Nachos Overview

Operating Systems is one of the fundamental courses for a student who majors in computer software. A good way to obtain a deeper understanding of modern operating system concepts is to get your hands dirty, to read and analyze the code for operating systems and to see how it works at a low level, to build significant pieces of operating systems, and to observe their effects. Our course project, which is based on *Nachos*, provides you an opportunity to learn how basic concepts may be used to solve real problems.

1 Introduction

Nachos has an excellent balance between simplicity and realism. On the one hand, Nachos runs as a UNIX process instead of directly on bare hardware. Thus much tedious code dealing with real I/O devices is unnecessary keeping Nachos small enough; it is also easier for students to do experiments since they don't need to alter between development environment and Nachos. On the other hand, Nachos was originally developed for use on DEC MIPS systems (DEC was acquired by Compaq years ago, which was acquired in turn by HP recently.) and user programs are in binary format, making Nachos real enough avoiding the sense of "toy". Nachos includes a MIPS simulator to execute MIPS instructions in user programs. Some other instructional operating systems use their own pseudo instructions, which is interpreted by an interpreter. With these systems, Students have more sense of working on an application than on an operating system.

Normally, an operating system runs on the machine that it manages. Nachos is unusual in that the operating system runs "side-by-side" with the simulated machine that it controls. The program `nachos` implements both the machine simulation and the operating system. Understanding this "side-by-side" relationship between the simulated workstation and the operating system is critical to understanding Nachos.

1.1 The architecture of Nachos

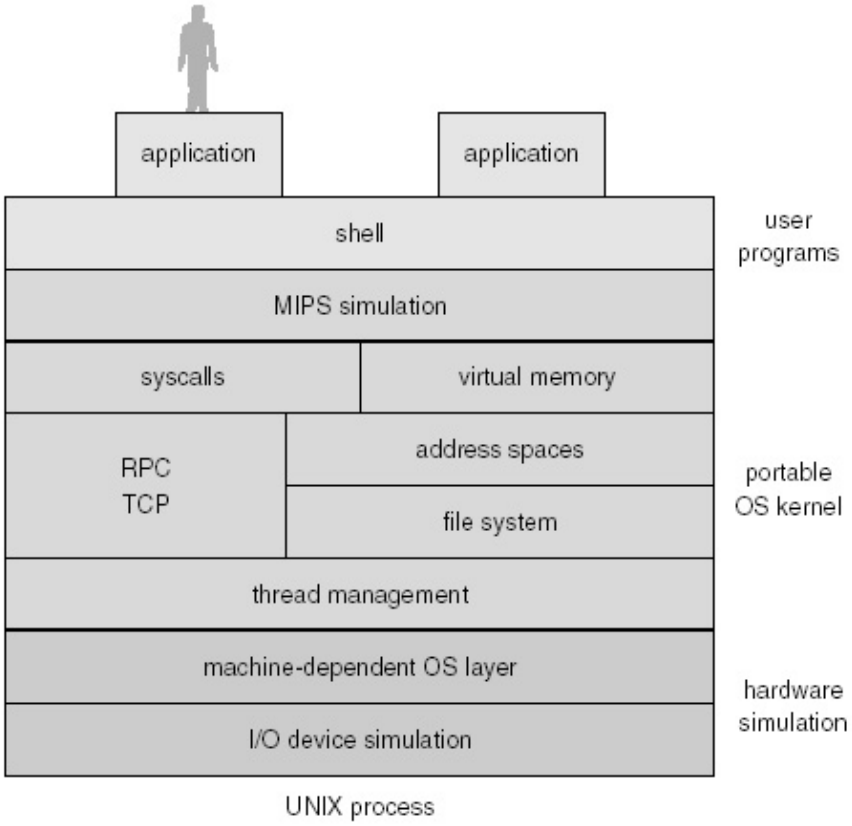

Figure 1: How the major pieces in Nachos fit together

1.2 MIPS simulator

The MIPS simulator includes the following components:

- Timer

  The timer acts like a periodic alarm clock. Each time the alarm goes off a timer interrupt is generated. Note that the timer runs in simulation time, not in real time. See `interrupt.h` for a discussion of how simulation time elapses.

- Disk

  The disk stores persistent data. The simulated disk is a very simple one. It has one recording surface. The surface is divided into tracks, and each track is divided into sectors. The operating system may issue requests to read or to write disk sectors. (Each request reads or writes a single sector.) Read and write requests are asynchronous. When a request is completed, a disk interrupt is generated.

  The simulator stores the contents of the simulated disk in a Unix file called `DISK`.

- Console

  The console simulates a display and a keyboard. The operating system may issue requests to put characters to the display, or to read characters from the keyboard. The requests are asynchronous. When incoming data is available, or when outgoing data has been sent to the display, interrupts are generated.

  By default, the console input is taken from the standard input of the `nachos` program, and the console output is sent to standard output. Thus, console output should appear on your (real) display, and keystrokes on your (real) keyboard are taken as console input. Console input and output can be redirected to files using `nachos` command line arguments.

- Network

  The network provides a means of communication among multiple independent simulated MIPS systems. The operating system can use a simulated network to send and receive small, fixed-length messages. Message delivery is unreliable - messages may be lost. Message sending and receiving is asynchronous. When data has been sent or when data is available to be received, interrupts are generated. The network simulation is implemented using Unix-domain sockets. See `network.h` and `network.cc` for more information.

- MIPS Instruction Processor

  The processor is responsible for simulating the execution of user programs. The processor has a set of registers and a small (simulated) physical memory for storing user programs and their data. The contents of the registers and of memory can be read and changed by the operating system.

  When the user program executes a system call instruction, or if a program error (such as divide-by-zero) occurs, an exception is generated.

- Interrupts

  This component simulates the interrupt and exception mechanism. Each of the other components of the simulated machine may cause interrupts or exceptions to occur. When one occurs, the hardware transfers control to an interrupt handler (or exception handler), a function that is part of the operating system. The operating system must provide a handler for each type of interrupt or exception.

1.3 Solaris/SPARC port

On the Sun SPARC version Nachos, the kernel and machine simulator would run on natively but the user level environment is still a MIPS in little endian format.

Why not port the simulator for running native programs for SPARC? Because it would take much more work to complete the port and change the CPU simulator to simulate the SPARC instruction set, which is not a RISC architecture like MIPS. Keeping the MIPS CPU causes a few problems:

1. You have to generate user-level code for the simulated machine. If you have a heterogeneous environment with both SPARC and MIPS workstations, this isn't so hard – students only need to compile a few small user programs. But if you only have Sparcstations, you need to get gcc to cross-compile to the DEC MIPS.

2. The Nachos kernel runs runs native mode while the user programs runs on the simulated CPU. This is a little weird on the SparcStation because the user programs are using little endian and the kernel is using big endian. Some information (such as the argv[] array) that is passed between the kernel and the user though user memory must be byte swapped. (Unfortunately, this isn't as easy to fix as simply cross-compiling to the SGI MIPS, which is big endian; in a few places, the simulation assumes little endian format. We're working on fixing this.)

2 Installation

2.1 Cross compiler

A cross compiler is a compiler which runs on one platform and produces code for another, as opposed to a native code compiler which produces code for the platform on which it runs. For our course project, we need a cross compiler running on Solaris but generating code for MIPS platform. To install and configure a cross compiler may be difficult for someone new to Unix. I have installed a cross compiler under `/home/cslab/public/mips-sparc.solaris-xgcc`. You may simply add this path into your `PATH` environment variable, or download `.bashrc` from the course web site and put it in your home directory. If you use shells other than bash (Bourne Again SHell), you have to figure out yourself.

2.2 Nachos

Once the GNU cross compiler is available, it is time to install Nachos itself. Nachos was originally developed for use on DEC MIPS systems (DEC was acquired by Compaq years ago, which was acquired in turn by HP recently.). People later ported it to Solaris/SPARC platform, that is what we have in UNIX lab, but since every school has different system configuration environment, it is not easy to set up Nachos on our machines based on the available Nachos-Solaris packages all over the Internet. To help avoid the error-prone procedure, I have successfully

build a Nachos package, which is both available on the course web site and locally in Solaris file system. The following procedure shows how to obtain and build the Nachos package:

- Copy the package to `~/os-proj`:

```
$ cd
$ mkdir os-proj
$ cd /home/cslab/phd/jniu/nachos/archives/ccny/
$ cp nachos-3.4-solaris.tar.gz ~/os-proj
```

- Uncompress the package to the current directory:

```
$ gunzip nachos-3.4-solaris.tar.gz
$ tar xvf nachos-3.4-solaris.tar
```

- Build Nachos:

```
$ cd nachos-3.4/code
$ make
```

3 Nachos directory structure

After you uncompress the Nachos package, a directory named `nachos-3.4` appears in the working directory. Its subdirectory `code` contains all source code and configuration files for Nachos. The original Nachos package also contains `doc` and `c++example` subdirectories. To save space, they have been removed from our customized package. In `nachos-3.4/code`:

- `bin`

  contains the source code for `coff2noff`, which converts a normal MIPS executable into a Nachos executable. *COFF*, or *Common Object File Format*, is the executable and object file format used by Unix System V Release 3, and *NOFF*, short for *Nachos Object File Format*, is an object file format exclusively used for Nachos user programs.

- `machine`

  The machine simulation. Except as noted in `machine.h`, you may not modify the code in this directory.

- `threads`

  Nachos is a multi-threaded system. Thread support is found here. This directory also contains the `main()` routine of the `nachos` program, in `main.cc`, where you may also find the information about what parameters you may use to run `nachos` after you build it. In fact, this is the best place to start reading the Nachos code, since this is where it all begins.

- `userprog`

  Nachos operating system code to support the creation of address spaces, loading of user (test) programs, and execution of test programs on the simulated machine. The exception handling code is here, in `exception.cc`.

- `filesys`

  Two different file system implementations are here. The "real" file system uses the simulated workstation's simulated disk to hold files. A "stub" file system translates Nachos file system calls into UNIX file system calls. This is useful initially, so that files can be used (e.g., to hold user programs) before you have had a chance to fix up the "real" file system. By default, the "stub" file system is build into the nachos program and the "real" file system is not. This can be changed by setting a flag in the Nachos makefile.

- `network`

  Nachos operating system support for networking.

- `test`

  User test programs to run on the simulated machine. As indicated earlier, these are separate from the source for the Nachos operating system and simulation. This directory contains its own `Makefile`. The test programs are very simple and are written in C rather than C++.

4 Development and maintainance

4.1 Access to UNIX lab remotely

You may always access our UNIX lab remotely, say from your home, through `ssh`. `ssh` is a program for logging into a remote machine and executing commands in a remote machine. It is intended to replace `rlogin` and `rsh`, and provide secure, encrypted communications between two untrusted hosts over an insecure network. Thus you may stay at home physically while working on your Nachos project on our Solaris machines.

If you use Linux, normally you already have `ssh` at hand for use. For Windows, you download corresponding implementation from `http://www.ssh.com/`.

4.2 Sharing files among project group members

You are required to do this project together with your classmates, so naturally you need to consider how to share files under development among your group members.

There is no single best way to do this. A possible option is to set up a user group in the Solaris system for your group, and by assigning proper access rights to your project files, you may allow only your group members to access. Unfortunately, we do not have someone to help me to do this, and I am not an administrator of the system myself.

If you are familiar with CVS (a code management system that helps program developers keep track of version history, releases, etc.), you may take advantage of some websites which provide free services for developing open-source projects, such as `www.sourceforge.com`, and `savannah.nongnu.org`.

If you are not familiar with CVS, or simply do not want to try it, I am afraid the only option left is that every member in your group has a private source code directory and somehow you have to agree upon who is allowed to update which files. However, technically a member can do whatever he/she wants on his/her directory (:-(This is not really sharing at all). Whenever necessary, files may be transferred between you and your partners by email or other mechanisms.

4.3 Archiving

It is a good habit to backup your current development files so that you will not lose much if you delete your project files by mistake. You may use `tar` and `gzip` to generate a compressed file from your project directory, and name it with appropriate date information. The process is as follows:

```
$ cd nachos-3.4/code
$ make clean
$ cd ../..
$ tar cvf mj-nachos-20030922.tar nachos-3.4
$ gzip mj-nachos-20030922.tar
```

You will see a file named `mj-nachos-20030922.tar.gz` is generated in the current directory.

To restore files from an `.tar.gz` archive, do the following:

```
$ gunzip mj-nachos-20030922.tar.gz
$ tar xvf mj-nachos-20030922.tar
```

Thus a directory named `nachos-3.4` is created containing all the project files.

4.4 UNIX file permissions

Every Unix file has an owner and a group. You can find out the groups of all of the files in a directory by using the command `ls -l` in the directory. Each file will be listed, along with its owner and group, access control, and some other information. For example,

```
$ pwd
/home/cslab/phd/jniu/mips-sparc.solaris-xgcc
$ ls -l
...
-rwxr-xr-x   1 jniu     guest     218592 Sep  5  1999 gcc
...
```

The 10 characters at the far left describe the access permissions of the file. The first (leftmost) character is `-` if the file is a regular file, and `d` if the file is a directory. The remaining 9 characters are interpreted in groups of three. The first group of three describes the access permissions of the *owner* of the file, the next group of three describes the access permissions for *members of the file's group*, and the last group of three describes the access permissions for *everyone else*.

There are three characters in each group because there are three types of permissions one can have for a file: *read permission*, *write permission*, and *execute permission*. You need read permission to read a file, write permission to change a file, and execute permission to execute a file (if it is an executable program).

The example above shows that `gcc` is a directory, owned by `jniu` who belongs to group `guest`. The owner of the file has all three permissions. The members of the file's group and everyone else have read and execute permissions but not write permissions.

You can change a file's permissions using the `chmod` command. Run `man chmod` for more information. Here are some examples:

- `chmod g+r` file

  adds read permission for members of the file's group (`g` = group)

- `chmod o+w` file

  adds write permission for everyone else (`o` = others)

- `chmod o-r` file

  removes read permission for everyone else

- `chmod u-x` file

  removes execute permission for the owner(`u` = user/owner)

You may also need `chown` and `chgrp` to change the owner and the group the files belong to.

4.5 `make`

`make` is a tool to automate the recompilation, linking etc. of programs, taking account of the interdependencies of modules and their modification times. It reads instructions from a "makefile" which specifies a set of targets to be built, the files they depend on and the commands to execute in order to produce them.

4.6 `gcc/g++`, `ld`, and `as`

5 Projects

5.1 Project 0: Refreshing C/C++

This project is not based on Nachos, but helps you to refresh C/C++ knowledge so as to ease your Nachos development.

Please check the course web page for details.

6 Solaris basics

6.1 File systems

From the point of view of end users, the file system of Solaris is quite different from that of Windows. In the former, there is only one root directory, that is the root of the directory tree, while the latter may have multiple root directories, like `C:\`, `D:\`, etc.. In Solaris, the file system on a device have to be `mount`ed onto the only directory tree before it is available for access.

6.2 Redirection

6.3 Editors

6.3.1 `vi`

6.3.2 `emacs`

6.4 Commands

6.4.1 `man`

To know how to use a specific command, whose name is already known, you may always try

```
% man command
```

to obtain the usage of `command`. `man` is a command that finds and displays the usage pages associated with specified commands.

6.4.2 `showrev`

returns system information about the current host.

```
css4c3% showrev

Hostname: css4c3
Hostid: 83181f11
Release: 5.8
Kernel architecture: sun4u
Application architecture: sparc
Hardware provider: Sun_Microsystems
Domain: cs.ccny
Kernel version: SunOS 5.8 Generic 108528-18 November 2002
```

Another command that may return similar information is **uname**.

6.4.3 `cp`

copy files

6.4.4 `cat`

concatenate and display files

6.4.5 `more`

Browse or page through a text file

6.4.6 `ls`

list contents of directories

6.4.7 `mv`

move files and directories

6.4.8 `rm`

remove files

6.4.9 `mkdir`

make directory

6.4.10 `rmdir`

remove directories

6.4.11 `pwd`

print working directory

6.4.12 `cd`

change working directory

6.4.13 `chown`

change ownership of a file

6.4.14 `chmod`

change the permissions mode of a file

6.4.15 `find`

6.4.16 `grep`

6.4.17 `tar`

6.4.18 `df`

displays the current usage information on disk device in 512 byte blocks.