Chapter IV - Android Runtime Services

Note: This file is a sample chapter from the full book - "Android Internals: A confectioner's cookbook" - which can be found on http://NewAndroidBook.com/. The chapter was made available for free as a preview of the book (think of it like Amazon's "Look Inside" :-). I encourage you to check out Technologeeks.com Android Internals training, which builds on the book and expands it further with Instructor Led Training.

You can also preorder the book by emailing p r eorder @ The Book's domain.

Note some links (to other chapters in the book) will not work in this file (since it is partial), but external links will. Feedback, questions and requests are always welcome.

Android has quite a few daemons running in the background for providing its miscellaneous housekeeping and operational functions. The services are mostly strewn in `/init.rc` without much ordering, save the service class. The "core" services start first, followed by the "main" ones. The rc also defines a "late_start" class, for services which depend on the `/data` partition, though no default services belong to it. In this section, we adopt the service class division, but - since most services are in "main" - further subcategorize by function.

We begin with a discussion of init, which is the very first process to launch (PID 1), as thus serves as the progenitor of all user mode processes. The Android `init` is different than that of Linux, with the most important differences being in its support of System Properties and using a particular set of rc files. Following the explanation of those two features, we piece together the flow of init: its Initialization and Run-Loop.

As it so happens, `init` also fills additional roles - assuming the guise of ueventd and watchdogd, two important core services which are also implemented by init, loaded through a symbolic link. The discussion continues to cover the other Core Services - adbd, the servicemanager and KitKat's healthd, as well as new core services added in L: lmkd and logd.

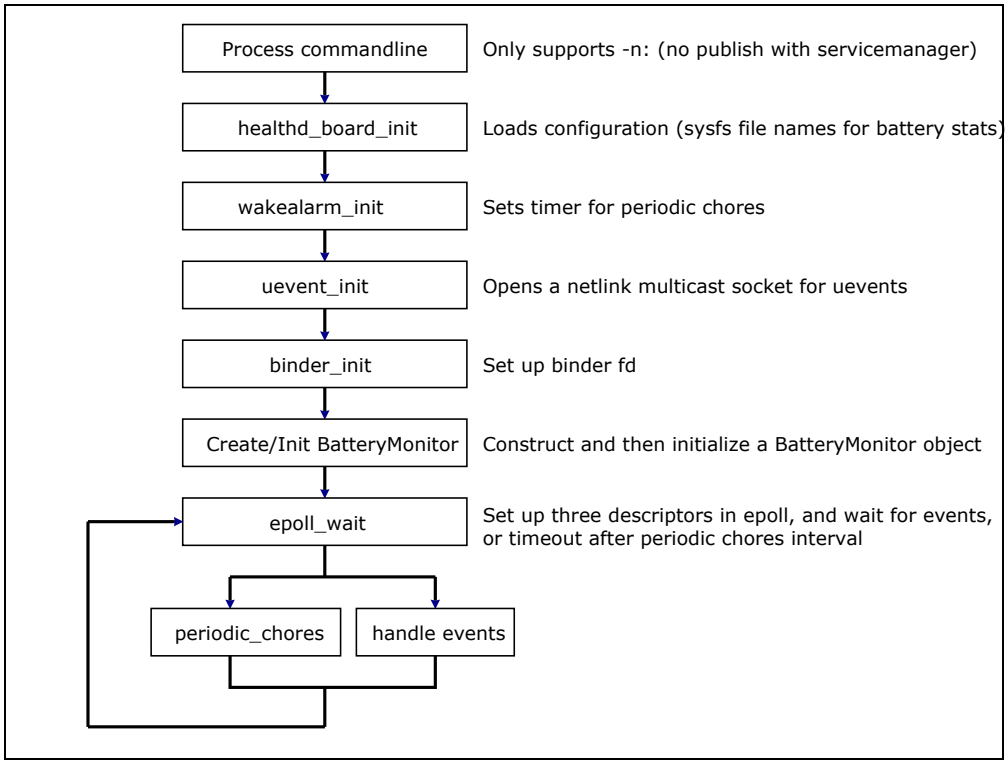
All other services are generally classified into the "main" category, so a subcategorization by Network Services (netd, mdnsd, mtpd and rild), and Graphics and Media Services (surfaceflinger, bootanimation, mediaserver and drmserver) follows. The remaining services are hard to group, so they are placed into the "Other Services" category, which includes installd, keystore, debuggerd, sdcard and - last, but far from least - Zygote.

healthd

The "health daemon" is meant to service general "device health" tasks periodically, though at present the only tasks are battery related (this will likely change in future releases). The daemon registers itself as the `BatteryPropertiesRegistrar` service (`batterypropreg` or `batteryproperties` in L). As the Registrar, healthd provides the framework services (e.g. `BatteryStatsService`) with up-to-date battery statistics, which it obtains from sysfs.

Like most daemons, `healthd` sets up an initial configration, and then enters a run loop. The detailed flow is shown in Figure 4-hdsf:

Figure 4-hdsf: The flow of healthd



Healthd main loop blocks on the Linux `epoll(2)` API to multiplex read operations on three descriptors, and registers actions for each, as shown in the following table:

Table 4-hdfdt: The file descriptors held by `healthd` and their purpose

Descriptor	Type	Purpose
wakealarm_fd	TimerFD	Timer set to fire every `periodic_chores_interval` seconds. Upon wakeup, `healthd` runs `periodic_chores`.
event_fd	Netlink	Reads kernel notification events. `healthd` only concerns itself with those of the power subsystem (`SUBSYSTEM=POWER`). These events include battery and charger notifications, and `healthd` runs `battery_update()`.
binder_fd	/dev/binder	Listener updates by framework clients (when acting as `batterypropreg`)

The first descriptor polled is the `wakealarm_fd`, which `healthd` uses for its periodic chores. Two interval types are used: fast (1 minutem, when on AC power), and slow (10 minutes, on battery)*. The only chore presently defined is `battery_update()`, which updates battery statistics in healthd's role as the `BatteryPropertiesRegistrar`. This is also called when events from the `POWER` subsystem are received over netlink from `event_fd`: `healthd` makes no attempt to parse the events, and merely refereshes the battery statistics. The latter mode is required in order for `healthd` to respond to events such as charger [dis]connection, or other power management alerts. Finally, the `binder_fd` is used to interact with the framework listeners (primarily, the `BatteryStatsService`), as described in the next chapter.

---

* - Interestingly enough, in many Android releases the `timerfd_create` call returns -EINVAL (Invalid argument), not creating `wakealarm_fd` and thus defaulting to polling on the `event_fd` as an event source alone.

Experiment: Observing `healthd`

Using the powerful `strace(1)` utility you can watch `healthd` behind the scenes: By attaching to its process ID (as root) and calling on the `ptrace(2)` API, `strace(1)` can get notifications of system calls. Because anything meaningful a process does goes through a system call, this will provide a detailed trace of the activity, and reveal the names of the sysfs files `healthd` uses to obtain its statistics, as shown in the following annotated output:

Output 4-hdstr: Using `strace(1)` on `healthd`

```
root@htc_m8wl:/ # ls -l /proc/$healthd_pid/fd | cut -c'1-10,55-'
lrwx------ 0 -> /dev/null
lrwx------ 1 -> /dev/null
lrwx------ 2 -> /dev/null
l-wx------ 3 -> /dev/__kmsg__ (deleted)                    # Output: Log to kernel
lrwx------ 4 -> socket:[6951]                              # event_fd (Netlink socket)
lrwx------ 5 -> /dev/binder                                # binder_fd
lrwx------ 6 -> anon_inode:[eventpoll]                     # epollfd
l-wx------ 7 -> /dev/cpuctl/apps/tasks                     # fg_cgroup_fd (libcutils)
l-wx------ 8 -> /dev/cpuctl/apps/bg_non_interactive/tasks # bg_cgroup_fd (libcutils)
lr-x------ 9 -> /dev/__properties__                        # r/o property fd
root@htc_m8wl:/ # strace -p $healthd_pid
Process $healthd_pid attached - interrupt to quit
# healthd patiently polling (0xffffffff = indefinitely) until an fd signals an event
epoll_wait(0x6, 0xbebb5898, 0x2, 0xffffffff) = 1
# Netlink msg received on fd 4 (event_fd) - indicating core state change (going offline)
recvmsg(4, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000001},
msg_iov(1)=[{"offline@/devices/system/cpu/cpu1"..., 1024}], msg_controllen=24,  ....
# healthd's not interested, so it goes back to polling
epoll_wait(0x6, 0xbebb5898, 0x2, 0xffffffff) = 1
# message indicating change in battery status:
recvmsg(4, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000001},
msg_iov(1)=[{"change@/devices/platform/htc_bat"..., 1024}], msg_controllen=24,
{cmsg_len=24, cmsg_level=SOL_SOCKET, cmsg_type=SCM_CREDENTIALS{pid=0, uid=0, gid=0}},
msg_flags=0}, 0) = 488
#
# healthd goes into a flurry of statistics collection, opening and closing files:
#
open("/sys/class/power_supply/battery/present", O_RDONLY) = 10     # Is battery present?
read(10, "1\n", 16)                     = 2                        # Yes (1)
close(10)                               = 0
open("/sys/class/power_supply/battery/capacity", O_RDONLY) = 10    # What is its capacity?
read(10, "96\n", 128)                   = 3                        # 96%
close(10)                               = 0
open("/sys/class/power_supply/battery/batt_vol", O_RDONLY) = 10    # Voltage?
read(10, "4303\n", 128)                 = 5
close(10)                               = 0
open("/sys/class/power_supply/battery/batt_temp", O_RDONLY) = 10   # Temperature?
read(10, "265\n", 128)                  = 4
close(10)                               = 0
open("/sys/class/power_supply/battery/status", O_RDONLY) = 10      # Charge status?
read(10, "Charging\n", 128)             = 9                        # Charging
close(10)                               = 0
open("/sys/class/power_supply/battery/health", O_RDONLY) = 10      # Battery Health
read(10, "Good\n", 128)                 = 5
close(10)                               = 0
open("/sys/class/power_supply/battery/technology", O_RDONLY) = 10  # Battery Type
read(10, "Li-poly\n", 128)              = 8
close(10)                               = 0
open("/sys/class/power_supply/ac/online", O_RDONLY) = 10           # AC is not connected
read(10, "0\n", 128)                    = 2
close(10)                               = 0
open("/sys/class/power_supply/usb/online", O_RDONLY) = 10          # USB is connected
read(10, "1\n", 128)                    = 2
close(10)                               = 0
open("/sys/class/power_supply/usb/type", O_RDONLY) = 10
read(10, "USB\n", 128)                  = 4
close(10)                               = 0
open("/sys/class/power_supply/wireless/online", O_RDONLY) = 10   # Alas, no wireless charging
read(10, "0\n", 128)                    = 2                      # for the M8
close(10)                               = 0
write(3, "<6>healthd: battery l=96 v=4 t=2".., 51) = 51          # Report to kernel log
ioctl(5, BINDER_WRITE_READ, 0xbebb5070)       = 0               # Report to client listeners
epoll_wait(0x6, 0xbebb5898, 0x2, 0xffffffff) = ..               # Back to polling
```

Note the sysfs psuedo files (/sys/class/power_supply/*) are standard - in practice they are symbolic links to the specific platform device nodes, which change between devices.

Experiment: Observing `healthd` (cont.)

As an improvement on the above, you might want to send the `strace` into the background (by using `&`) and then disconnect and reconnect the USB cable. You will then see the netlink notification for battery change, followed by a change in /sys/class/power_supply/usb/online (from 1 to 0 on disconnect, or vice versa on connect).

As of Android L, healthd supports `dumpsys`. You can actually take the Android L binary (from the Google Nexus 5 or Emulator) and copy it to a real device, as shown in this output, from the author's HTC One M8:

```
#
# Before: Only KK healthd - note old service name (batterypropreg)
#
root@htc_m8wl:/ # service list | grep batteryprop
91      batterypropreg: [android.os.IBatteryPropertiesRegistrar]
root@htc_m8wl:/ # /data/local/tmp/healthd.L &                    # Run healthd from L
[1] 7287
#
# After: new service name (batteryproperties) added. Name is different, so no conflict
#
root@htc_m8wl:/ # service list | grep batteryprop
0       batteryproperties: [android.os.IBatteryPropertiesRegistrar] # Note same interface
92      batterypropreg: [android.os.IBatteryPropertiesRegistrar]
root@htc_m8wl:/ # dumpsys batteryproperties                      # Calling dumpsys
ac: 0 usb: 1 wireless: 0
status: 5 health: 2 present: 1
level: 100 voltage: 4 temp: 273
```

If you use `strace` to watch behind the scenes of `dumpsys`, you'll see the following output (file descriptors are different here, so they've been symbolically replaced)

```
epoll_pwait(epoll_fd, {{EPOLLIN, {u32=37597, u64=12884939485}}}, 2, -1, NULL) = 1
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1748)         = 0        # Incoming binder req
write(...tasks, healthd_pid, 4)                         = 4        # Make healthd foreground
..
write(new_fd, "ac: 0 usb: 1 wireless: 0\n", 25) = 25               # Write output
write(new_fd, "status: 5 health: 2 present: 1\n", 31)  = 31        # to binder supplied
write(new_fd, "level: 100 voltage: 4 temp: 273\n", 32) = 32        # file descriptor.
fsync(new_fd)              = -1 EINVAL (Invalid argument)
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1600)       = 0
close(new_fd)                                 = 0
write(...tasks, healthd_pid, 4)                         = 4        # Make healthd background
..
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1758)       = 0
```

This example, aside from showing the inner workings of `healthd` on L, also demonstrates an important part of Android: IPC over binder. In the above, you can see how a file descriptor has been passed from the calling process (`dumpsys`) to `healthd`. Binder internals are a complicated discussion in their own right, and are left for Chapter 20.

Though merely a speculation, it is likely that `healthd` will be augmented and play an increasingly larger role in Android, possibly starting with L. A hint as to its importance can be found in the fact that, aside from it being critical, it is also one of the few daemons that have made it into the root file system (it's in /sbin, and not /system/bin like most others).

lmkd (Android L)

Android L uses another specialized core service class daemon called `lmkd`. It is defined in the /init.rc as follows:

Listing 4-lmkdrc: The `lmkd` definition in /init.rc

```
service lmkd /system/bin/lmkd
    class core
    critical
    socket lmkd seqpacket 0660 system system
```

The `lmkd` provides an interface to the kernel's **Low Memory Killer** (LMK) mechanism, which is an Androidism (i.e, a feature present in Android kernels, but not Linux ones). The LMK allows Android finer control over the Linux Out-Of-Memory (OOM) mechanism, which automatically kills tasks during memory pressure (Both OOM and LMK are described in detail in Chapter 19). Using the /proc/*pid*/oom_score_adj files, the `lmkd` can adjust the OOM score of processes, making them more or less "killable" - that is, prone to being killed when the system experiences memory pressure.

Like the other daemons discussed in this chapter, `lmkd` uses `epoll_wait` to simultaneously wait on input from multiple sockets. The main socket - /dev/socket/lmkd - is the one created for it by init, which is listening for connections. The only expected client is the `ActivityManagerService` (discussed in the next chapter), which uses this socket to notify the daemon which process needs to have its score adjusted. The `lmkd` also uses an output socket to log messages using `logd`, another recent addition in L (described in the next section).

Experiment: Observing `lmkd`

At the time of writing, the Android source code for L hasn't been made available (aside from a very limited preview, which hasn't proven helpful). The binaries, however, are available for both the Nexus 5 and the Android Emulator. It is therefore easy to reverse engineer them in a level of detail sufficient for this work. Both static analysis (i.e. disassembly) and dynamic analysis (runtime debugging) methods have been used. The method shown previously with `healthd` (using `strace`) proves its efficacy once again. Note that `lmkd` cannot be backported into KitKat, as it relies on the seqpacket sockets created for it by init to communicate with the frameworks.

Output 4-lmkds: Using `strace` to figure out `lmkd`

```
root@LEmulator:/# ls -l /proc/$lmkd_pid/fd | cut -c1-10,55-
lrwx------ 0 -> /dev/null
lrwx------ 1 -> /dev/null
lrwx------ 10 -> socket:[7360]           # /dev/socket/lmkd (listening)
lrwx------ 2 -> /dev/null
lrwx------ 3 -> anon_inode:[eventpoll]
lrwx------ 4 -> socket:[7364]            # /dev/socket/logdw (to logd)
lrwx------ 5 -> socket:[7653]            # /dev/socket/lmkd (to ActivityManager)
lr-x------ 8 -> /dev/__properties__
root@LEmulator:/# strace -p $lmkd_pid
epoll_pwait(3, {{EPOLLIN, {u32=3069216345, u64=37428954713}}}, 2, -1, NULL, 8) = 1
read(5, "\0\0\0\1\0\0\4\5\0\0\0\v", 52) = 12
openat(AT_FDCWD, "/proc/1029/oom_score_adj", O_WRONLY) = 6
write(6, "647", 3)                      = 3
close(6)                                 = 0
```

Looking at the above, you can see `lmkd`, like other daemon, blocks on the `epoll_wait` (FD 3), waiting for an event. The fd used for input - 5 - is the /dev/socket/lmkd, the other end of which is connected to the Android `ActivityManagerService`. Messages are variable length (up to 52 bytes), starting with a message type. Two message types have been observed:

Table 4-lmkdm: `lmkd` protocol messages

Type	Parameters
0x00000000	Integer array of parameters which `lmkd` writes to /sys/module/lowmemorykiller/parameters/minfree
0x00000001	PID to adjust (e.g. "\0\0\4\5" above for PID 1029), and oom_score_adj to set for it