NachOS Tutorial

J-F. Méhaut
UJF-CEA/LIG

Introduction

In teaching operating system at undergraduate level, it is very important to provide a project that is realistic enough to show how real operating systems work, yet simple enough that the student can understand and modify it in significant ways.

Tom Anderson

Washington University

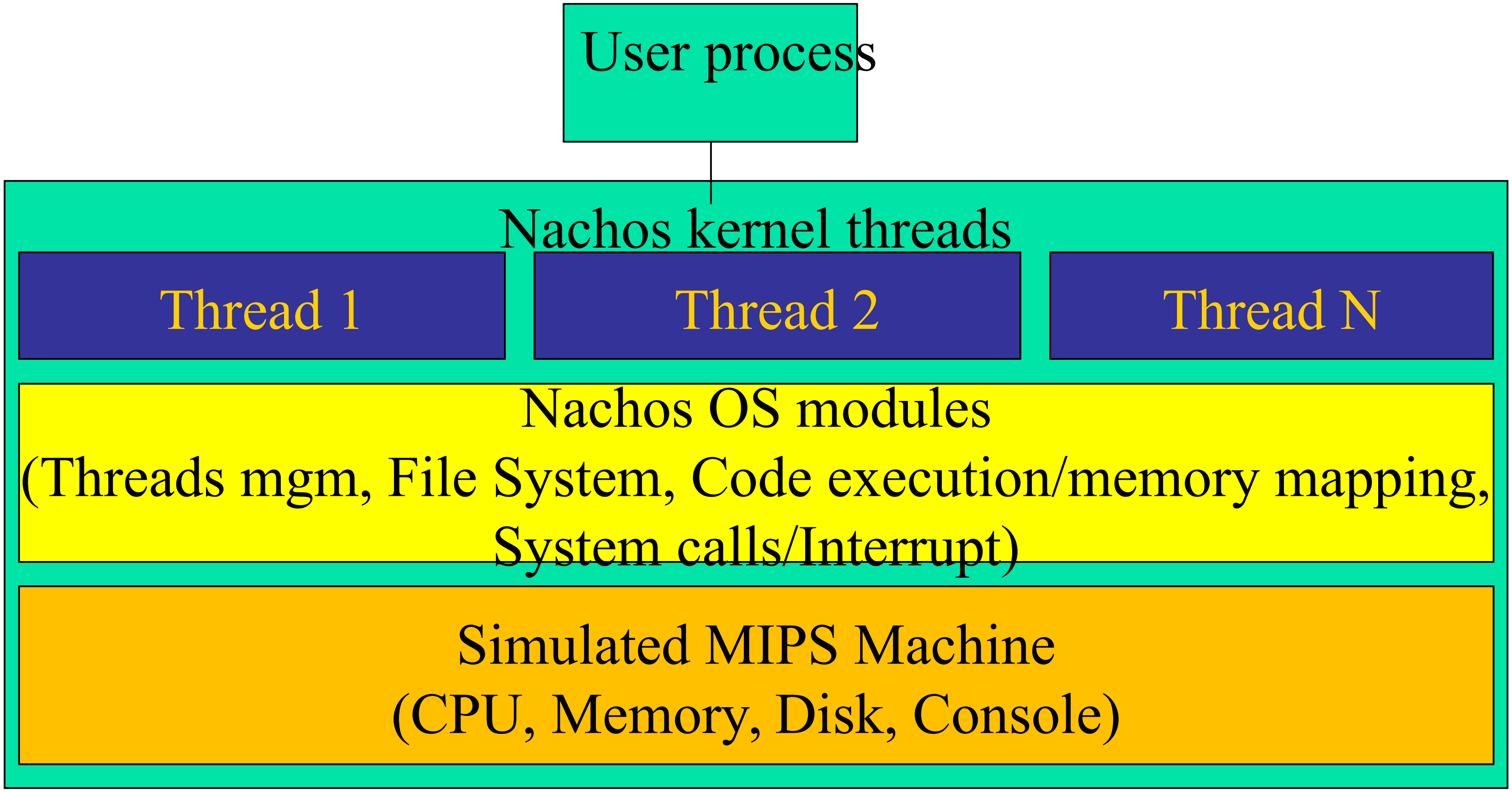
NachOS: a chinese proverb

- I hear and I forget
- I see and I remember
- I do and I understand

What is NachOS?

- Nachos is an instructional operating system
  - Designed for use in operating system classes
  - Allow students to modify an almost real OS
  - Has the same components of a real OS
    - Threads
    - Memory Management
    - File System
    - Network
- Kept small to be easy to modify
  - Small and easy compared to real OSes

What is NachOS?



What is NachOS

- NachOS runs as a Normal Unix Process
  - Single binary image
  - Make changes then just recompile
  - Simulate underlying hardware

- Not Another Completely Heuristic Operating System

Why use NachOS?

- To really see what OS is all about...

- NachOS allows students to learn in a hand on environment
    - Take lessons from class and implement them
    - Look under the hood of a realistic OS
- By implementing concepts you learn how things really work
    - Important addition to the theoretical knowledge from class

Why use NachOS?

- Implement useful things without writing million of lines of code...
- NachOS and project designed to minimize the amount of busy work
  - Maximize the learning/time spent ration
- Highly modularized design
  - Keeps changes for a project relatively localized
- Code you write early on will be used in later projects

Why use NachOS?

- Because it is simple!
- Real Oses are huge
  - Generally not designed for readability
  - Never designed for simplicity
- In NachOS, you can understand the code!
- NachOS allows you to make the design decision between performance and simplicity
  - But remember, correctness always comes first

Why use NachOS?

- Did I say it is simple?

- NachOS gretly simplifies the development process
  - Make changes, then recompiles (no reboots)
  - Can run under the debugger (GDB)
  - Deteministic/repeatable behavior

Why use NachOS?

- Answer to basic questions like
  - How does the OS start a thread? A process?
  - What happens on a system call?
  - How does address translation work?
  - What data needs to be written to disk on file creation?
  - How deos the OS interface with I/O devices?
  - What FS structures are stored on disk? In Memory?
  - What happens on a thread context switch?
  - How do all the pieces of an OS fit together?

Projects

- Project 1: Getting started
- Project 2: Systems calls and I/O functions
- Project 3: Multithreading
- Project 4: Virtual memory and multiprograming
- Project 5: File System
- Project 6: Networking

Project 1: Getting Started

- This is what you are supposed to do during the first week
- Install, compile and test the installation
- Understand: read the code, get into the code structure and understand NachOS organization

- This is a difficult project because it is all so new
- Without this first effort, the next projects will be rather complex

Project 2: System calls and I/O

- In NachOS, you can execute user programs
- The problem is that the initial version of NachOS does not allow user IO
  - The following proram does not run

```
#include "syscall.h"
    void print (char c, int n)
     {
        int i;
        for (i=0; i<n;i++)
             PutChar (c+i);
        PutChar ('\n') ;
     }

     int main ()
     {
        print ('a',4) ;
        Halt () ;
     }
```

Project 2: Goals

- Understand the system calls
- Implement system call for user I/O

- This is a working lab and you are supposed to get it finished at the end of second week

Project 3: Multithreading

- You know what a multithreaded program is
  - Several threads execute in the same process, sharing data and code, executing in parallel
  - Well, this is not possible with the initial version of NachOS

Project 3: Multithreading

- Here is one thread...

```
#include “syscall.h”
    void print (char c, int n)
     {
      int i;
      for (i=0; i<n;i++)
        PutChar (c+i);
      PutChar ('\n') ;
     }

     int main ()
     {
       print ('a',4) ;
       Halt () ;
    }
```

Project 3: Multithreading

- Two threads
  - main thread
  - another thread

```
#include “syscall.h”
    void print (char c, int n)
     {
      int i;
      for (i=0; i<n;i++)
         PutChar (c+i;
      PutChar ('\n');
     }

     int main ()
     {
       param=createParam(b,10);
       CreateThread(print,param);
       print ('a',4) ;
       Halt () ;
      }
```

Project 3: Goal

- Implement multithreading
  - Understand how the address space of a process is organized and managed
  - Implement sharing of the address space for multiple threads
  - Be able to execute programs with 2,3… threads

Project 4: Virtual memory

- The initial version of NachOS may launch only one process
- The memory management is very simple
  - Direct use of physical memory and hence of physical addresses
  - There are the bases for paging

Logical page	frame
0	0
1	1
2	2
3	3

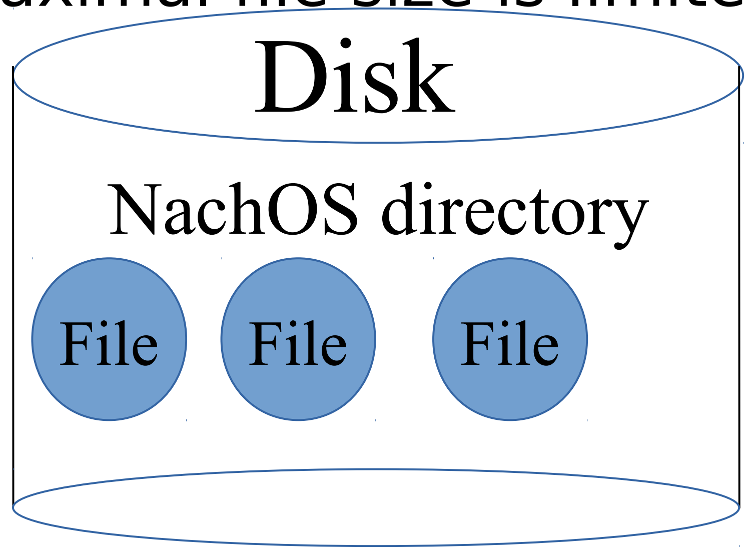
Projects 4: Goals

- Implement paging
  - To be able to load each program anywhere in physical memory
  - Non contigous storage of processor memory space in physical memory
  - Implement malloc/free in NachOS

- Implement multi-programming
  - Launch multiple processes
    - Manage their respective address spaces
    - Implement the fork system call
    - Implement a shell

Project 5: File System

- NachOS comes with a very simple file system
  - 1 directory
  - 10 files
  - Maximal file size is limited



Project 5: Goals

- Understand File Management
  - File Headers
  - Directory Management Structure
  - Disk Space Management
- Implement Tree Directories
  - Implement . And ..
  - Implement File Paths
    - Current Directory
  - Increase Max File Size (with I-node)

Project 6: Networking

- Is it possible to launch several machines and make them communicate?
  - The protocol is not reliable (some messages are lost)

Project 6: Goals

- Implement a reliable communication protocol
  - TCP/IP like
- Implement a file transfer protocol (ftp)
- Implement process migration

Outline

- Directory & File Structure
- Threads & Synchronization
- Unix vs. Kernel vs. User Programs
- MIPS Simulator & Nachos
- Address Spaces & Executables
- Common Problems

Directory & File Structure

- code/
  - filesys/
  - lib/
  - machine/
  - network/
  - test/
  - threads/
  - userprog/

Directory & File Structure

- code/filesys/
  - Holds implementation of both stub and real file system

Directory & File Structure

- code/lib/
  - Holds the class library and debug routines.
  - Learn and make good use of the Nachos class library (list, hash table, etc.) and debug facilities.
  - Avoid the STL (Standard Template Library) as it incurs too much disk space.

Directory & File Structure

- code/machine/
  - Holds the MIPS simulator that Nachos uses to execute user programs
  - Do NOT change anything in this directory
  - Allowed to change a few constants such as the NumPhysPages (in machine.h)
  - Familiarizing yourself with the simulator and the flow of control will help in debugging and design

Directory & File Structure

- code/network/
  - Holds the networking code for Nachos
  - Doesn’t interfere with the working of Nachos
  - The networking code does create one thread call “postal worker” that is always dormant

Directory & File Structure

- code/test/
  - Holds all the test cases and the environment to build new test cases.
  - Follow the format in Makefile to make new tests

```
PROGRAMS = . . .  <test>

<test>.o: <test>.c
        $(CC) $(CFLAGS) -c <test>.c
<test>: <test>.o start.o
        $(LD) $(LDFLAGS) start.o <test>.o
<test>.coff:
        $(COFF2NOFF) <test>.coff <test>
```

Directory & File Structure

- code/threads/
  - Holds the threading and related routines for Nachos.
  - No changes needed in here unless you really want to change the threading internals, or are modifying the scheduler
  - Good idea to familiarize yourself with the threading and synchronization in Nachos

Directory & File Structure

- code/userprog/
  - Holds the beginnings of user program support and system calls handling
  - This is the only non functional portion of the Nachos directory structure.

Threads & Synchronization

- Nachos contains a complete threading library
- Nachos contains a complete synchronization library.
- The scheduler is already in place and uses simple FCFS scheduling.
- Use of provided synchronization primitives will implicitly control threads and scheduling.
- No need to explicitly control thread execution and scheduling

Threads & Synchronization

- Try to solve problems using the thread abilities before writing a new solution
- Use the synchronization classes as much as possible.
- Example (Join):

```
Process A (in Join()):
    Semaphore* s = new Semaphore("AJoinsB");
    S->P(); // A blocks on
Process B (in Exit()):
    Semaphore* s = GetSemaphore("AJoinsB");
    S->V(); // wake up A
```

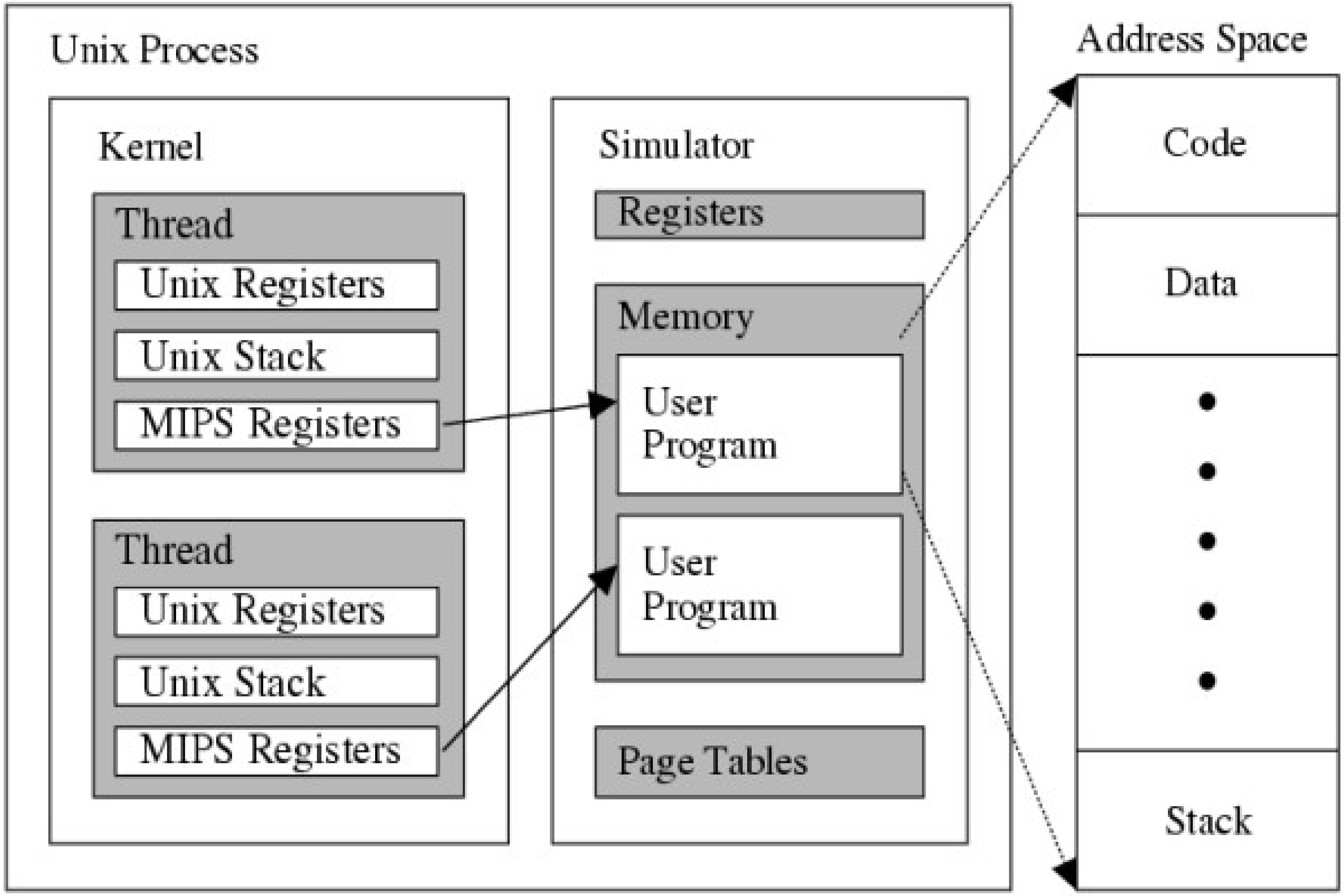
Threads & Synchronization

- Thread Miscellany
  - Threads do not interrupt at any point in time (not preemptive).
  - Thread switching happens at various places as a result of calling certain functions
  - A `while(1);` will stop Nachos
  - Stack space for threads is limited so don't define local variables like `char bigString[20000];`
  - Concurrency & Synchronization issues are a BIG deal.

Unix vs. Kernel vs. User Programs

- The kernel runs inside of a Unix process
- The simulator runs alongside the kernel inside the Unix process
- The user program run inside the simulator
- The are many things called the same thing that are different depending on where they are (i.e. stacks, registers, threads, processes)
- It is easy to get mixed up about these things

Unix vs. Kernel vs. User Programs



MIPS Simulator & Nachos

- The simulator is in control of Nachos from the beginning (from Machine::Run)
- Kernel code only gets executed as a result of a few specific events
- Interrupts cause the simulator to call the appropriate interrupt handler
- Exceptions cause the simulator to call the exception handler
- System Calls cause the simulator to call the exception handler

MIPS Simulator & Nachos

- Interrupts
  - Interrupts are generated by the simulated hardware in response to particular external events
  - These include the disk I/O, console I/O and timers
  - The interrupt mechanism is completely automatic and you have no control over it
  - For instance when a timer interrupt happens the kernel will yield the current thread and then the scheduler will automatically schedule the next thread to run (see timer.cc and alarm.cc)

MIPS Simulator & Nachos

- Exceptions and System Calls
  - Exceptions are things like divide by zero and page faults which need to be handled
  - System Calls are requests from user programs for the kernel to perform a desired action
  - The entry point is ExceptionHandler() in exception.cc
  - Once ExceptionHandler returns the simulator is in control again

MIPS Simulator & Nachos

- Running the Simulator
  - The simulator is started by calling Machine::Run
  - This should be done ***only once per process***
  - The simulator is self contained and only uses the registers, memory and page tables (or TLB)
  - During a context switch the register swapping is handled by the thread
  - During a context switch the page table (or TLB) information needs to be updated (the beginnings are in addrspace.cc)

Address Spaces & Executables

- Address Spaces
  - The current address space implementation is very basic
  - Need to extend this to support nonlinear frame mapping and allocating memory
  - Need to add support for translating and reading to/from an address space
  - Take care when modifying the address space to include all the sections of the NOFF file and the stack

Address Spaces & Executables

- Executables
  - Nachos uses an executable format called NOFF
  - NOFF files consist of a few sections:
    - .code
      - The program instructions that the simulator will execute
    - .initdata
      - The initialized data that holds predefined variable values
    - .uninitdata
      - The uninitialized data. The is the only section where the values are not read from the file. Initialize to zero.
    - .rdata
      - The read-only data in the executable. This is comprised mainly of literal strings (i.e. `char* temp = "Kevin";`)S

Address Spaces & Executables

- Creating Address Spaces
  - When creating address spaces, deal with every section of the NOFF file explicitly
  - Don’t forget the required stack space
  - Make sure to mark the pages that are to be read-only as such
  - Deal with pages that contain more than one section (i.e. pages with half code and half data)
  - Create the page table for the process and efficiently allocate the required memory for that process

Common Problems

- New & Delete
  - New & Delete can cause crashes because of invalid memory accesses that occurred at other locations
  - Hard to track down source
  - Example (fictitious):

```
// in one function
char* temp = new char[10];
temp[11] = ‘a’; // incorrect, but works

. . .

// in another function further down
char* temp2 = new char[10]; // causes segfault
```

Common Problems

- Creating a new thread
  - Once a thread is created it is automatically scheduled
  - The new thread can start running at any time
  - Cannot pass a member function to the Thread::Fork routine.
  - Incorrect Solution:

```
Thread* t = new Thread;
Process* p = new Process;
t->Fork(Process::Start, p); // compiler error
```

Common Problems

- Creating a new thread (cont.)
  - Correct solution:

```
void ProcessStart(void* arg)
 {
 Process* p = (Process*) arg;
 p->Start();
 }
. . .
Thread* t = new Thread;
Process* p = new Process;
t->Fork(ProcessStart, (void*)p);
```

Common Problems

- Segmentation Faults (and Bus Errors)

```
fred@mud: ~ > ./test
Segmentation Fault (core dumped)
fred@mud: ~ > gdb test
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0xef6a4734 in strlen () from /usr/lib/libc.so.1
(gdb) bt
#0  0xef6a4734 in strlen () from /usr/lib/libc.so.1
#1  0xef6da65c in _doprnt () from /usr/lib/libc.so.1
#2  0xef6e37b8 in printf () from /usr/lib/libc.so.1
#3  0x1095c in func1 () at test.c:6
#4  0x10970 in func2 () at test.c:11
#5  0x10984 in main () at test.c:16
(gdb)
```

Common Problems

- Translation
  - The translation routines in the machine class are a good start but not general purpose.
  - These routine are designed for single byte/integer
  - In designing translation routines consider larger translations for reading and writing
  - In particular consider cross page conditions and dealing with null terminated strings
  - Also watch out for the endianness change between MIPS & Kernel

Conclusion

This was only a brief introduction to Nachos

Understanding Nachos internals will help a great deal in debugging and designing

http://imag-moodle.e.ujf-grenoble.fr/course/view.php?id=126

Get started on *Etape 1* ASAP

A good work on *Etape 1* will reduce the workload needed for future *Etapes*