# Debugging and Profiling

## Lecture 7

Android Native Development Kit

8 April 2014

# Outline

# Android logging framework

- Logger kernel module
- 4 separate buffers in memory
  - *Main* for application messages
  - *Events* for system events
  - *Radio* for radio-related messages
  - *System* for low-level system debug messages
- Pseudo-devices in /dev/log
- Main, radio and system - 64KB buffers, free-form text
- Event - 256KB buffer, binary format

# Message Structure

- Priority - severity
  - Verbose, debug, info, warning, error, fatal
- Tag identifies the component generating the message
  - Logcat and DDMS can filter log messages based on the tag
- Message: actual log text
  - Buffers are small => do not generate long messages

# Native Logging API

- Exposed through android/log.h
- `#include <android/log.h>`
- `Android.mk` dynamically link native code to log library
  - `LOCAL_LDLIBS += -llog`
  - Before `include $(BUILD_SHARED_LIBRARY)`

# API

- `__android_log_write`
  - Generate a simple string message
  - Params: priority, tag, message

```
__android_log_write(ANDROID_LOG_WARN, "my_native_code",
"Warning message!");
```

- `__android_log_print`
  - Generate formatted string (like printf)
  - Params: priority, tag, string format, other params

```
__android_log_print(ANDROID_LOG_ERROR, "my_native_code",
"Errno =%d", errno);
```

# API

- `__android_log_vprint`
  - Additional parameters as `va_list`

```
void log_verbose(const char* format, ...){
        va_list args;
        va_start(args, format);
        __android_log_vprint(ANDROID_LOG_VERBOSE, "my_-
native_code", format, args);
        va_end(args);
}
```

- `__android_log_assert`
  - Assertion failures
  - Priority is not specified, always fatal

```
__android_log_assert("0 != errno", "my_native_code", "Big
error!");
```

  - SIGTRAP to process - debugger inspection

# Log control

- Cannot suppress log messages based on priority
- Preprocessor based solution

```
#define MY_LOG_NOOP (void) 0

#define MY_LOG_PRINT(level,fmt,...) \
        __android_log_print(level, MY_LOG_TAG, "(%s:%u)␣%s:␣" fmt
        __FILE__, __LINE__, __PRETTY_FUNCTION__, ##__VA_ARGS__)

#if MY_LOG_LEVEL_WARNING >= MY_LOG_LEVEL
#       define MY_LOG_WARNING(fmt,...) \
        MY_LOG_PRINT(ANDROID_LOG_WARN, fmt, ##__VA_ARGS__)
#else
#       define MY_LOG_WARNING(...) MY_LOG_NOOP
#endif
```

# Log Control

- In native code

```
#include "my-log.h"

...

MY_LOG_WARNING("Message!");
```

- In Android.mk

```
MY_LOG_TAG := \"my_native_code\"


ifeq ($(APP_OPTIM),release)

       MY_LOG_LEVEL := MY_LOG_LEVEL_ERROR

else

       MY_LOG_LEVEL := MY_LOG_LEVEL_VERBOSE

endif


LOCAL_CFLAGS += -DMY_LOG_TAG=$(MY_LOG_TAG)

LOCAL_CFLAGS += -DMY_LOG_LEVEL=$(MY_LOG_LEVEL)
```

# Log Console Messages

- STDOUT and STDERR not visible by default
- Redirect stdout and stderr to logging system

```
adb shell stop
adb shell setprop log.redirect-stdio true
adb shell start
```

  - Display with logcat - tags stdout and stderr
  - Temporary config -> erased when booting device
- Permanent config -> modify `/data/local.prop` on device

# Outline

# GDB

- NDK supports debugging using GNU Debugger (GDB)
- `ndk-gdb` script
  - Handles error conditions
  - Outputs error messages
- Requirements
  - Use `ndk-build` -> build system generates files needed for debugging
  - `android:debuggable` in `AndroidManifest.xml`
  - Android version 2.2 or higher

# Debug Session Setup

- `ndk-gdb` script sets up the debug session
- Launches the app using activity manager through ADB
  - Activity manager sends the request to Zygote
  - Zygote forks and creates new process
- `ndk-gdb` starts GDB server and attaches to the app
- Configures port forwarding to make GDB server accessible from the host machine (debug port)
- Copies binaries for Zygote and shared libraries to the host
- Starts GDB client
- Debug session is active -> You can start debugging app
  - Commands sent over the debug port

# Debug from CLI

- Make sure Eclipse is closed
- Go to project directory
- `rm -rf bin obj libs`
- Compile native code using `ndk-build`
- We need `build.xml` -> `android update project -p`
- Compile and package the whole project in debug mode `ant debug`
- Deploy app on device `ant installd`
- `ndk-gdb --start` to start app and the debugging session
- When GDB prompt appears run commands

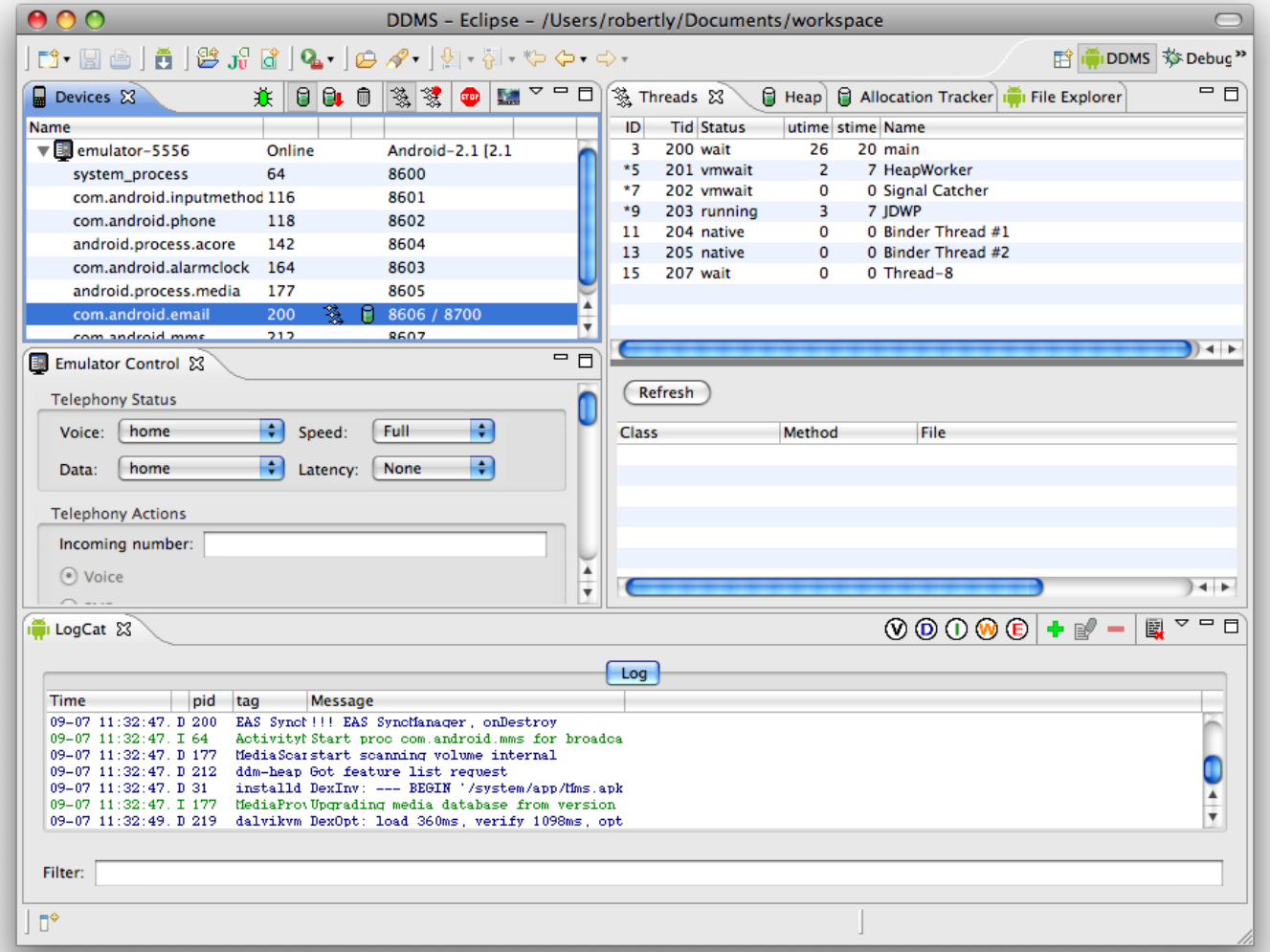
# GDB Commands

- `break`: Breakpoint in a location (function name, file name & line number)
- `clear`: deletes all breakpoints
- `enable/disable/delete`: operations on a certain breakpoint
- `next`: go to the next line in source code
- `continue`: continue execution
- `backtrace`: display call stack
- `backtrace full`: call stack with local variables on frames
- `print`: display variable, expression, memory address, register
- `display`: continue printing value after each step
- `info threads`: list running threads
- `thread`: select a certain thread

# DDMS

- Dalvik Debug Monitoring Server
- Debugging Android applications
- Port-forwarding, screen capture, thread info, heap info, process state, radio state, incoming call, SMS spoofing, location spoofing, etc.
- Integrated in Eclipse, tools/ddms (SDK)
- When started DDMS connects to adb
- VM monitoring service is created between adb and DDMS
- The service notifies DDMS when a VM is started or terminated
- Obtains the pid, opens a connection to the VM's debugger through adbd
- Talks to the VM using a custom wire protocol

# DDMS




Source: http://developer.android.com

# DDMS

- View how much heap is the process using
  - Select process in *Devices* tab
  - *Update Heap* to obtain heap info
  - *Cause GC* to invoke Garbage Collection (refresh data)
  - Select object type to view number of allocated objects
- Track memory allocation
  - *Start Tracking* in the *Allocation Tracker* tab
  - *Get Allocations* to obtain list of allocated objects
  - Finally *Stop Tracking*
  - Detailed info about the method and line that allocated a certain object
- Examine thread info
  - *Update Threads* to obtain thread info for the selected process

# Outline

# Stack Trace

- Use troubleshooting tools and techniques to identify the cause of a problem
- Observe the stack trace when an app crashes with logcat
  - Lines starting with # represent stack calls
  - Line #00 is the crash point
  - After #00 the address is specified (pc)
- `ndk-stack`
  - To add file names and line numbers to the stack trace
  - `adb logcat | ndk-stack -sym obj/local/armeabi`
  - Run command in the project directory
  - Obtain exact file name and line number where it crashed

# CheckJNI

- Extended series of checks before calling JNI functions
- Enable CheckJNI on a device
  - Rooted device

```
adb shell stop
adb shell setprop dalvik.vm.checkjni true
adb shell start
```

    - Logcat: `D AndroidRuntime: CheckJNI is ON`
  - Regular device

```
adb shell setprop debug.checkjni 1
```

    - Logcat: D Late-enabling CheckJNI
- Error detected by CheckJNI

```
W JNI WARNING: method declared to return
'Ljava/lang/String;' returned '[B'
W failed in LJniTest;.exampleJniBug
```

# Libc Debug Mode

- Troubleshoot memory issues
- Enable libc debug mode

```
adb shell setprop libc.debug.malloc 1
adb shell stop
adb shell start
```

- Libc debug mode values
  - 1 - detects memory leaks
  - 5 - detects overruns by filling allocated memory
  - 10 - detects overruns by filling memory and adding sentinel

```
...  testapp using MALLOC_DEBUG = 10 (sentinels, fill)
...  *** FREE CHECK buffer 0xa5218, size=1024, corrupted 1
bytes after allocation
```

# Valgrind

- Advanced memory analysis
- Open-source tool for memory debugging, memory leaks detection and profiling
- Support for Android
- Build from sources
  - Binaries and components in `Inst` directory
  - `adb push Inst /data/local/`
  - Give execution permissions
- Helper script

```
#!/system/bin/sh
export TMPDIR=/sdcard
exec /data/local/Inst/bin/valgrind --error-limit=no $*
```

- Push in /data/local/Inst/bin and set execution permissions

# Valgrind

- To run app under Valgrind, inject the script into the startup sequence

```
adb shell setprop wrap.com.example.testapp  "logwrapper
/data/local/Inst/bin/valgrind_wrapper.sh"
```

- Property wrap.packagename
- Execute app
- Logcat displays Valgrind output

# Strace

- Intercepts system calls and signals
- System call name, arguments and return value
- Useful for analyzing closed-source applications
- Included in Android emulator
- Run the application and obtain pid

```
adb shell ps | grep com.example.testapp
```

- Attach strace to running app

```
adb shell strace -v -p <PID>
```

# Outline

# GNU Profiler (Gprof)

- Unix-based profiling tool
- Compute absolute execution time spent in each function
  - Instrumentation with gcc when using -pg at compile time
  - Sampling data stored at run-time in gmon.out
  - gprof uses gmon.out to produce profiling reports
- Android NDK includes gprof tool
  - Android NDK toolchain lacks the implementation of __gnu_mcount_nc used for timing
- Open-source project Android NDK Profiler

- Install module
  - Download zip, extract in `$NDK_HOME/sources`, rename directory to `android-ndk-profiler`
- Enable profiler
  - Update `Android.mk` to statically link profiling library
  - Include `prof.h` in the native code

```
#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
#include <prof.h>
#endif
```

  - Start collecting profiling data

```
#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
      monstartup("libModule.so");
#endif
```

  - Stop collecting data

```
#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
      moncleanup();
#endif
```

# Generate Reports

- The collected data is stored in `/sdcard/gmon.out`
- App needs permission to write on the SD card

```
<uses-permission android:name="android.permission.WRITE_-
EXTERNAL_STORAGE" />
```

- Pull `gmon.out` from the SD card
- Run gprof

```
$NDK_HOME/toolchains/arm-linux-androideabi-4.4.3/prebuilt/
linux-x86/bin/arm-linux-androideabi-gprof
obj/local/armeabi-v7a/libModule.so gmon.out
```

  - Gprof analyses data and generates a report
  - Two sections: flat profile and call graph
  - Duration of each function

# Outline

# Bibliography


- Onur Cinar, Pro Android C++ with the NDK, Chapter 5, 14
- Sylvain Ratabouil, Android NDK, Beginner's Guide, Chapter 11
- https://code.google.com/p/android-ndk-profiler/wiki/Usage
- http://developer.android.com/tools/debugging/ddms.html

# Outline

- Logger
- Logging API
- Log control
- GDB
- DDMS
- Stack trace
- CheckJNI
- Libc Debug Mode
- Valgrind
- Strace
- Gprof
- Android NDK Profiler