datoteku. Kao argument ova funkcija treba da prihvata ICallback.

U postojećem Android activity-u instancirajte serversku logiku (Stub) nad ICallback spregom i realizujte callback funkciju:

```
public ICallback callback = new ICallback.Stub() {
    @Override
    public void callbackFunction() throws RemoteException {
        Log.d(TAG, "[callbackFunction][enter]");
        Log.d(TAG, "[callbackFunction][exit]");
    }
};
```

Nakon uspešnog povezivanja na native servis pozovite register funkciju i prosledite callback objekat.

Na native strani napravite novu ICallback klasu koja nasleđuje IInterface po uzoru na već postojeći interface.

Realizujte proxy, klijentsku stranu ICallback sprege: BpCallback.

Realizujte register funkciju u okviru postojeće binder sprege native servisa. Prilikom dobavljanja callback objekta upotrebite readStrongBinder funkciju.

```
sp<ICallback> callback = interface_cast<ICallback>
(data.readStrongBinder());
```

Napravite pthread koji će ukoliko je callback postavljen na početnu vrednost pozivati isti na svaki sekund.

Ashmem

Binder kao najkorišćenija komponenta za razmenu poruka između procesa ima svoja ograničenja koja se ogledaju u količini podataka koje je moguće preneti (1MB) i broju procesa koji istovremeno mogu koristiti jedan binder (15).

U situacijama kada je potrebno podeliti više informacija između procesa moguće je koristiti Android deljenu memoriju, tj ashmem. Ashmem modul je dodat u Android Kernel iz razloga što za razliku od procesa, kernel nema ograničenje u pristupu memoriji, tj može da pristupi memoriji svih procesa. Ashmem arhitektura je orijentisana kao server-klijent, gde procesi (klijenti) komuniciraju sa Ashmem modulom (serverom).

Da bi dva procesa imala pristup istoj memorijskoj zoni, potrebno je da mapiraju memoriju sa istim file deskriptorom. To znači da je pre nego što se započne deljenje memorije potrebno poslati file deskriptor iz jednog procesa drugom pomoću bindera ili nekog drugom IPC mehanizma.

Ashmem C/C++ - C/C++ primer

Ideja ovog primera jeste da pokaže kako da se napravi ashmem region u izvršnom server programu, kako da se file descriptor podeli sa izvršnim klijent programom preko binder sprege i kako da se izvrši čitanje-pisanje memorije u izvršnom klijent programu nad prenetim file descriptor-om.

Napravite kopiju binder native service-client primera i smestite ga u external direktorijum.

U servisnom projektu otvorite datoteku u kojoj se nalazi implementacija servica i u konstruktoru napravite novi ashmem region. Mapirajte taj region na lokalnu memoriju i upišite proizvoljni sadržaj.

```
//! Create Ashmem region
fd = ashmem_create_region("Ashmem region name",
        ASHMEM_BUFFER_SIZE * sizeof(char));
//! Set permissions
ashmem_set_prot_region(fd, PROT_READ | PROT_WRITE);

//! Map memory
void* ptr = mmap(NULL, ASHMEM_BUFFER_SIZE * sizeof(char),
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
memset(ptr, 0x0, ASHMEM_BUFFER_SIZE * sizeof(char));

//! Write some data to memory
*((int *) ptr) = 0xCAFED00D;

//! Release memory
munmap(ptr, ASHMEM_BUFFER_SIZE);
```

Ne zaboravite da oslobodite file descriptor u destruktoru.

```
close(fd);
```

U funkciji get vratite file descriptor koji ste dobili kao povratnu vrednost funkcije ashmem_create_region(...).

Iako je file descriptor celobrojnog tipa, ne može se preneti sa read|writeInt32 funkcijama, već za njegov prenos koristiti read|writeFileDescriptor funkcije.

Otvorite native deo binder implementacije (Bn*) i umesto reply->writeInt32(retVal); upotrebite writeFileDescriptor funkciju:

```
reply->writeFileDescriptor(retVal);
```

Pošto je promenjen proces serijalizacije na native strani, potrebno je izvršiti simetričnu operaciju na proxy (klijent) strani.

Otvorite Bp* cpp datoteku i umesto retVal = reply.readInt32(); upotrebite readFileDescriptor funkciju.

```
retVal = dup(reply.readFileDescriptor());
```

Potrebno je pozvati i funkciju dup koja vrši dupliranje file descriptora u klijentaskom procesu.

Da bi primer bio funkcionalan potrebno je proveriti stanje deljene memorije u klijentskoj izvršnoj datoteci.

Otvorite datoteku u klijentu koja sadrži poziv na main funkciju i nakon dobavljanja bindera mapirajte lokalnu memoriju na file deskriptor i proverite sadržaj deljene memorije:

```
int fd = example->get();

void * ptr = mmap(NULL, ASHMEM_BUFFER_SIZE * sizeof(char), PROT_READ,
MAP_SHARED, fd, 0);

ALOGI("Memory value: 0x%x", *((int *) ptr));
munmap(ptr, ASHMEM_BUFFER_SIZE);
```

Prevedite servis i klijent izvršne datoteke, spustite ih na emulator i verifikujte pomoću logcat alata funkcionalnu ispravnost primera.

[opciono] Ashmem C/C++ - Java primer

Ideja ovog primera jeste da pokaže kako da se napravi ashmem region u Java Android aplikaciji, kako da se file descriptor podeli sa native programom preko binder sprege i kako da se izvrši čitanje-pisanje memorije u native programu nad prenetim file descriptor-om.

Proširite Binder C/C++ - Java rešenje iz prethodne vežbe i nakon uspešne uspostave veze između native service i native klijenta napravite novi ashmem region sa MemoryFile klasom i upišite proizvoljne podatke u isti:

```
memoryFile mf = new MemoryFile("java-ashmem-region", 1024);

mf.writeBytes(new byte[] { (byte) 170, (byte) 187, (byte) 204, (byte)
221 }, 0, 0, 4);
```

Dohvatite privatno polje mFD iz MemoryFile klase preko refleksije, napravite novi ParcelFileDescriptor i pošaljete ga servisu:

```
Field f = mf.getClass().getDeclaredField("mFD");
f.setAccessible(true);
FileDescriptor fileDescriptor = (FileDescriptor) f.get(mf);

ParcelFileDescriptor pfd = ParcelFileDescriptor.dup(fileDescriptor);
nativeService.set(pfd);
```

Pošto je sada preko metode set šalje ParcelFileDescriptor tip umesto int, potrebno ja napraviti odgovarajuće izmeni u AIDL datoteci:

Otvorite aidl datoteku i promenite potpis set metode.

```
void set(in ParcelFileDescriptor arg);
```

Uključite odgovarajuće zaglavlje:

```
import android.os.ParcelFileDescriptor;
```

Odgovarajuće izmene je potrebno na praviti i na serverskoj strani.

Otvorite Bn* datoteku i u onTransact funkciji promenite način čitanja podataka za SET binder poziv:

```
int outCommChannel;
int fd = data.readParcelFileDescriptor(outCommChannel);
```

Otvorite datoteku u kojoj je izvršena implementacija servisnih funkcija, pronađite set funkciju, povežite file descriptor sa memorijskom lokacijom i pročitajte prva 4 bajta.

```
void* ptr = mmap(NULL, ASHMEM_BUFFER_SIZE * sizeof(char), PROT_READ |
PROT_WRITE, MAP_SHARED, arg, 0);

ALOGI("Memory value set: 0x%x", *((int *) ptr));

munmap(ptr, ASHMEM_BUFFER_SIZE);
```

Prevedite native server i Android aplikaciju. Verifikujte da je server uspešno pročitao upisani sadržaj u ashmem iz Java aplikacije.

Java Native Interface (JNI)

JNI predstavlja programsku spregu Java programskog jezika za pristup platformski zavisnim programskim rešenjima. Pod pojmom platformski zavisnih programskih rešenja podrazumevaju se rešenja pisana programskim jezicima koji zahtevaju prevođenje za svaku ciljnu platformu, kao što su C i C++.

Implementacija JNI programske sprege započinje deklaracijom metoda programske klase koje će naknadno biti implementirane u C/C++ programskim jezikom. Ključnom rečju **native** naznačava se prevodiocu da je implementacija metode deo native programskog rešenja.

```
package rtrk.pnrs.jniexample;

public class FibonacciNative {
    public native int get(int n);
}
```

Funkcije kojima se implementiraju native metode u programskom jeziku C/C++ moraju poštovati pod stroga pravila u pogledu nazivanja i liste parametara. Naziv funkcije u sebi mora sadržati pun naziv Java programske klase i metode koju implementira. Broj i tipovi argumenata moraju se poklapati sa deklaracijom u Javi, gde su prva dva argumenta obavezno kontekst virtuelne mašine i objekta programske klase kojoj metoda pripada. Deklaracija native funkcije za metodu iz prethodnog primera izgleda ovako:

```
jint Java_rtrk_pnrs_jniexample_FibonacciNative_get(JNIEnv *, jobject, jint);
```

Dobra je praksa da se pri deklaraciji native funkcija koristi javah alat. Javah alat predstavlja generator programskog koda kojim se mogu generisati deklaracije native funkcija. Opšti oblik poziva javah alata dat je sledećom komandom:

```
javah -o <output file> -classpath <classes.jar path> <classes>
```

U opštem slučaju poziva javah komande dovoljna su tri parametra: -o predstavlja putanju do datoteke u koju će biti smeštene deklaracije native funkcija, -classpath predstavlja putanju do classes.jar datoteke koja sadrži rezultat prevođenja Java programskog koda i <classes> puni nazivi programskih klasa za čije

metode se generišu deklaracije native funkcija. Sledeća komanda ilustruje poziv javah komande za klasu iz prethodnog primera.

```
javah -o
vendor/rtrk/example_product/JNIExample/jni/rtrk_pnrs_jniexample_FibonacciNativ
e.h -classpath out/target/common/obj/APPS/JNIExample_intermediates/classes.jar
rtrk.pnrs.jniexample.FibonacciNative
```

Nakon deklaracije native funkcija sledi njihova implementacija i prevođenje u vidu deljene programske biblioteke. Deljena programska biblioteka učitava se eksplicitno iz Java programske klase kojoj pripada data native metoda. Učitavanje treba izvršiti u okviru static bloka na sledeći način:

```
package rtrk.pnrs.jniexample;

public class FibonacciNative {
    public native int get(int n);

    static {
        System.loadLibrary("fibonacci");
    }
}
```

U prethodnom primeru parametar loadLibrary metode predstavlja naziv deljene programske biblioteke koja sadrži implementaciju native metoda FibonacciNative klase. Naziv biblioteke zadaje se bez lib prefiksa i ekstenzije datoteke, odnosno podrazumeva se da za naziv iz primera postoji biblioteka sa nazivom libfibonacci.so.

Na sledećoj lokaciji možete pronaći primer aplikacije koja određuje vrednost elementa fibonačijevog niza na osnovu zadatog indeksa "resursi/JNIExample". Prepravite FibonacciNative programsku klasu tako da get bude native metoda. Native deo rešenje implementirati u vidu deljene biblioteke čiji programski kod treba smestiti u JNIExample/jni direktorijum.

[opciono] Upotreba ashmem-a preko JNI-a i binder sprege

Napravite ashmem region u Android java aplikaciji, spustite Memory File klasu u JNI, pristupite preko refleksije file descriptoru Memory File klase i pošaljite ga preko binder sprege native servisu. Za realizaciju Iskoristite prethodne binder i ashmem primere. Primer pristupu preko refleksije file descriptoru iz memory file klase potražite u ashmem predavanjima.

[9] Multimedijalni podsistem

Očekivani rezultati ove vežbe su:

- Ukoliko se vežbe izvode na novijoj verziji Android-a: izvorni kod Beta media player-a, kao i potvrda da on ispravno funkcioniše (logcat ili snimci ekrana).
- Izvorni kod Delta media player-a kao i potvrda da on ispravno funkcioniše (logcat ili snimci ekrana).

Sistemi boja

Razvoj televizora tekao je tako što su se prvo pojavili televizori koji su prepoznavali samo Y komponentu, odnosno televizori koju su prikazivali crno belu sliku. Tokom vremena tehnika je napredovala i pojavili su se televizori u boji. Kako se u jednom periodu vreme istovremeno bili prisutni i crno-beli i televizori u boji, bio je potreban način da se isti signal prikazuje na obe vrste televizora. Iz tog razloga dodate su U i V komponente koje nose informaciju o boji. Na taj način crno-beli televizori su iz TV signala prihvatali i razumeli samo Y komponentu i prokazivali crno-belu sliku, dok su televizori u boji razumeli i U i V komponente i prikazivali sliku u boji. Pored YUV sistema boja, postoje i drugi sistemi kao što su: RGB, CMYK i drugi.

Tokom vremena, došlo se do zaključka da je ljudsko oko daleko osetljivije na gubitak kontrasta (Y komponente) u odnosu na gubitak informacije o bojama (U i V komponente). Pri tom, javila se potreba za prenosom više kanala u etru što je rezultovalo za pronalaženjem načina za kompresiju video okvira. Jedan od osnovnih načina da se video okvir kompresuje jeste da se redukuje količina informacija o bojama i na taj način nastaju razni sistemi: YUV444, YUV422, YUV411 i YUV420. Od svih nabrojanih sistema YUV420 je možda najkorišćeniji jer se koristi u MPEG-2 standardu za kompresiju. Ovaj sistem je organizovan tako da na 4 Y vrednosti ide jedna U i jedna V vrednost.

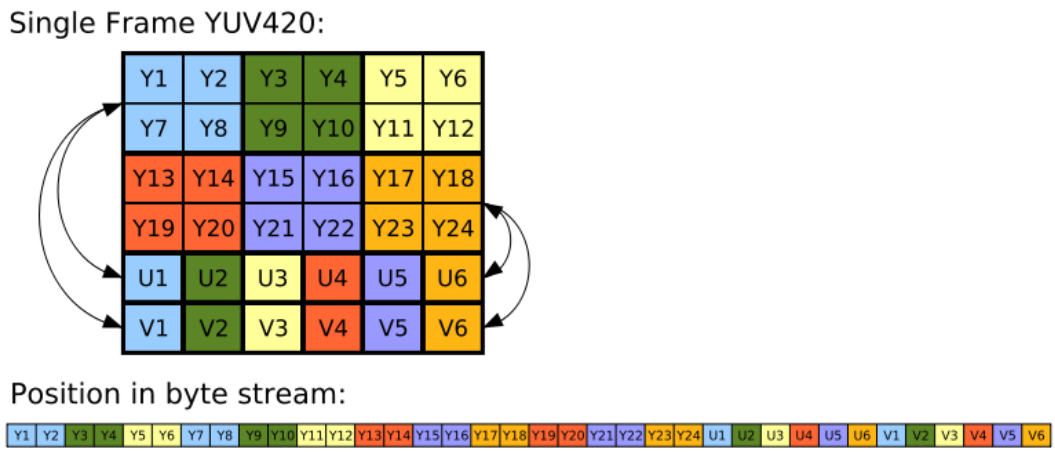


Figure 1 YUV420 video okvir[1]

Prilikom prikazivanja video okvira na ekran, U i V vrednosti se pridružuju odgovarajućim Y vrednostima, tako na primer u kompresovanom formatu uz Y1, Y2, Y7 i Y8 vrednosti idu U1 i V1. Prilikom prikazivanja redosled je sledeći: Y1 U1 V1, Y2 U1 V1, Y3 U1 V1 i Y4 U1 V1.

Beta player

U sledećem primeru biće prikazano kako se može napraviti novi media player u okviru samog jezgra Andoird-a na novijim verzijama (> Jelly Bean).

Android emulator po sebi nema podršku fizičke arhitekture za media codec-e, već se celokupno puštanje audio i video datoteka odvija uz pomoć programske podrške. Naredni primer prikazuje pravljenje i dodavanje novom media player-a baziranog na ffmpeg biblioteci u Android multimedijalni podsistem. Player će biti dodat u media player servise proces posredstvom media player factory mehanizma.

Najlakši način za prevođenje poslednje verzije ffmpeg biblioteke je pomoću NDK programske podrške. Preuzmite sa sledeće stranice ili preko FTP-a NDK verziju 11 za Linux: https://developer.android.com/ndk/downloads/index.html. Raspakujte preuzetu arhivu na putanju: /home/rtrk/ndk/

Podesite NDK promenljivu do putanje NKD programske podrške:

```
export NDK=/home/rtrk/ndk/android-ndk-r11c
```

[1]Preuzeto sa https://en.wikipedia.org/wiki/YUV

Izvedite i ostale promenljive potrebne za korišćenje NDK programske podrške. Proveriti koja verzija toolchaina-a je sadržana u preuzetoj arhivi (x86-4.9).

```
export PLATFORM=$NDK/platforms/android-24/arch-x86
export PREBUILT=$NDK/toolchains/x86-4.9/prebuilt/linux-x86_64
export CPU=x86
export OPTIMIZE_CFLAGS="-march=atom -ffast-math -msse3 -mfpmath=sse"
export PREFIX=./android/$CPU
export ADDITIONAL_CONFIGURE_FLAG=
```

Preuzmite sa FTP ffmpeg 3.0.2 biblioteku i raspakujte je na putanji /home/rtrk/ffmpeg/

Podesite konfiguraciju za prevođenje ffmpeg biblioteke i prevedite je.

```
rtrk@rtrk:~/ffmpeg/ffmpeg-3.0.2$ ./configure \
    --target-os=linux \
    --prefix=$PREFIX \
    --enable-cross-compile \
    --extra-libs="-lgcc" \
    --arch=x86 \
    --cc=$PREBUILT/bin/i686-linux-android-gcc \
    --cross-prefix=$PREBUILT/bin/i686-linux-android- \
    --nm=$PREBUILT/bin/i686-linux-android-nm \
    --sysroot=$PLATFORM \
    --enable-shared \
    --disable-static \
    --disable-doc \
    --disable-ffmpeg \
    --disable-ffplay \
    --disable-ffprobe \
    --disable-ffserver \
    --disable-avdevice \
    --disable-doc \
    --disable-symver \
    --extra-cflags="-Os -fpic $ADDI_CFLAGS" \
    --extra-ldflags="$ADDI_LDFLAGS" \
    $ADDITIONAL_CONFIGURE_FLAG
rtrk@rtrk:~/ffmpeg/ffmpeg-3.0.2$ make -j10 | tee build-make.log
rtrk@rtrk:~/ffmpeg/ffmpeg-3.0.2$ sudo make install | tee build-
install.log
```

Na putanji /home/rtrk/ffmpeg/ffmpeg/3.0.2/android/x86 treba da se nalaze zaglavlja biblioteka i same ffmpeg dinamičke biblioteke.

Napravite ffmpeg direktorijum u external direktorijumu Android-a i tamo iskopirajte include i lib direktorijume dobijene prevođenjem ffmpeg biblioteke.

Da bi se ffmpeg biblioteke uključile u AOSP sliku potrebno ih je uključiti ili ručno kopiran je u out/… direktorijum i pozivanjem make snod komande ili prevođenjem biblioteka kao prebuild.

Napravite novu Android.mk datoteku za „prevođenje“ prevedenih ffmpeg biblioteka na putanji: : /home/rtrk/android-5.1.1_r9/external/ffmpeg/.

Dodajte kod za “prevođenje” svih prevedenih ffmpeg biblioteka: libavcodec, libavformat, libavfilter, libswscale, libavutil i libswresample.

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := libavcodec
LOCAL_SRC_FILES := lib/$(LOCAL_MODULE).so.57.24.102
LOCAL_MODULE_STEM := $(LOCAL_MODULE)
LOCAL_MODULE_SUFFIX := .so.57
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_MODULE_TAGS := optional
include $(BUILD_PREBUILT)

// TODO
// Add recipe for other libraries
```

“Prevedite” prevedene ffmpeg biblioteke

```
rtrk@rtrk:~/android-5.1.1_r9$ mmm external/ffmpeg | tee build-
prebuild-ffmpeg.txt
rtrk@rtrk:~/android-5.1.1_r9$ cat build-prebuild-ffmpeg.txt | grep
Install –color
Install:
out/target/product/example_product/system/lib/libavcodec.so.57
Install:
out/target/product/example_product/system/lib/libavformat.so.57
Install:
out/target/product/example_product/system/lib/libavfilter.so.6
Install: out/target/product/example_product/system/lib/libswscale.so.4
Install: out/target/product/example_product/system/lib/libavutil.so.55
Install:
out/target/product/example_product/system/lib/libswresample.so.2
```

API i upotreba ffmpeg biblioteke prevazilazi okvire ovog primera i iz tog razloga napravljen je tanak omotač oko ffmpeg biblioteke pomoću koga je moguće obaviti osnovne multimedijalne operacije: učitavanje multimedijalne datoteke, dobavljanje osnovnih informacija i dobavljanje video okvira. API omotača je definisan u dve datoteke: FFmpegPlayerBase.h i NextPacket.h.

Pogledajte komentare funkcija i promjenljivih u ove dve datoteke.

Implementacija ove dve datoteke nalazi se u deljenoj biblioteci: libffmpegplayerbase.so. Da bi se ova biblioteka mogla koristiti potrebno ju je iskopirati u /out/target/product/example_product/system/lib direktorijum. To je moguće uraditi na dva načina:

1. Ručno
2. Pomoću Andorid.mk datoteke i BUILD_PREBUILT šablona

Iskopirajte ffmpegplayerbase_prebuild direktorijum u external direktorijum. Otvorite i analizirajte Android.mk datoteku. Obratiti pažnju na LOCAL_MODULE, LOCAL_SRC_FILES i LOCAL_MODULE_CLASS promenljivu. Nakon analize „prevedite" libffmpegplayerbase biblioteku sa mm ili mmm komandom.

Novije verzije Android sistema, umesto u Media Player Servisu, prave novi media player u okviru Media Player Factory klase. Ideja je da svaki media player u multimedijalnom pod sistema ima odgovarajuću Factory klasu koja će biti pozivana od strane Media Player Servisa pre nego što se napravi novi media player. Zadatak Factory klase je da proveri da li traženi multimedijalni sadržaj može da se reprodukuje sa uparenim media player-om i sa kojom verovatnoćom (0.0 – 1.0).

Dodajte identifikator za FFmpeg media player u datoteku MediaPlayerInterface u enumeraciji player_type.

Otvorite MediaPlayerFactory.cpp i napravite FFmpegPlayerFactory, po ugledu na TestPlayerFactory tako da ffmpeg player factory napravi novu instancu ffmpeg media player-a opisanog u FFmpegPlayer.h. Pošto se TestPlayerFactory koristi za reprodukciju multimedijalnog sadržaja sa mreže, a FFmpegPlayerFactory za reprodukciju multimedijalnog sadržaja iz datoteke, postojeću scoreFactory funkciju treba zameniti sa novom:

```
virtual float scoreFactory(const sp<IMediaPlayer>& client, int fd,
   int64_t offset, int64_t length, float curScore) {
           ALOGI("[scoreFactory][by file descriptor]");
           return 1.0;
     }
```

U funkciji registerBuiltinFactories, po ugledu na postojeće Factory klase, dodajte FFmpegPlayerFactory sa identifikatorom navedenim u MediaPlayerInterface.h

Na posletku, da bi Android mogao da pusti multimedijalnu datoteku sa ffmpeg bibliotekom potrebno je implementirati sam media player.

Iskopirajte kostur FFmpeg media player-a: FFmpegPlayer.h i FFmpegPlayer.cpp datoteke na putanju: frameworks/av/media/libmediaplayerservice.

Dodajte u odgovarajuću Andorid.mk datoteku podršku za prevođenje novog media player-a. Ne zaboravite da FFmpeg media player koristi libffmpegplayerbase dinamičku biblioteku.

Implementacija je zamišljena tako da se u okviru setDataSource funkcije izvrši postavljanje FFmpeg biblioteke u početno stanje i spremi multimedijalna datoteka za puštanje. U ovom primeru vršiće se puštanje samo video okvira, jer zbog kompleksnosti sinhronizacije i najverovatnijeg nedostatak zvučnika, zvuk se neće obrađivati i puštati.

Sama setDataSource funkcija kao argument prihvata file descriptor. Pošto ffmpeg nema funkciju za rad sa file desriptorima, potrebno je izvršiti konverziju u oblik koji sadrži apsolutnu putanju do multimedijalne datoteke.

U funkciji status_t setDataSource(int fd, int64_t offset, int64_t length) dodajte poziv setup funkcije iz ffmpegplayerbase biblioteke koristeći ffmpegPlayerBase pokazivač. Kao argument prenesite stvarnu putanju do multimedijale datoteke koja je opisana u realPath promenljivoj.

Ključni resurs za prikazivanje video okvira je surface.

U funkciji status_t FFmpegPlayer::setVideoSurfaceTexture(const sp<IGraphicBufferProducer> &surfaceTexture) sačuvajte vrednost surfaceTexture u promenjljivu mySurfaceTexture. Ova promenljiva će se kasnije koristiti prilikom prikazivanje video okvira korisniku.

```
mySurfaceTexture = new Surface(surfaceTexture);
```

Puštanje video sadržaja otpočinje kada se pozove start funkcija. Ideja primera je da se u tom trenutku napravi i pokrene nit koja će dobavljati jedan po jedan video okvir i vršiti prikaz istog korisniku.

U funkciji status_t start() napravite i pokrenite novu nit:

```
pthread_t decodingThread_thread;
pthread_create(&decodingThread_thread, 0, decodingThread, 0);
```

U implementaciji niti potrebno je dobaviti osnovne informacije o video okviru, kao što su širina i visina. Nakon toga potrebno je konstruisati AMessage promenljivu i napuniti je osnovnim informacijama.

Za prikazivanje video okvira koristi se SoftwareRenderer i funkcija render.

Dodajte implementaciju decodingThread niti u FFmpeg Player klasu:

```
void * decodingThread(void * param){
    unsigned int uiHeight = ffmpegPlayerBase->getHeight();
    unsigned int uiWidth = ffmpegPlayerBase->getWidth();

    sp<AMessage> format;

    ALOGV("[decodingThread][width %d]", uiWidth);
    ALOGV("[decodingThread][height %d]", uiHeight);

    format = new AMessage();
    format->setInt32("stride", uiWidth);
    format->setInt32("slice-height", uiHeight);
    format->setInt32("width", uiWidth);
    format->setInt32("height", uiHeight);
    format->setInt32("color-format", OMX_COLOR_FormatYUV420Planar);
    format->setString("mime", MEDIA_MIMETYPE_VIDEO_RAW);

    SoftwareRenderer sr(mySurfaceTexture);
    NextPacket packet;

    while(bPlaying) {
        if(bPause == true){
            usleep(500 * 1000);
            continue;
        }
        packet = ffmpegPlayerBase->getNextFrame();
        if(packet.bLastPacket == true){
            bPlaying = false;
            break;
        }
        if(pucBuffer == NULL || packet.pucVideoBuffer == NULL){
            ALOGE("[decodingThread][no memory]");
            continue;
        }
        uiFrameCounter ;
        memcpy(pucBuffer, packet.pucVideoBuffer, uiWidth * uiHeight *
1.5);

        sr.render(pucBuffer, uiWidth * uiHeight * 1.5, 0, NULL,
format);
    }

    pthread_exit(0);
    return NULL;
}
```

Prevedite libmedaplayerservice, napraviti novu sistemski sliku (make snod) i pokrenite emulator.

Prođite kroz funkcije start, stop, pause i isPlaying i podesite bPlaying i bPause promenljive na odgovarajuće vrednosti.

Podignite test_file_01.mp4 test datoteku u system direktorijum i promenite joj vlasnika na media_rw:media_rw.

Instalirajte FileManager aplikaciju, otvorite je, pozicionarajte se na /system direktorijum i odaberite test_file_01.mp4 datoteku.

Verifikujte da puštanje video okvira funkcioniše.

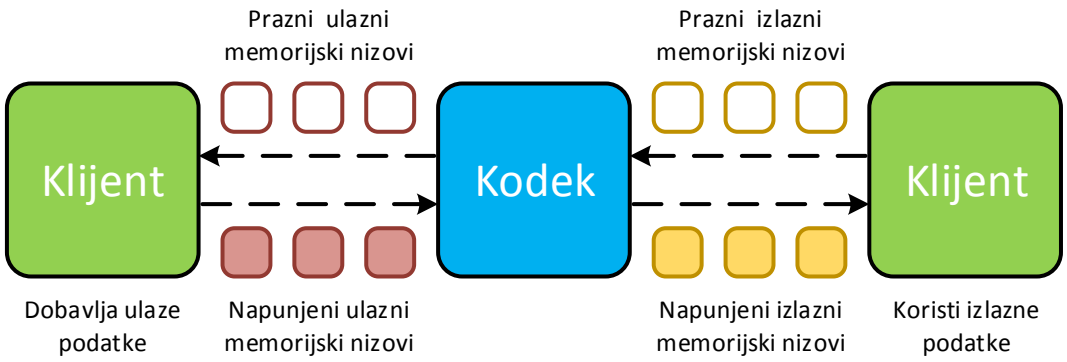
Media Codec players

U sledećim primerima sadrže media player-e bazirane na Media Codec Android API. Cult player je baziran na Media Codec AOSP kodu, a Delta player na Media Codec iz NDK.

Od kada se pojavio sa verzijom Android 4.1 (Jelly Bean, API verzija 16), Media Codec API se sve više koristi za razvoj i pisanje media player aplikacija. Sastoji se od par klasa: MediaExtractor, MediaCodec, MediaSync, MediaMuxer, MediaCrypto, MediaDrm, Image, Surface i AudioTrack koje omogućavaju pristup low-level media kodecima i predstavlja deo infrastrukture za multimedijalnu podršku u sklopu Android-a.

Audio i video sadržaj se može preuzimati iz lokalne datoteke ili mrežnog toga i može se pakovati u različitim oblicima, npr: mp4, ts, avi i drugi. Takođe, pošto audio i video sadržaji zauzima dosta memorije i radi efikasnijeg prenosa oni se na predajnoj strani koduju, čime se smanjuje njihova veličina, i tako prenose do krajnje korisnik. Zadatak Media Extractor-a je da enkapsulira način prenošenja i oblik pakovanja, dok je zadatak Media Codec-a je da na mestu prijema izvrši dekodovanje.

Tok podataka je takav da klijent najpre dobavlja prazne memorijske nizove od Media Codec-a i puni ih kodovanim informacijama dobijenim od Media Extractor-a. Tako napunjene memorijske nizove prosleđuje kodeku na dekodovanje. Drugi klijent (ili isti) preuzima dekodovane memorijske nizove i prikazuje iz korisniku. Nakon prikazivanja, potrošeni izlazni memorijski niz se vraća kodeku na ponovnu upotrebu. Sama komunikacija između klijenata i kodeka se može odvijati sinhrono ili asinhroni. Za prikazivanje video sadržaja koristi se Surface klasa koja apstahuje grafički sloj na Android uređaju.



Slika 1 Tok podataka

Za potrebe primera koristiće se testvideo.mp4 multimedijalna datoteka.

Preuzmite testvideo.mp4 datoteku sa FTP server i podignite je na emulator na putanju: /data/data.

Delta Player

Prezmite i otvorite kostur primera sa FTP-a pod nazivom: DeltaPlayer.

U nastavku će biti prikazano kako se otvara ulazna datoteka i kako se koristi Media Extractor.

Pronađite konstruktor klase DeltaPlayer i otvorite ulaznu datoteku:

```
//! Open test file
mTestFileFD = open(uri, O_RDONLY);
if(mTestFileFD < 0) {
    ALOGE("[DeltaPlayer][failed to open file %s]", uri);
    return;
}

//! Get test file size
mTestFileSize = lseek(mTestFileFD, 0, SEEK_END);
lseek(mTestFileFD, 0, SEEK_SET);
ALOGI("[DeltaPlayer][test file successfully open: %s, size: %u]", uri,
mTestFileSize);
```

U nastavku konstruktora napravite novu instancu Media Extractor klase i prosledite file deskriptor od otvorene multimedijalne test datoteke:

```
//! Create media extractor and set data source
mExtractor = AMediaExtractor_new();
if (mExtractor == NULL) {
    ALOGE("Can't create new extractor.");
    return;
}

status = AMediaExtractor_setDataSourceFd(mExtractor, mTestFileFD, 0,
        mTestFileSize);
if (status != AMEDIA_OK) {
    ALOGE("Can't set fd data source: %d", status);
    return;
}
```

U nastavku konstruktora pronađite koliko traka poseduje ulazna multimedijalna test datoteka i dobavite njihov format:

```
//! Get number of tracks
int trackNum = AMediaExtractor_getTrackCount(mExtractor);
for (int i = 0; i < trackNum; i++) {
    AMediaFormat * format = AMediaExtractor_getTrackFormat(mExtractor,
i);

    //! Here will we create decoders for each track
}
```

U okviru for petlje napravite dobavite MIME tip trake, napravite dekoder i konfigurišite ga:

```
//! Create video decoder
if (isVideo(format) && mVideoIndex == -1) {
    mVideoIndex = i;

    if (AMediaExtractor_selectTrack(mExtractor, mVideoIndex)
            != AMEDIA_OK) {
        ALOGE("Can't select video track");
        return;
    }

    const char * mime = (char *) malloc(
            sizeof(char) * FORMAT_BUFFER_SIZE);
    if (!AMediaFormat_getString(format, "mime", &mime)) {
        ALOGE("Can't get mime string");
        return;
    }

    ALOGI("Video mime: %s", mime);

    mVideoDecoder = AMediaCodec_createDecoderByType(mime);
    if (mVideoDecoder == NULL) {
        ALOGE("Can't create video decoder by type: %s", mime);
        return;
    }

    status = AMediaCodec_configure(mVideoDecoder, format,
gNativeWindow,
            NULL, 0);
    if (status != AMEDIA_OK) {
        ALOGE("Can't configure video decoder");
        return;
    }
    continue;
}
```

```
if (isAudio(format) && mAudioIndex == -1) {
    mAudioIndex = i;

    if (AMediaExtractor_selectTrack(mExtractor, mAudioIndex)
            != AMEDIA_OK) {
        ALOGE("Can't select video track");
        return;
    }

    const char * mime = (char *) malloc(
            sizeof(char) * FORMAT_BUFFER_SIZE);
    if (!AMediaFormat_getString(format, "mime", &mime)) {
        ALOGE("Can't get mime string");
        return;
    }

    mAudioDecoder = AMediaCodec_createDecoderByType(mime);
    if (mAudioDecoder == NULL) {
        ALOGE("Can't create audio decoder by type: %s", mime);
        return;
    }

    status = AMediaCodec_configure(mAudioDecoder, format, NULL, NULL,
            0);
    if (status != AMEDIA_OK) {
        ALOGE("Can't configure audio decoder");
        return;
    }

    int32_t sampleRate;
    if (!AMediaFormat_getInt32(format, AMEDIAFORMAT_KEY_SAMPLE_RATE,
            &sampleRate)) {
        ALOGE("Can't get sample rate for audio decoder");
        return;
    }
    int32_t channelCount;
    if (!AMediaFormat_getInt32(format, AMEDIAFORMAT_KEY_CHANNEL_COUNT,
            &channelCount)) {
        ALOGE("Can't get channel count for audio decoder");
        return;
    }
    mAudioTrack = new AudioTrack(AUDIO_STREAM_MUSIC, sampleRate,
            AUDIO_FORMAT_PCM_32_BIT,
            audio_channel_out_mask_from_count(2));
   continue;
}
```

Pošto je dobavljane podataka dugotrajan posao potrebno je napraviti nit koja će podatke slati dekoderu i prikazivati ih korisniku. U funkciji threadLoop potrebno je implementirati prethodno nabrojane operacije.

Pronađite threadLoop funkciju i dodajte deo za dobavljanje traka i čitanje podataka:

```
while (true) {
    trackIndex = AMediaExtractor_getSampleTrackIndex(player-
>mExtractor);
    sampleTime = AMediaExtractor_getSampleTime(player->mExtractor);

    //! Here we will get input buffer for each track

    //! Fill input buffer with coded data
    ssize_t read = AMediaExtractor_readSampleData(player->mExtractor,
            buffer, bufferSize);
    ALOGI("read sample data: %d", read);
    if (read == -1) {
        ALOGI("End of stream reached");
        sem_post(&player->mPlaybackComplete);
        return NULL;
    }

    //! Decode data

    //! Render data

    //! Read next data
    AMediaExtractor_advance(player->mExtractor);
}
```

U nastavku će biti prikazano kako se dobavljaju ulazni i izlazni memorijski nizovi:

Pronađite threadLoop funkciju i nakon dobavljanja informacije o trakama iz Media Extractor-a u zavisnosti od vrste trake dobavite ulazni memorijski niz za video ili audio sadržaj:

```
if (trackIndex == player->mVideoIndex) {
    //! Get index of free buffer
    bufferIndex = AMediaCodec_dequeueInputBuffer(player-
>mVideoDecoder,
            BUFFER_QUEUE_WAIT_TIME_US);

    if (bufferIndex < 0) {
        continue;
    }

    //! Get free buffer
    buffer = AMediaCodec_getInputBuffer(player->mVideoDecoder,
            bufferIndex, &bufferSize);
}
if (trackIndex == player->mAudioIndex) {
    //! Get index of free buffer
    bufferIndex = AMediaCodec_dequeueInputBuffer(player-
>mAudioDecoder,
            BUFFER_QUEUE_WAIT_TIME_US);
    if (bufferIndex < 0) {
        continue;
    }

    //! Get free buffer
    buffer = AMediaCodec_getInputBuffer(player->mAudioDecoder,
            bufferIndex, &bufferSize);
}
```

U istoj funkciji nakon čitanja podataka od Media Extractor-a prosledite ulazni memorijski niz na dekodovanje:

```
//! Decode data
if (trackIndex == player->mVideoIndex) {
    ALOGI("video queue input buffer");
    AMediaCodec_queueInputBuffer(player->mVideoDecoder, bufferIndex,
0,  read, sampleTime, 0);
}
if (trackIndex == player->mAudioIndex) {
    ALOGI("audio queue input buffer");
    AMediaCodec_queueInputBuffer(player->mAudioDecoder, bufferIndex,
0,  read, sampleTime, 0);
}
```

Nakon uspešnog dobavljanja i dekodovanja audio i video sadržaja potrebno je isti prikazati koristniku. Za prikazivanje video okvira potrebno je dobaviti Surface objekat od Surface Composer servisa.

Dodajte na početku konstruktora klase DeltaPlayer poziv ka getSurface() funkcije čija se implementacija nalazi u istoj datoteci kao i sama klana. Ova funckija dobavlja Surface objekat od Surface Composer servisa i smešta je u globalnu promenljivu gSurface.

Pozovite renderVideo i renderAudio funkcije u ffthreadLoop funkciji nakon dekodovanje memorijskih nizova, a pre dobavljanja sledećih podataka od Media Extractor-a.

```
//! Display decoded data
player->renderVideo(player->mVideoDecoder);
player->renderAudio(player->mAudioDecoder);
```

Render funkcije preuzimaju izlazni memorijski niz od dekodera i prikazuju ih korisniku. Nakon uspešnog prikazivanja, memorijski nizovi se „otpuštaju“, tj vraćaju se dekoderu na ponovno korišćenje.

Da bi primer bio potpuno funkcionalan potrebno je pokrenuti dekodere i nit.

Pronađite funkciju start u DeltaPlayer klasi i pokrenite dekodere i nit za procesiranje:

```
//! Start engine
status = AMediaCodec_start(mVideoDecoder);
if (status != AMEDIA_OK) {
    ALOGE("Can't start video decoder");
    return;
}

status = AMediaCodec_start(mAudioDecoder);
if (status != AMEDIA_OK) {
    ALOGE("Can't start audio decoder");
    return;
}

mAudioTrack->start();

int x;
pthread_create(&threadId, NULL, threadLoop, &x);
```

Prevedite DeltaPlayer i verifikujte njegovu funkcionalnost u Android emulator-u.

[10] Testiranje

Očekivani rezultati ove vežbe su:

- Uspešno pokretanje skupa i pojedinačnih Android JUnit testova.

Ideja vežbe je da se pokaže kako funkcioniše jednostavan Android JUnit test

Prevedite AndroidJUnitExample primer i spustite apk datoteku na emulator.

Analizirajte AndoridManifest.xml i primetiti vrednost android:targetPackage atributa.

Analizirajte ExampleTest klasu.

Povežite se sa konzolom emulatora i pokrenite sve testove sa sledećom komandom:

```
am instrument -w -e class com.rtrk.course.android.ExampleTest
com.rtrk.course.android.AndroidJUnitTest/android.test.InstrumentationT
estRunner
```

Očekivan rezultat je dat u nastavku:

```
com.rtrk.course.android.ExampleTest:....
Test results for InstrumentationTestRunner=....
Time: 0.004
```

Umesto svih testova, pomoću am alata, moguće je pokrenuti pojedinačne testove.

Povežite se sa konzolom emulatora i pokrenite testMultiplication testsa sledećom komandom:

```
am instrument -w -e class
com.rtrk.course.android.ExampleTest#testMultiplication
com.rtrk.course.android.AndroidJUnitTest/android.test.InstrumentationT
estRunner
```

Očekivan rezultat je dat u nastavku:

```
com.rtrk.course.android.ExampleTest:.
Test results for InstrumentationTestRunner=.
Time: 0.002
OK (1 test)
```