KallistiOS

A Miniature Operating System for Video Games

UMBC CMSC 421 - Spring 2016

Video Game Consoles

- Video game consoles are essentially locked down general purpose computers
  - The same kinds of components go into their makeup — from the CPUs on down to the optical disc drives
  - Usually largely non-upgradable/non-serviceable black boxes to users though.
- Two out of the three current-generation home video game consoles are essentially PCs — they use x86 CPUs, relatively standard GPUs, and share much of their architecture with normal PCs
  - The Playstation 4 even runs a (modified) version of a normal PC operating system (FreeBSD) and (at least originally) used the same boot loader that we've been using for our Linux VMs

Video Game Consoles

- Current generation consoles look a lot more like PCs, both in hardware and in their software stacks, but even older consoles had their similarities.
- It is only fairly recently that operating systems on these consoles have become so apparent to the consumer
    - We take for granted things like multiprocessing and built-in applications these days
- Older consoles usually had no user-facing operating system, or an extremely limited “BIOS” that handled things such as playing audio CDs and managing memory cards

Operating Systems on Past Video Game Consoles

- The “BIOS” of most older consoles isn’t really what one would typically refer to as an OS
  - Usually provides little to no services to applications running on the console
- It would be more apt to say that the games themselves on these consoles were the operating system
  - All hardware access is usually done directly by the game itself, potentially through libraries provided with a software development kit (although often in much older consoles, even SDK libraries weren’t a given)

Operating Systems on Past Video Game Consoles

- Older consoles were often very limited compared to what is expected today
  - Very little RAM
  - Slow CPUs
  - Very little or no writable secondary storage
- Many general-purpose operating system features we’ve discussed are either useless or detrimental to the user experience on such systems!

A Retrospective…

- Let us take a look at one particular video game console: The Sega Dreamcast
- Released in 1998 in Japan, 1999 in the US/Europe/Australia

Dreamcast

- Very limited hardware by today's standards:
    - CPU: 200 MHz (single core) Hitachi SuperH 4
    - RAM: 16 MB (essentially PC100 SDRAM)
    - Video: NEC/VideoLogic PowerVR 2DC (HOLLY) @ 100MHz, 8MB RAM
    - Sound: ARM7DI (nominally 22-25MHz, actual performance much worse) + Yamaha AICA (descendant of the SCSP) w/ 2MB RAM
    - No hard drive, all games run from Gigabyte Disc (GD-ROM) media.
    - Various accessories including memory cards, network adapters, etc.
- No real built-in OS — very basic shell for managing memory cards and playing audio CDs which provides very little in the way of services to applications
    - Only service provided is a set of system calls for accessing the GD-ROM drive

So… What?

No operating system was provided on the console — games functioned as their own OSes through SDK libraries!

Katana and Windows CE

- Two SDKs were available for this console to licensed developers
- The most widely used was Sega’s “Katana” SDK, named after the codename of the Dreamcast
    - Provided extensive hardware support and debugging functionality
    - No explicit threading model or support for loadable code modules, etc.
- Microsoft Windows CE was also available as an SDK
    - Based on the mobile OS that Microsoft had developed, with extensions ported from the PC version of Windows, like DirectX
    - Functioned much more like a traditional OS than Sega’s Katana and was much more familiar to PC developers
    - However, this had a detrimental impact on performance of games written with this SDK

Homebrew Development

- Early on, it was discovered that it was possible to run unlicensed code on the Dreamcast through a “vulnerability” in the console’s boot up sequence
- Specially written CD-Rs can be used to load code
- To this day, there is an active development community, even though the console was discontinued over 15 years ago

KallistiOS

- The most widely used homebrew "SDK" for the Dreamcast is known as KallistiOS
- More or less continuously developed since 2000
- Has been used many freely-available homebrew games, as well as commercially produced indie games
- Contains no Sega-copyrighted code — based completely on reverse engineering of the hardware
- Distributed in the form of a library which provides all functionality to user programs by statically linking to the kernel
    - User programs run in Kernel mode
- Originally written by largely by Dan Potter, now maintained primarily by me, with many other developers having contributed code

What is KOS?

- A “pseudo real-time OS”
    - Provides a monolithic kernel with the ability to dynamically load modules
- A fairly small codebase
- Hardware management
    - Interrupt handling, DMA, MMU support, threading, etc.
- Pseudo-POSIX layer
    - Virtual Filesystem, Pthreads, libc, BSD Sockets API, etc.
- Hardware abstraction layer
    - Video hardware access (including OpenGL-like functionality), and drivers for most other hardware internal to the system or available as an add-on

What KOS doesn’t do…

- Full POSIX-compliance
    - Provides basic functionality, eschewing features not useful/relevant to gaming
- Multi-tasking
    - Threads are provided, but not processes
- Memory protection
    - No process model means that tasks can overwrite each other at will
    - MMU support not on by default
        - MMU support is available, if you really want memory protection…

How is that an OS?

- Think back to the beginning of the semester…
- The two most important tasks of an OS?
    - Resource allocation
    - Acting as a control program
- Does KOS do these? Of course!

Structure of KOS

- KOS is a monolithic kernel, with the ability to use dynamically loaded modules
- Divided into several interdependent layers:
  - Hardware access layer
    - Including most platform-dependent pieces of the code, like hardware drivers, crt0, task switching, etc.
  - Virtual Filesystem, C library, and Pseudo-POSIX layer
  - Threading System
  - Network Stack and BSD Sockets

Programming with KOS

- Designed to be as simple for developers familiar with PC programming to pick up as possible
  - Includes several ported libraries and convenience functionality like an OpenGL-like video stack
- Direct hardware access, through the built-in functionality for those more experienced with the system or who want more performance
- Kernel is statically linked to user binaries
  - No built-in distinction between user programs and the kernel — everything runs in kernel mode for performance
- Uses a relatively standard compiler setup (GNU Binutils, GCC, Newlib libc/libm)
- Support for C, C++, Objective C, and SuperH Assembly programming languages
  - Lua and Python have also been ported to the system, however these do not include any direct hardware access and are generally meant for embedding into other code