David Merrick

CS 411

4 June, 2013

Writeup for Final Exam

The Android operating system is built on top of a Linux foundation. We have spent the semester studying in-depth the ways in which the Linux kernel schedules processes, schedules I/O, handles memory allocation, encryption, and how the kernel interacts with devices through drivers. I will be comparing and contrasting the implementation we used for class with the Android operating system in terms of process scheduling and I/O scheduling.

Overview:

The Android kernel is a modified version of Linux. Over the course of development, some changes were made. The number of changes is relatively small, however. There have been approximately 250 patches that take up about 3 megabytes of differences, comprising 25,000 lines of code. Some of this code is used to implement a flash filesystem, and some of it is to patch the network security [**?**]. Android is based on Linux 2.6.32. It has some power-saving features built in and excludes many libraries, shells, editors, and programming frameworks that ship with most distributions of Linux. On top of the kernel runs Dalvik, which comprises the Java virtual machine and many basic runtime essentials. Most Android apps run inside this VM [**?**].

The majority of changes that distinguish Android from other versions of Linux were not made to the kernel, but to user-space. Android is essentially a software stack built on top of Linux. The changes made to the kernel consisted mainly of a file-based shared memory system, Binder (an inter-process communication system), logger (a kernel logging mechanism optimized for writes), Paranoid Networking (used to restrict network I/O to certain processes), a driver for mapping large chunks of physical memory to user space, Viking Killer (a replacement Out-Of-Memory process killer to kill least recently used applications under low memory conditions), and wakelocks (a power management solution). Nearly all of these changes were implemented as device drivers, with very limited changes to core kernel code [**?**].

1 Process Scheduling

Overview:

A scheduler is the mechanism for sharing the CPU between multiple processes in an operating system. A policy is the behavior of the scheduler that determines what process runs when. Changes in policy can result in operating systems that are optimized for specific tasks (such as user responsiveness). Therefore, this is a critical aspect of an operating system [**?**](pg. 43).

In the version of the Linux kernel we were working with, the default scheduler was the Completely Fair Scheduler (CFS). The idea behind CFS is to model scheduling as if the system had an ideal processor that could multitask perfectly. In this kind of system, each runnable process n would receive 1/n of the CPU time. In this way, CFS assigns each process a proportion of the CPU. For example, in pre-CFS UNIX systems, if two processes were to run, one would be run right after the other and each would receive 100% of the CPU power. In CFS, the two processes would run simultaneously at 50% of the CPU power. CFS performs this by running processes round-robin style for very small timeslices, so that in a given period of time it will appear as though many processes are running in parallel [**?**](pg. 48-50).

CFS, however, is not the only type of scheduling available in the Linux kernel. Real-time scheduling means to schedule processes within timing deadlines. The Linux scheduler provides soft real-time behavior, meaning that it tries to schedule processes within timing deadlines (but does not make guarantees to always achieve this). Linux provides two real-time scheduling policies, FIFO and round-robin [**?**](pg. 64). Both round-robin and FIFO use runqueues, which are essentially queues containing process descriptors for runnable processes. Round-robin scheduling cycles through processes, running each for a pre-specified amount of time known as a timeslice. On a more technical level, this means placing the process descriptor at the end of the runqueue after its timeslice, then running all subsequent tasks until it reaches the end of the queue, at which time it will start the cycle again. FIFO (First In, First Out) scheduling runs each FIFO process of the same priority until it is finished, then runs the next process in the runqueue. It does this in order of when processes were placed in the runqueue. On a technical level, FIFO scheduling behaves almost identically to round-robin scheduling but with infinite timeslices [**?**].

Real-time scheduling depends on prioritizing processes. Processes with a higher priority run before processes with a lower priority, and processes of the same priority run one right after the other, round-robin style. Our Linux kernel used two different priority ranking systems. One system ranked processes based on their "nice value." This value ranged from -19 to 20 with 0 as the default. A larger nice value corresponds to a lower priority. Meaning that a process is being "nice" to other higher-priority processes by yielding control of the CPU. The second priority ranking system in Linux is the real-time priority. These numbers range from 0-99. Unlike nice values, higher priority values correspond to higher priority [**?**].

Android, as mentioned earlier, is basically a software stack built on top of Linux with very few changes to core kernel code. It uses CFS as its default process scheduler. However, it implements several user-space modifications for scheduling processes [**?**].

By default, Android applications run as processes in a single thread of execution. Every component of the application is run in the same process. However, different components of an application can be run as separate processes, and extra threads for each process can be created by the developer's specifications. This is all determined at runtime in the manifest file for each application. Only a certain amount of memory is able to be allocated for all processes, so the kernel may decide to put a process to sleep when memory is required for others. The kernel chooses which applications to put to sleep based on an algorithm that estimates their relative importance to the user. A process whose activities are visible on the screen, for example, takes a higher priority than one whose activities are not visible [**?**]. When two applications have the same priority, the process that has been at that priority longest will be killed first [**?**].

Android has separate process groups for background and foreground processes. Android uses a number of priorities so that background processes do not interrupt foreground ones. User interface threads run at default priority. Applications that are moved to the background have all their threads forced into the background process priority group. Background processes in Android are not allowed to take up more than 10% of the CPU time needed by foreground threads. This is implemented using a Linux feature called cgroups [**?**]. Cgroups was invented by engineers at Google. It is short for "control groups" and limits, accounts, and isolates resource usage of process groups [**?**].

The goal behind the default Android scheduling policy is responsiveness to the user. Android, like the Linux foundation it is built on, uses timeslices for scheduling processes. Processes can have a dynamic priority from 19 to -20, very low to very

high, respectively. Higher priority processes will get more CPU time when needed. At startup, Service processes are run in the background process group unless this has been specified otherwise. Background threads are given a priority level of 10 [**?**].

Processes are ranked in an importance hierarchy that consists of foreground, visible, service, background, and empty process types, ranked from highest to lowest, respectively. The foreground process type are processes that are required for what the user is currently doing. Generally only a few of these exist at once and they are killed only as a last resort. The visible process type are processes that do not have any foreground components but still can affect what the user sees. These are considered very important and will not be killed unless doing so is required to keep foreground processes running. The service process is a type that do not fall into the visible or foreground categories. These processes play music in the background or download data from the network, for example. The kernel keeps them running unless there is not enough memory to run them along with all foreground and visible processes. The background process type holds activities that are not currently visible to the user. Killing these types of processes will not have a visible effect on the user experience. An empty process does not hold any active application components. These processes exist simply for caching purposes so an application component will run faster the next time it is used [**?**].

2 Block I/O Scheduling

Overview:

Block devices are characterized by random access of fixed-sized blocks of data. The smallest addressable unit on a block device is the sector. All device I/O must be done in units of sectors, which are then mapped to blocks in memory [**?**](pg. 289-290). The Linux kernel provides an entire subsystem for block devices.

The subsystem dedicated to performing these I/O operations is called the I/O scheduler. Block devices maintain request queues to store their pending I/O requests [**?**](pg. 297). The I/O scheduler functions by managing this request queue, deciding the order of requests in the queue and when to dispatch each request to the block device. Seek time is the amount of time required to position the hard disk's head at the location of a specific block. One of the primary optimizations that can be made to block I/O performance is minimizing seek time. I/O schedulers sometimes do this by merging and sorting the requests in the queue. The entire request queue is kept sorted sectorwise so that all seeking along the queue moves sequentially [**?**](pg. 298-299).

There are several types of I/O schedulers built into the Linux kernel we worked with already and these schedulers are optimized for different purposes. The Deadline I/O Scheduler is designed to prevent starvation of certain requests (such as write operations starving reads) [**?**](pg. 300). The Anticipatory I/O Scheduler is designed to maximize throughput and is ideal for servers. The Completely Fair Queueing (CFQ) I/O Scheduler performs well in nearly all workloads and is the default scheduler in Linux [**?**](pg. 303). The Noop I/O Scheduler only performs merging of requests, maintaining the request queue in near-FIFO order. It is intended for block devices that are truly random-access, such as flash memory cards [**?**](pg. 304).

Like the Linux kernel we worked with, Android has the Deadline, Noop, Anticipatory, and CFQ I/O schedulers built in. It also uses Budget Fair Queueing (BFQ), Simple I/O (SIO), and V(R). BFQ makes use of budgets instead of timeslices. The disk is granted to a process until its budget (number of sectors) expires. This scheduler works well for USB data transfer, HD video recording/streaming due to less jitter, and achieves more throughput than CFQ on most workloads. The disadvantages are that a higher budget assigned to a process can increase latency for other processes. The Simple I/O scheduler is basically a hybrid of the Noop and Deadline schedulers. It aims to minimeze overhead and latency. Like Noop, it does not reorder or sort requests, resulting in minimized request starvation and high reliability. However, unlike Noop, it has relatively slow reads on random-access devices like flash drives. The VR scheduler imposes a fairness deadline on requests. It treats synchronous and asynchronous requests as the same. This results in a scheduler that is excellent for benchmarking, but has below-average performance at times due to fluctuations and is therefore the least reliable of the schedulers [**?**].

The default I/O scheduler in the Linux kernel we were working with was CFQ. The default in Android depends on which version of Android is running. For example, Cyanogen uses CFQ by default, while many versions use Noop. In the version of Linux we were using on the Squidly, CFS was a much better I/O scheduler than Noop because we were using a hard disk. So why would Noop be advantageous to use on Android? As briefly mentioned earlier, Noop actually performs very well with solid state block devices (such as flash memory), which do not depend on mechanical movement to access data. These devices do not need re-ordering of multiple I/O requests, so Noop reduces average seek time. Since flash memory is used universally on the mobile devices on which Android is installed, it makes sense that Noop would be the preferred I/O scheduler on many versions of Android [**?**].

3 Group Evaluation

I've been very satisfied with my group. Initially, we had one member, Ke Fan, who did not contribute, communicate, or show up to any of our meetings. So we asked that he be placed in the slacker group. But besides that, my other group members, Michael and Doug, have been great to work with. Despite juggling senior design projects and commitments in other classes, they were both always responsive to my texts and calls, always set aside time to finish the project, and we delegated the work very effectively. Doug was usually in charge of scheduling demo times and doing the Code section of our writeups, Michael would do the Implementation section of our writeups, I would do the Background section, and we all worked together to write the code for our projects.

References

[1] Daniel Bovet. Understanding the linux kernel. http://oreilly.com/catalog/linuxkernel/chapter/ch10.html.

[2] Tim Bray. What android is. http://www.tbray.org/ongoing/When/201x/2010/11/14/What-Android-Is.

[3] eLinux.org. Android kernel features. http://elinux.org/Android_Kernel_Features.

[4] farmatito. Speed up your system with the noop scheduler. http://forum.xda-developers.com/showthread.php?t=576004.

[5] gomo. Scheduling in android. https://github.com/keesj/gomo/wiki/AndroidScheduling.

[6] Android Developer Guide. Processes and threads. http://developer.android.com/guide/components/processes-and-threa

[7] Dianne Hackborn. Google+ post, dec 9 2011. https://plus.google.com/u/0/105051985738280261832/posts/XAZ4CeVP6DC.

[8] Optimus Kernel. Bfs vs cfs: Some personal observations on linux kernel performance. http://optimus-linux.blogspot.com/2012/03/for-some-time-now-i-have-stopped-using.html.

[9] Robert Love. What are the major changes that android made to the linux kernel? http://www.forbes.com/sites/quora/2013/05/13/what-are-the-major-changes-that-android-made-to-the-linux-ke

[10] Robert Love. *Linux Kernel Development*. Pearson Education, Inc., third edition, 2010.

[11] StackOverflow. Android process scheduling. http://stackoverflow.com/questions/7931032/android-process-scheduling.

[12] Wikipedia. cgroups. http://en.wikipedia.org/wiki/Cgroups.

[13] zhanjia. Governors, i/o schedulers, optimization tips. http://forum.xda-developers.com/showthread.php?t=1989824.