Weenix OS

Nick Gaya

Spring 2014

1 Overview

My capstone project was an implementation of the Weenix OS. Weenix[1] is a small Unix-like operating system that has been developed over several years as a class project for CS 169. The project is compiled into x86 and runs in the QEMU emulator. Simple Unix binaries can be run within the operating system in user mode. (Figure 1)

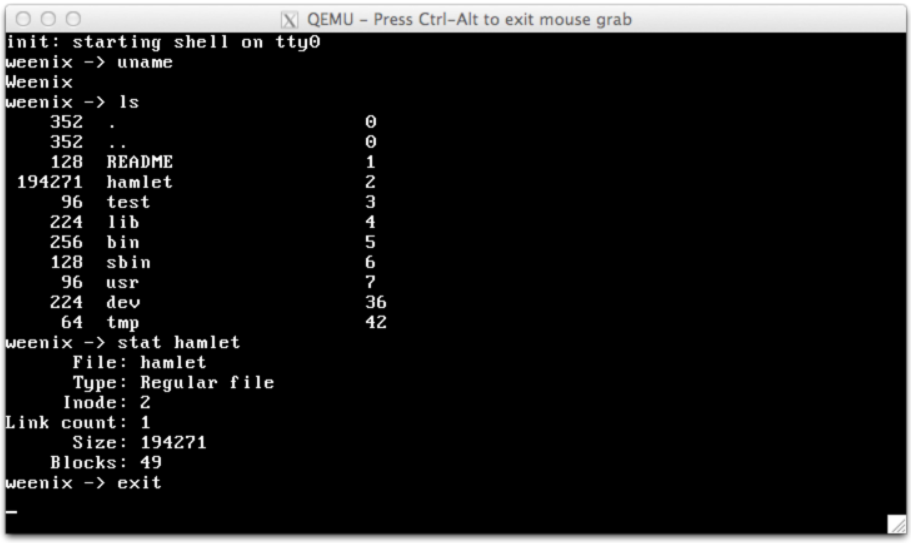

Figure 1: Weenix in action

[1] http://weenix.cs.brown.edu/mediawiki/index.php/Weenix

2 Architecture

2.1 Processes

Weenix supports the concepts of distinct processes and threads of execution, with one thread per process. This could potentially be extended to support multiple threads per process, although there is not currently a mechanism for user processes to make use of this functionality. The OS runs on a single-processor machine, simplifying some concurrency issues.

2.2 File system

File operations pass through several layers of abstraction. User processes access files via a virtual file system interface that provides functions such as opening files, reading and writing to file, and creating and removing directories. The VFS functions perform error checks and translate file paths into the corresponding file system nodes, then call down to the filesystem layer to perform core operations.

Weenix's filesystem is an implementation of the System V filesystem. The file system consists of a series of inodes representing files and directories. Each inode is associated with some number of disk blocks that hold the actual contents of the file.

File system operations usually involve looking up a given block of an inode, then reading or modifying that block. This process is mediated by the memory management system, described below.

In addition to on-disk files, Weenix also supports the concept of device files such as TTYs and `/dev/null`.

2.3 Memory management

The memory system divides the virtual and physical memory space into a series of fixed-size regions called pages. Pages of physical memory that are currently in use are associated with a data structure called a page frame. Memory objects are used to keep track of the page frames for an abstraction such as a file or other region of memory.

Each process is associated with a virtual memory map containing a list of mapped regions, each associated with a memory object. When a given virtual address is first accessed in user mode, the OS looks up the region containing the specified address in the VM map and uses the associated memory object to retrieve or create a page frame for the page in question. The OS then updates the hardware page table with the physical address of the page, or generates a segfault if no page table could be found, or if there is a permissions conflict (e.g. trying to write to a read-only memory region).