NachOS 101

Bradford Hovinen
David Pariag
School of Computer Science
University of Waterloo

18 January 2004

Outline

1. Installing and Building NachOS
2. NachOS Directory and File Structure
3. NachOS Architecture
4. NachOS Executable Files and Address Space
5. Debugging Tips
6. Collaboration Strategies
7. Assignment Submission Information
8. Design Document
9. Testing Strategies
10. Testing Document

Installing and Building NachOS

- When you know your group number, make sure every group member creates a directory `~/cs350_<group>` where `<group>` is your group number.
- From the CS student environment, use `install_nachos` to install NachOS in your account
- Go to `code/build_solaris` and type `make` to build NachOS
- NachOS should compile and run on GNU/Linux as well; use the directory `code/build_linux`
- Make sure you have your assignment running on the CS student environment before submitting.

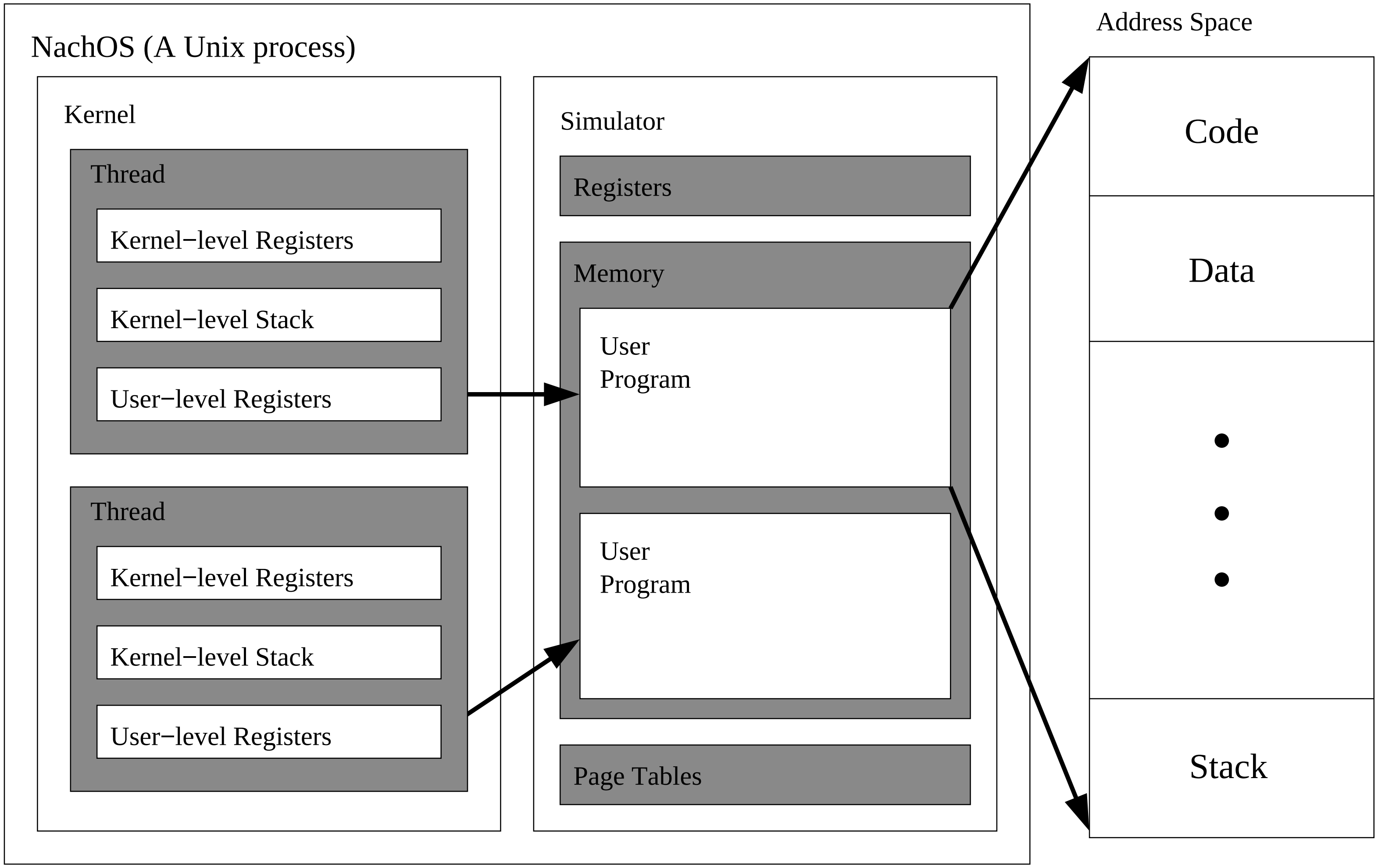
NachOS Directory and File Structure

`code/filesys`	Filesystem (used in A3)
`code/lib`	Library routines
`code/machine`	MIPS simulator and simulated hardware
`code/network`	Networking (don't worry about this)
`code/test`	Test suite (put your tests here)
`code/threads`	Heart of the kernel – scheduler, etc.
`code/userprog`	Support for user-level processes

NachOS Architecture

- NachOS kernel is a normal (UNIX-level) process
- Processes under NachOS are run by the MIPS simulator
- By “kernel-level” we refer to the *NachOS* kernel
- By “user-level” we refer to processes running under NachOS
- NachOS kernel has a complete threading library
- Each (user-level) thread under NachOS has a corresponding kernel-level thread
- Thus each (user-level) thread under NachOS has two sets of registers and two stacks: one under the MIPS simulator and one at the kernel level
- Be careful about which entity you’re talking about!

NachOS Architecture (cont)

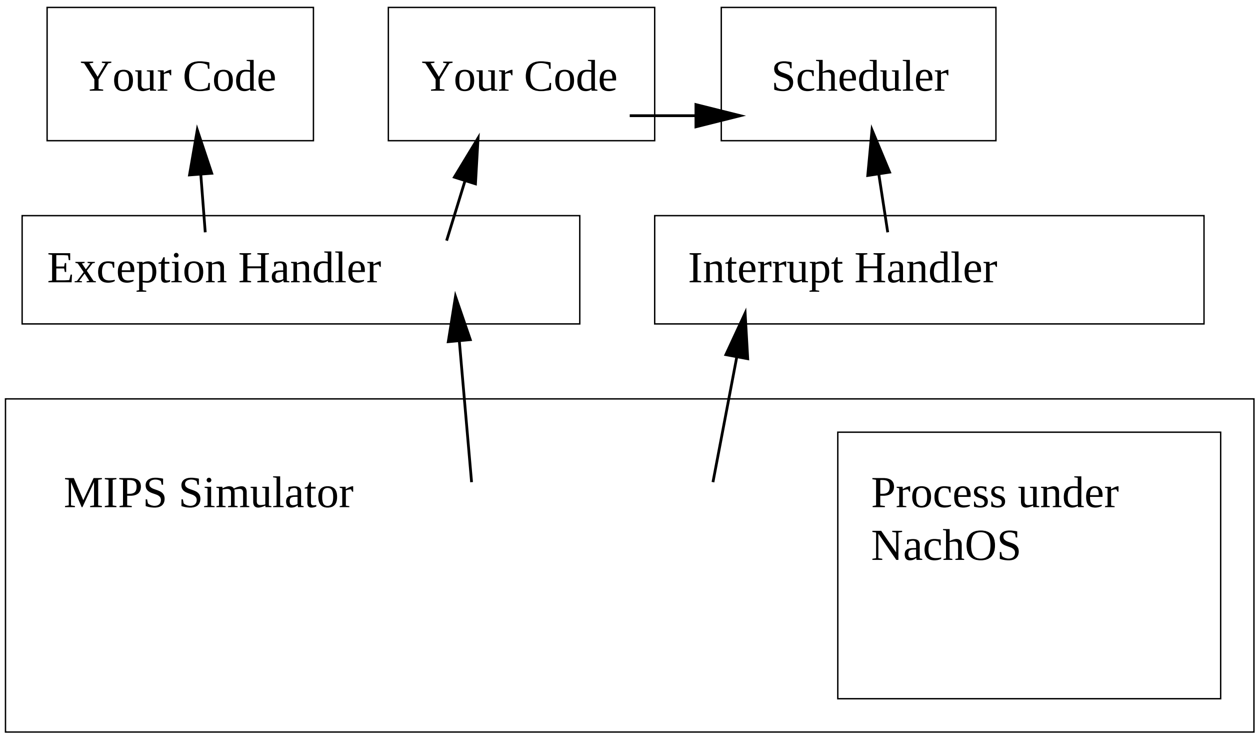


NachOS Architecture (cont)

- MIPS Simulator runs as a main event loop, invoked with `Machine::Run`
- Kernel code gets called from the simulator through (simulated) exceptions and interrupts
- Interrupts cause the simulator to call the appropriate interrupt handler
- Exceptions and system calls cause the simulator to call the exception handler (`userprog/exception.cc:ExceptionHandler`)
- Returning from the interrupt handler or exception handler returns control to the simulator
- `Machine::Run` gets called *once* per thread – you should not call it yourself

NachOS Architecture (cont)

- Kernel stack space is limited – don't allocate huge items on the stack
- The NachOS kernel is not preemptible – interrupts can only happen from within the simulator



NachOS Executable Files and Address Space

- NachOS uses an executable format called NOFF
- NOFF file is divided into sections:
  - .code

    The program instructions that the simulator will execute
  - .initdata

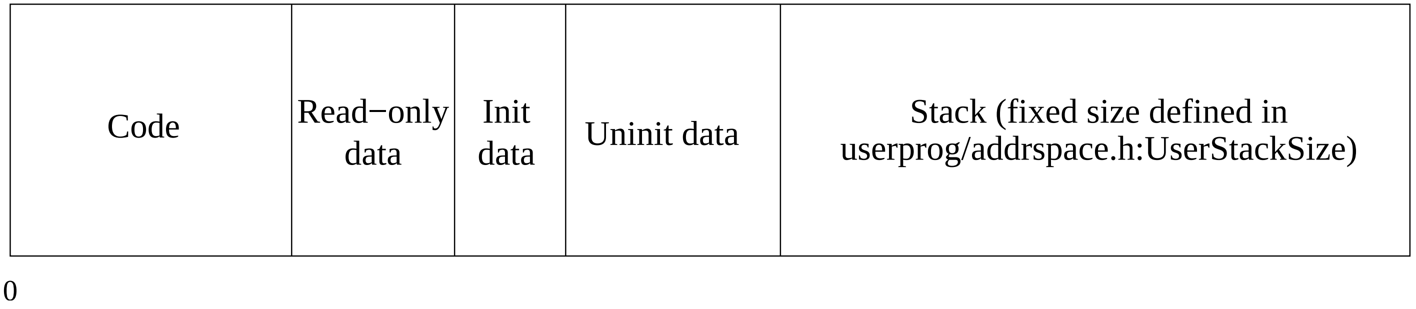
    The initialised data that hold predefined variable values (e.g. `static int a = 20;`)
  - .uninitdata

    Uninitialised data; these are not read from the file but are initialised to zero by the kernel (e.g. `static int a;`)
  - .rdata

    Read-only data (e.g. `char *tmp = ‘‘My String’’;`)

NachOS Executable Files and Address Space (cont.)

- NachOS user programs are linked to COFF format using a linking script that forces sections to be page-aligned
- A program (distributed with NachOS) called `coff2noff` converts the COFF file to a NOFF file
- Current address space layout is as follows:



Code

Read−only data

Init data

Uninit data

Stack (fixed size defined in userprog/addrspace.h:UserStackSize)

0

- You may need to modify this layout in future assignments
- If you change the way a program loads (e.g. adding dynamic loading in A2), you should make sure that *each* of these sections still works.

Debugging Tips

- NachOS programming involves C and C++ – be aware of the memory model!
- Most segmentation faults and bus errors are the result of memory allocation problems
- Warning: There are things you can do in Java but not C or C++

Debugging Tips (cont)

Examples of bad code:

```
char *f() {
  char array[20];
  // Do something with array
  return array;
}
```

```
char *f() {
  char *s;
  strcpy (s, "My text");
  return s;
}
```

Debugging Tips (cont.)

- Learn GDB! (see `http://www.gnu.org/software/gdb/documentation/`)
- If that's too scary, learn DDD! (see `http://www.gnu.org/software/ddd/`)
- If you see a crash in new or delete, you probably corrupted the memory allocator data structures (e.g. you walked off the end of an array, used memory that was already freed, etc.)
- On GNU/Linux systems, you can debug memory problems with Electric Fence (see `http://perens.com/FreeSoftware`)
- Also, look at Valgrind (see `http://valgrind.kde.org/`)
- We're seeing if we can get these and Purify on CSCF-administered machines
- If the above fail, see the TAs/instructors

Collaboration Strategies

- You need to share files among your group members
- Best way is to use CVS: see `http://www.student.cs.uwaterloo.ca/~cs350/W04/common/cvs.html`
- We recommend *against* copying files between group members, creating symlinks, giving all group members write access to the project directory, etc.
- Be careful about the account you use to submit the assignment – don't submit the wrong code!

Assignment Submission Information

Be careful about permissions – make sure cs350asst.zip is world-readable and its directory and all ancestors are world-executable!

The commands to do this are as follows:

In the directory where your assignment submission is located:

```
chmod o+r cs350asst.zip
```

Then, for that directory and all of its ancestors (back to your home directory):

```
chmod o+x .
```

Remember: That zip file is the only copy of your assignment.

- Do *not* modify or remove it after submitting it.
- Do *not* use the submission script after the deadline until after the assignment has been marked.

Design Document

- We (the TAs) are looking for answers to specific questions about your design
- We will tell you many of the questions we have for each assignment
- Divide your document into sections corresponding to the cover sheet
- Do the same with your one-page revision
- Avoid rambling, restating the obvious, etc.
- Proofread your document – TAs may deduct marks for grammar/spelling/usage errors!

Testing Strategies

- Scour the assignment description for every required behaviour.

  E.g. Such-and-such system call should return foo on success and bar on failure

- Think of all the ways a process can send invalid data to the kernel

  E.g. `Create(NULL);`

- Think of how the different components of the OS interact

  E.g. `Read` or `Write` across page boundaries

- Think of different scenarios

  E.g. A given page is not in memory

Testing Strategies (cont)

- Test limits that exist on your system

  E.g. A filename cannot be more than n characters long

  E.g. A process cannot have more than m files open at one time

  Note that, in any practical situation, some limits must exist. You should define them clearly, document them, and test them.

- Try to write some "stress-tests"

Remember: Marks for testing and implementation are separate! So you can get marks for testing something that isn't working or even implemented.

Testing Document

- We want to know two things
  1. How to run your tests
  2. What each test is testing
- Write your tests to be self-explanatory when run so you don't need a lot of external documentation
- In the document, a table layout is recommended
- Try to break your document down according to the sections in the cover sheet