Nachos: Not Another Completely Heuristic Operating System

Tom Anderson
Computer Science 162
January 24, 1993

Project Overview

"I hear and I forget, I see and I remember, I do and I understand."
— Chinese proverb

The project for this course is to build an operating system from scratch. We believe that the best way for you to understand operating systems concepts is to build a real operating system and then to experiment with it. To help you get started, we have built a very simple, but functional, operating system called Nachos. Over the course of the semester, your job will be to improve the functionality and performance of Nachos; we expect that you will eventually modify or replace much of what we have written.

The project has five phases, corresponding to each of the major pieces of a modern operating system: thread management, multiprogramming, virtual memory, file systems, and networking.

Some phases build on previous phases; for example, the file system uses thread management routines. As far as possible, we have structured the assignments so that you will be able to finish the project even if all of the pieces are not working, but you will get more out of the project if you use your own software. Part of the charm of building operating systems is that you get to "use what you build" – if you do a good job in designing and implementing the early phases of the project, that will simplify your task in building later phases.

The end result of the project is to build a distributed application, for instance, a game such as "tic-tac-toe" or "battleship", with each player on a different computer connected by a network. But! before you do this, you have to build the operating system that the distributed application needs in order to be able to run, that is, you have to build the infrastructure for running distributed programs.

Part of the code we provide is a software emulation of a network of MIPS-like workstations. For instance, our code emulates turning on and off interrupts, taking exceptions and page faults, as well as the actions of physical devices (e.g., a disk, a console, and a network controller).

It is important to realize that while we run Nachos on top of this emulation as a user program on UNIX, the code you write and most of the code we provide are exactly the same as if Nachos were running on bare hardware. We run Nachos as a user program for convenience: so that we can use gdb and so that multiple students can run Nachos at the same time on the same machine. These same

reasons apply in industry – it is usually a good idea to test out system code in a simulated environment before running it on potentially flaky hardware.

In order to port Nachos to real hardware, we would have to replace our emulation code with a little bit of machine-dependent code and some physical machines. For example, in assignment 1, we provide routines to enable and disable interrupts; on real hardware, these functions are provided by the CPU. In assignment 4, we emulate the behavior of a physical disk; disk read and write requests instead go to a UNIX file and an interrupt is generated after some period of time. The details of how to make disk read and write requests varies tremendously from disk device to disk device; in practice, you would want to hide these details behind something like the Disk abstraction that we provide.

Unless you work for a really smart company, when you develop operating system software you usually cannot change the hardware to make your life easier. Thus, you are not permitted to change any of our emulation code, although you are permitted to change any of the Nachos code that runs on top of the emulation.

Nachos is coded in a subset of C++; a separate document covers the parts of C++ that you will need to know.

Finally, a former student had the following suggestions for doing well in this course, and since we agree with all of them, we include them here:

Read the code that is handed out with the assignment first, until you pretty much understand it. *Start* with the ".h" files.

Don't code until you understand what you are doing. Design, design, design first. Only then split up what each of you will code.

Talk to as many other people as possible. CS is learned by talking to others, not by reading, or so it seems to me now.