PhD Diary

Wei Minn
Singapore Management University

Version 1.0.5
Last compiled: November 18, 2023

Contents

I 2023 **1**

1 November **3**

1.1 November 14, 2023 . . . 3
1.1.1 AOSP . . . 3
1.1.2 C++ Primer . . . 6
1.2 November 15, 2023 . . . 8
1.2.1 C++ Primer . . . 8
1.2.2 AOSP . . . 11
1.3 November 17, 2023 . . . 13
1.3.1 AOSP Hardware Abstraction Layer . . . 13
1.4 November 18, 2023 . . . 14
1.4.1 Configuring AOSP for Acme Device . . . 15

Part I

2023

Chapter 1

November

1.1 November 14, 2023

1.1.1 AOSP

Zygote

Zygote initializes by pre-loading the entire Android framework. Unlike desktop Java, it does not load the libraries lazily; it loads all of them as part of system start up. After completely initializing, it enters a tight loop, waiting for connections to a socket. When the system needs to create a new application, it connects to the Zygote socket and sends a small packet describing the application to be started. Zygote clones itself, creating a new kernel-level process.

Memory is organized into uniformly sized **pages**. When the application refers to memory at a particular address, the device hardware reinterprets the address as an index into a **page table**. Newly cloned Zygote processes for newly started applications are simply clone of Zygote's page table, pointing to the exact same pages of physical memory. Only the pages the new application uses for its own purposes are not shared:

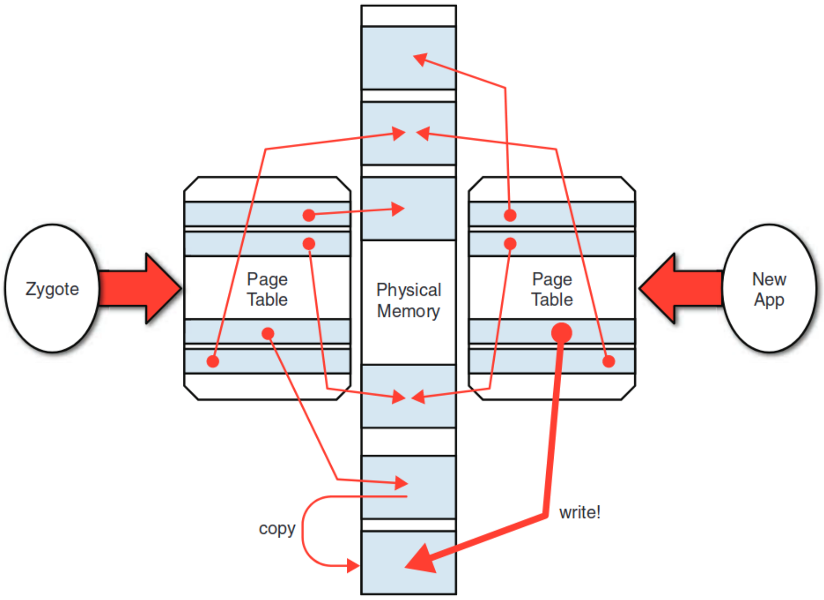

Figure 1.1: Zygote Copy-on Write

Zygote Initialization

Zygote is started by `init`. `ro.zygote` system variable set at platform build time decides which of four types of Zygotes are started and which one is "primary". Both the `init` and Zygote scripts are stored inside `$AOSP/system/core/rootdir`. In the following `init.zygote64_32.rc`, 2 Zygote processes, primary and secondary, are started at 2 different sockets:

```
service zygote /system/bin/app_process64 -Xzygote \
        /system/bin --zygote --start-system-server --socket-name=zygote
    class main
    priority -20
    user root
    group root readproc reserved_disk
    socket zygote stream 660 root system
    socket usap_pool_primary stream 660 root system
    onrestart exec_background - system system -- /system/bin/vdc volume abort_fuse
    onrestart write /sys/power/state on
    onrestart restart audioserver
    onrestart restart cameraserver
    onrestart restart media
    onrestart restart media.tuner
    onrestart restart netd
    onrestart restart wificond
    task_profiles ProcessCapacityHigh MaxPerformance
    critical window=${zygote.critical_window.minute:-off} target=zygote-fatal

service zygote_secondary /system/bin/app_process32 -Xzygote \
        /system/bin --zygote --socket-name=zygote_secondary --enable-lazy-preload
    class main
```

```
    priority -20
    user root
    group root readproc reserved_disk
    socket zygote_secondary stream 660 root system
    socket usap_pool_secondary stream 660 root system
    onrestart restart zygote
    task_profiles ProcessCapacityHigh MaxPerformance
    disabled
```

The actual application that is started as user root at the very highest priority by `init` is `/system/bin/app_process64`. The script requests that `init` create a stream socket for the process and catalog it as `/dev/socket/zygote_secondary` which will be used by the system to start new Android applications.

Zygote is only started once during the system startup, by `app_process64` and `app_-process32`, and is simply cloned to start subsequent applications. Zygote initialization sequence is described below:

Method	Description	Source
`init.rc`	Imports the `init.zygote64_32.rc` that contains the script that starts Zygote service.	`$AOSP/system/core/rootdir`
`init.zygote64_-32.rc`	Runs `app_process64` and `app_process32` which will initialize the starting of Zygote service.	`$AOSP/system/core/rootdir`
`app_process`	Creates `AppRuntime`, a subclass of `AndroidRuntime`, that does bookkeeping, naming the process, setting up parameter, and the name of the class to run when not running Zygote, and then calls `AndroidRuntime.start()` to invoke the runtime.	`$AOSP/frameworks/base/cmds/app_-process`
`AppRuntime::start`	Invokes `startVM.startVM` which invokes `JNI_CreateJavaVM`.	`$AOSP/frameworks/base/core/jni/Andro`
`JNI_CreateJavaVM`	Calls `Runtime::Create`.	`$AOSP/art/runtime/jni/java_-vm_ext.cc`
`Runtime::Create`	Initializes the ART runtime, loading the system OAT files and the libraries they contain.	`$AOSP/art/runtime/runtime.cc`

Table 1.1: Zygote Initialization Sequence

The argument that `app_process` passed to `start` is `com.android.internal.os.Zygote.Init`, the source for which is in `$AOSP/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java`. `app_process` is the launcher for all Java programs (not apps!) in the Android system, and Zygote is one example of the programs (system service) to be launched.

Zygote System Service

Zygote has 3 major tasks, on startup:

1. Register the socket to which the system will connect to start new application. Handled by `registerServerSocket` method which creates socket using the named passed as parameter for `init` script.
2. Preload Android resources (classes, libraries, resources and even WebViews) with a call to `preload` method. After `preload` is finished, Zygote is fully initialized and ready to clone to new applications very quickly.
3. Start Android System Server. Thus, `SystemServer` is the first application to be cloned by Zygote.

After it has completed these three tasks, it enters a loop, waiting for connections to the socket.

1.1.2 C++ Primer

Primitive Built-in Types

Includes **arithmetic types** and a special type named **void** which has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value.

The arithmetic types are divided into two categories: **integral types** (which include character and boolean types) and floating-point types.

Type	Meaning	Minimum Size
`bool`	boolean (`true` or `false`)	NA
`char`	character	8 bits
`w_char_t`	wide character	16 bits
`char16_t`	Unicode character	16 bits
`char32_t`	Unicode character	32 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`long long`	long integer	64 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits

`long double`	extended-precision floating-point	10 significant digits

Table 1.2: Zygote Initialization Sequence

Except for `bool` and extended character types, the integral types may be **signed** (can represent negative or positive numbers) or **unsigned**. By default, `int`, `short`, `long`, `long long` are all signed. To declare unsigned type, prepend `unsigned` to the type. `char` is signed on some machine and `unsigned` on others, and `unsigned int` is abbreviated as `unsigned`.

Conversions happen automatically when we use an object of one type where an object of another type is expected.

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // prints -84
std::cout << u + i << std::endl; // if 32-bit ints, prints 4294967264
```

In the above snippet, converting a negative number to `unsigned` will cause the value to "wrap around" because `unsigned` values can never be less than 0. Thus, extra care should be taken if we want to write loops with `unsigned` values and stopping conditions at negative values like the snippet below:

```
// WRONG: u can never be less than 0; the condition will always succeed
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

As such it is always advisable to not mix `signed` and `unsigned` types. By default, integer literals (42) are signed, while octal (`024`) and hexadecimal (`0x14`) may be signed or unsigned.

Escape sequences are used as if they were single characters:

```
std::cout << '\n'; // prints a newline
std::cout << "\tHi!\n"; // prints a tab followd by "Hi!" and a newline
```

Variables

Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

Four different ways to initialize:

```
int units_sold = 0;
int units_sold = {0}; // list initialization; does not work for built-in types if data loss is likely
int units_sold{0};
int units_sold(0);
```

Variables defined outside any function body are initialized to zero by default. Variables of built-in type defined inside a function are **uninitialized** and therefore undefined. Objects of class type that we do not explicitly initialize have a value that is defined by the class.

A **declaration** makes a name known to the program. A file that wants to use a name defined elsewhere includes a declaration for that name. A **definition** creates the associated entity. A definition involves declaration, allocates storage and may provide the variable with an initial value.

```
extern int i;   // declares but not define j
int j;          // declares and defines j
int k = 12;     // declares, defines and initializes j
extern double pi = 3.14;     // definition
```

Variables must be defined only once but can be declared several times. To use a variable in more than one file requires declarations that are separate from the variable's definition. To use the same variable in multiple files, we must define that in one - and only one - file. Other files that use that variable must declara - but not define - that variable.

1.2 November 15, 2023

1.2.1 C++ Primer

Scopes of Names

Most scopes in C++ are delimited by curly braces.

```
#include <iostream>
int main() {
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum=sum+val
    std::cout << "Sum␣of␣1␣to␣10␣inclusive␣is␣"
        << sum << std::endl;
    return 0;
}
```

In above program, `main` - like most names defined outside a function - has **global scope** and thus, is accessible throughout the program. `sum` has **block scope** and is accessible from its point of declaration throughout the rest of the `main` function. `val` is defined in the scope of the `for` statement and can be used in that statement but not elsewhere in `main`.

Names declared in the outer scope can also be redefined in an inner scope although it is always a bad idea:

```
#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42; // reused has global scope
int main() {
    int unique = 0; // unique has block scope

    // output #1: uses global reused;prints 42 0
```

```
    std::cout << reused << "␣" << unique << std::endl;

    int reused = 0; // new, local object named reused hides global reused
    // output #2: uses local reused; prints 0 0
    std::cout << reused << "␣" << unique << std::endl;

    // output #3: explicitly requests the global reused; prints 42 0
    std::cout << ::reused << "␣" << unique << std::endl;

    return 0;
}
```

When the scope operator (`::` **operator**) has an empty LHS, it is a request to fetch the name on the RHS from the global scope.

References

A **reference** defines an alternative name for an object. A reference type can be defined by writing a declarator of the form `&d` where `d` is the name being declared:

```
int ival = 1024;
int &refVal = ival; // refVal refers to (is another name for) ival
int &refVal2; // error: a reference must be initialized
```

When we define a reference, instead of copying the initializer's value, we bind the reference to its initializer. Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, references must be initialized.

A reference is not an object. Instead, a reference is just another name for an already existing object. Thus, *all* operation on that reference are actually operations on the object to which the reference is bound:

```
refVal = 2; // assigns 2 to the object to which refVal refers, i.e., to ival
int ii = refVal; // same as ii=ival
```

Becuase references are not objects, we may not define a reference to a reference. We can define references in a single definition with each identifier that is a reference being preceded by the `&` symbol:

```
int i = 1024, i2 = 2048; // i and i2 are both ints
int &r = i, r2 = i2; // r is a reference bound to i; r2is an int
int i3 = 1024, &ri = i3; // i3 is an int; riis a reference bound to i3
int &r3 = i3, &r4 = i2; // both r3and r4are references
```

Pointers

Like references, pointers are used for indirect access to other objects. Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. Unlike a reference, a pointer need not be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

We define a pointer type by writing a declarator of the form *d,whered is the name being defined. The * must be repeated for each pointer variable:

```
int *ip1, *ip2; // both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

A pointer holds the address of another object. We get the address of an object by using the address-of operator (& **operator**):

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival

double dval;
double *pd = &dval; // ok: initializer is the address of a double
double *pd2 = pd; // ok: initializer is a pointer to double
int *pi = pd; // error: types of pi and pd differ
pi = &dval; // error: assigning the address of a doubletoapointertoint
```

We can use the dereference operator (* **operator**) to access that object:

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
cout << *p; // * yields the object to which p points; prints 42

*p = 0; // * yields the object; we assign a new value to ival through p
cout << *p; // prints 0
```

When we assign to *p, we are assigning to the object to which p points. We may dereference only a valid pointer that points to an object.

void* is a special pointer type that can hold the address of any object. Its useful for when the type of the object at that address is unknown:

```
double obj = 3.14, *pd = &obj; // ok: void*can hold the address value of any data
    pointer type
void *pv = &obj; // objcan be an object of any type
pv = pd; // pvcan hold a pointer to any type
```

The modifiers * and & do not apply to all variables defined in a single statement:

```
int* p1, p2; // p1 is a pointer to int; p2is an int
int *p1, *p2; // both p1and p2are pointers to int
```

As pointers are objects in memory, they also have addresses of their own. Therefore, we can store the address of a pointer in another pointer:

```
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to an int
```

We indicate each pointer level by its own *. Dereferencing a pointer to a pointer yields the pointer. So in this case, you must dereference twice to access the underlying object.

1.2.2 AOSP

Starting Android System Server and Other Apps using Zygote

During its initialization, Zygote will check for `start-system-server` flag, and if set, will bring up `SystemServer` in the following sequence:

Method	Description	Source
`ZygoteInit.forkSystemServer`	Runs after the Zygote process has been initialized. It is hardcoded with System Server classpath[1] as one of the arguments to call `Zygote.forkSystemServer` to spawn `SystemServer` process.	`AOSP/framework/base/core/java/com/android/internal/os/ZygoteInit.java`
`Zygote.forkSystemServer`	`Zygote` class wraps native methods that communicate with Android Runtime, one of whom is `com_android_internal_os_nativeForkSystemServer`.	`AOSP/framework/base/core/java/com/android/internal/os/Zygote.java`
`com_android_internal_os_nativeForkSystemServer`	Calls `zygote::ForkCommon` and `SpecializeCommon` which does the actual forking.	`AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`
`SpecializeCommon`	Looks at the flags and Process ID for setting up sandboxing, configuring the correct SE Linux context, and process capabilities. Afterwards, it will call `Zygote` methods for post-fork procedures.	`AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`
`Zygote.callPostForkSystemServerHooks`	Calls `ZygoteHooks.java` at the end of specialization procedures. Only applicable for `SystemSever`.	`AOSP/framework/base/core/java/com/android/internal/os/Zygote.java`
`Zygote.callPostForkChildHooks`	Calls `ZygoteHooks.java` at the end of specialization procedures. Applicable to all applications and services including `SystemServer`	`AOSP/framework/base/core/java/com/android/internal/os/Zygote.java`
`ZygoteHooks.postForkSystemServer` and `ZygoteHooks.postForkSystemServer`	Wrappers for `ZygoteHooks` inside the Android Runtime. They call their respective native code inside the ART via Java Native Interface.	`AOSP/libcore/dalvik/src/main/java/dalvik/system/ZygoteHooks.java`

[1] `com.android.server.SystemServer`, the source for which is stored in `AOSP/frameworks/base/services/java/com/android/server/SystemServer.java`.

Method	Description	Source
ZygoteHooks_nativePostForkSystemServer	Loads the specialized class libraries to start the the System Server.	AOSP/art/runtime/native/dalvik_system_ZygoteHooks.cc
ZygoteHooks_nativePostForkChild	Loads the specialized class libraries to start the service/application.	AOSP/art/runtime/native/dalvik_system_ZygoteHooks.cc
handleSystemServerProcess	The control returns to ZygoteInit, and finish remaining work for the newly forked system server process, and calls ZygoteInit.zygoteInit.	AOSP/framework/base/core/java/com/android/internal/os/ZygoteInit.java
ZygoteInit.zygoteInit	The main function called when started through the zygote process, which calls RuntimeInit.applicationInit	AOSP/framework/base/core/java/com/android/internal/os/ZygoteInit.java
RuntimeInit.applicationInit	Calls the public static void main method of the application	AOSP/framework/base/core/java/com/android/internal/os/RuntimeInit.java
ZygoteServer.runSelectLoop	After forking has finished, the control enters ZygoteServer which starts an endless loop that handles incoming connections with ZygoteConnection.processCommand.	AOSP/framework/base/core/java/com/android/internal/os/ZygoteServer.java
ZygoteConnection.processCommand	Calls Zygote.forkAndSpecialize which is a version of ZygoteInit.forkSystemServer for the masses.	AOSP/framework/base/core/java/com/android/internal/os/ZygoteConnection.java
Zygote.forkAndSpecialize	A version of Zygote.forkSystemServer for the masses.	AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp
com_android_internal_os_nativeForkAndSpecialize	Calls zygote::ForkCommon and SpecializeCommon which does the actual forking, and returns to ZygoteInit and immediately enters ZygoteServer.	AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp

Table 1.3: System Server and Applications Initialization Sequence

1.3 November 17, 2023

1.3.1 AOSP Hardware Abstraction Layer

The interface to the hardware is a device drivers which are usually device specific and sometimes proprietary. A single set of C header files describes the functionality that a HAL provides to the Android system. HAL Code for a particular device is the implementation of the API defined by those header files, so that no code above the HAL needs to be changed to port Android to use the new device.

Building Hardware Abstraction Layer

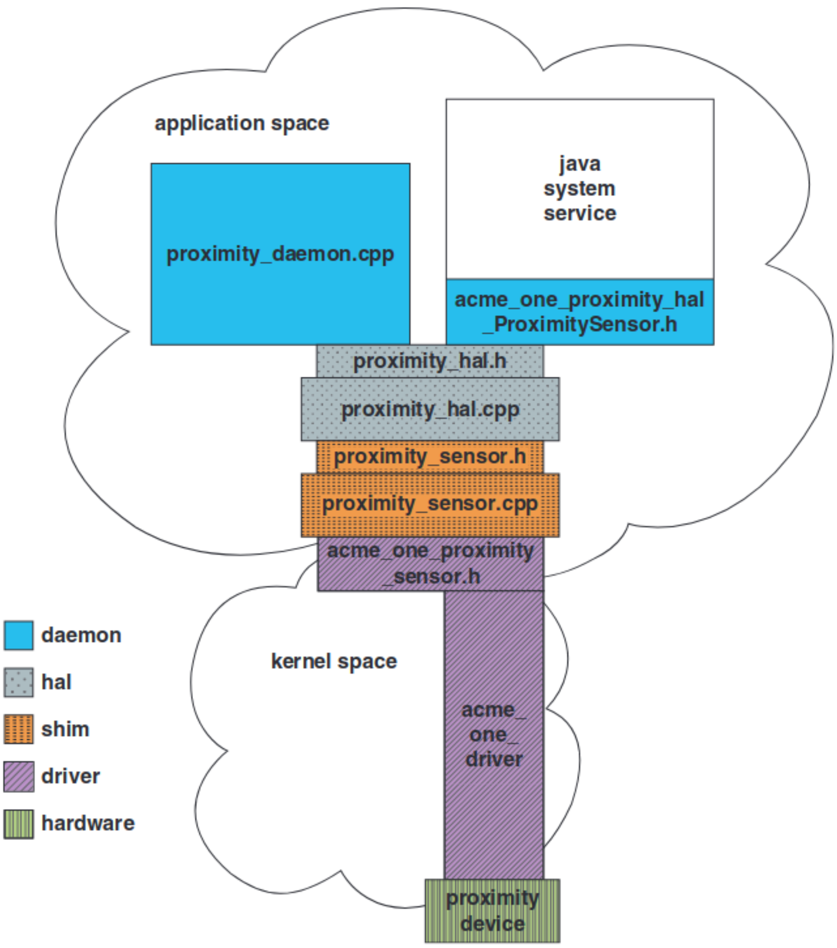

Figure 1.2: HAL Layer Structure

The code consists of four functional components as show in Figure 1.2:

1. **HAL code (dotted boxes):** Abstraction that separates the capabilities of hardware from its specific implementations. The `.h` file defines the HAL interface, and the implementation (`.cpp` file) specializes the Android HAL API for the target hardware.

2. **Shim code (dashed boxes):** Glue code that connects the HAL to a specific device hardware/driver. This code adapts the Android HAL API to the device driver for the hardware.

3. **Daemon (blue):** Stand-alone application that interacts with the hardware through the HAL.

4. **Java System Service (white):** System Service that Android applications will use to access the custom hardware.

The source for those components are structured like in the directory tree below:

```
one
├── app
├── native_daemon
│   └── ...
├── java_daemon
│   └── ...
└── proximity
    ├── include
    │   ├── dev
    │   │   └── ...
    │   └── ...
    ├── dev
    │   └── ...
    ├── hal
    │   └── ...
    └── jni
        └── ...
```

where `one` is the device folder of the AOSP project. All the code implementing the HAL for the proximity sensors goes into a new subdirectory `proximity`.[2]

1.4 November 18, 2023

[2]To be a "real" HAL, the interface `proxmity/include/proximity_hal.h` would have to be promoted from its current directory specifically for the One device, up into the Android source tree to a location that would make it visible to other code that needed to use it. Here, it is only shared by Acme devices, so it is put under the subdirectory of the Acme device directory. If it's visible across device from multiple vendors, it might be promoted into the `device` directory itself.

1.4.1 Configuring AOSP for Acme Device

Repo Manifest

Top-level subdirectory named `.repo` contains the `manifests` repository inside `manifests` subdirectory. The `manifests` repo contain one or more manifest files named as the argument of the `-m` command line option. `.repo/manifest` file controls the structure of the rest of the repository, and includes `.repo/manifests/default.xml`[3] which is a list of git repositories. `repo` program parses `manifest.xml`, and thus `default.xml`, and clone each repository into a location specfied inside `default.xml`.

Each `project` element in the XML identifies a git repository by its `name`, relative to some base URL, its `remote`; and where that repository should be placed in the local workspace, its `path`. If the full URL for the repository is not specified, repo will use the default remote specified in the `default` element near the top of the manifest:

```
<default revision="refs/tags/android-13.0.0_r11"
    remote="aosp"
    sync-j="4" />
```

where the remote, `aosp`, is defined likewise in the top of of the `default.xml`:

```
<remote  name="aosp"
    fetch=".."
    review="https://android-review.googlesource.com/" />
```

Instead of including a URL as its attribute, it includes the `fetch` attribute which indicatese the URL for this remote should be derived from the URL used to initialize the workspace (the argument to the `-u` option).

[3] https://gerrit.googlesource.com/git-repo/+/master/docs/manifest-format.md