• Alexandru (Sasha) Andrei
• alean@ida.liu.se
• Embedded Systems Laboratory
Introduction

- Course: Concurrent Programming and Operation Systems  
  Homepage: [http://www.ida.liu.se/~TDDB63/](http://www.ida.liu.se/~TDDB63/)

  check regularly
  read messages section
Laboratory Sessions

- SU rooms
- Solaris
- C/C++
- Tools: emacs (or another editor), gmake, ddd, man

=> No support without using debugger before!

- Mark your changes in the source code
• Marking changes

// 04-01-24: your_name: changed

... end of changes
Laboratory Sessions

• File header

// =========================================
// TDDB63: Lab 1: solution.cpp
// ----------------------------------------
// Group XYZ: Mister X / Miss Y
// ----------------------------------------
// This module extends Nachos to be able to work
// with semaphores.
// ----------------------------------------
// 00-02-01: misterX: created
// 99-02-12: missY: semSignal: changed signal type
// ----------------------------------------
Laboratory Sessions

• Function header

```c
// ---------------------------------------------------------------------------
// int readInput(int file)
// ---------------------------------------------------------------------------
// This function reads continuously from the file and returns the number of bytes read
// ---------------------------------------------------------------------------
// Parameters:
//   file: handle of the file to read from
// ---------------------------------------------------------------------------
```
Tips

• Read documentation and source code
  – Nachos Beginner’s Guide (html)
  – A Roadmap Through Nachos (pdf)
• Use a debugger (gdb / ddd)
• Use code browsing tools (emacs, grep)
• Use different test cases
• Check the homepage regularly
Nachos Overview

- “Not Another Completely Heuristic Operating System”
- Free operating system and simulation environment to study an operating system
- Runs as a single Unix process (real operating systems run on bare machines)
- Simulates low-level facilities (including interrupts, virtual memory and interrupt-driven device I/O)
- Implemented in subset of C++, ca. 2.500 lines of code

- See homepage material and links
- Browse the source code !!!!!!
Overview

• Nachos *simulates* (emulates ?) a machine approximating a *real machines architecture*

• Has:
  – Registers, Memory, CPU, interrupts
  – Event-driven simulated clock

• Can execute arbitrary programs
Nachos

Given

Lab. 1,2,3

- code/machine
- code/threads
- code/userprog
- code/test
- Test programs
Machine Object

- Available through variable **machine**
- Public attributes:
  - registers
  - mainMemory
  - virtualMemory
    (single linear page and software-managed TLB)

- Methods: **Machine** (constructor), **Translate** (translates virtual addresses), **OneInstruction**, **Run**, **ReadRegister**, **WriteRegister**, **ReadMem**, **WriteMem**

=> Read documentation and source code
Interrupt Management

• Nachos simulates interrupts with an event queue and a clock (clock ticks -> queue is examined for pending events)

• The clock ticks, when ...
  – Interrupts are restored
    (used often for mutual exclusion purposes)
  – One instruction is executed
  – “Ready list” is empty

• Object Interrupt:
  – Main methods: Schedule (schedule event), SetLevel (disable / enable), OneTick
    => Read documentation / source code
Real-Time Clock Interrupts

• Object Timer:
  – Simulates a real-time clock (generates interrupts at regular intervals)
  – Methods: **Timer** (creates a real-time clock, can be randomly influenced)
  – ./nachos –rs #number (nachos –rs 100)
Address Translation

- Nachos supports two virtual memory architectures:
  - **Linear page tables**  
    (easier to program)
  - **Software-managed TLB**  
    (closer to what current machines support)
  - Only one at a time

- Linear page tables:
  - Virtual address: page number and page offset
  - Physical address: machine->mainMemory + n * PageSize + m

- Read documentation / source code
Terminal Console Device

- Accessed through low-level primitives (initiated by I/O operation, performed by an “operation complete” interrupt)

- Object Console
  - Simulates behavior of a character-oriented CRT device (one character at a time)
  - Methods: Console (constructor, connects to files / streams), PutChar, GetChar, CheckCharAvail
Disk Device

- Accessed through low-level primitives (initiated by I/O operation, performed by an “operation complete” interrupt)

- Object Disk
  - Simulates behavior of a real disk (single platter, multiple tracks containing sectors, blocks uniquely identified by their sector number)
  - Methods: **Disk** (constructor), **ReadRequest** (read single sector), **WriteRequest** (write single sector), **ComputeLatency** (used to decide when to schedule an “I/O complete” interrupt when servicing a read or write request)
Nachos Processes and Threads Overview

- Process: a user program in execution
- Thread (Nachos): the kernel correspondent of a user process
Lab1: Threads

- Nachos threads are in one of four states
  - READY
  - RUNNING
  - BLOCKED
  - JUST_CREATED (temporary state)
- The scheduler decides which thread to run next
- Synchronization facilities are provided through semaphores
Threads and Scheduling

- **Scheduler** decides which thread to run next (invoked whenever the current thread gives up the CPU or an interrupt arrives)
- **Ready list** (threads which are ready to run)
- Nachos: unprioritized, round-robin fashion
- Object Scheduler methods:
  - `ReadyToRun(thread_object)` - place the thread in ready list
  - `FindNextToRun`
  - `Run`
Thread switching

- **Suspend** current thread
- **Save** thread **state** (registers)
- **Restore state** of thread to switch to
- => function `Switch(oldThread, nextThread)` (written in assembly language)
Thread Object

- Thread Object methods:
  - **Thread** (constructor)
  - **Fork** (making thread executable)
  - **StackAllocate**
  - **Yield** (suspend and select next waiting one)
  - **Sleep** (suspend)
  - **Finish** (terminate)

- Read documentation / source code
Nachos Threads

void ThreadFunction1(int which);
void ThreadFunction2(int which);
void ThreadTest() {
    Thread *t1 = new Thread("first thread");
    Thread *t2 = new Thread("second thread");
    t1->Fork(ThreadFunction1, 1);
    t2->Fork(ThreadFunction2, 2);
    ThreadFunction(1);
}

Synchronization and Mutual Exclusion

- Mutual exclusion achieved by disabling / enabling interrupts
- Synchronization provided through semaphores
- Object Semaphore methods:
  - **Semaphore** (constructor)
  - **P** (decrement semaphore’s counter, blocking caller if 0)
  - **V** (increment semaphore’s counter, releasing one thread if any are blocked)
Lab1: Semaphores

- A semaphore is an integer variable, accessible only by the operations P and V (Claim, and Release)

Wait(S): \textbf{while} S <= 0 \textbf{do} no-op; \\
S := S -1 ;

Signal(S): S := S + 1 ;
Locks

- A binary semaphore
- Checks that only the thread that acquired the lock can release it
- Use the Semaphore class + additional stuff
Condition Variables

• Semaphores and Locks are not very efficient when dealing with multiple resources

P1
Lock1->Acquire();
Lock2->Acquire();
...
Lock2->Release;
Lock1->Release();

P2
Lock3->Acquire();
Lock1->Acquire();
...
Lock1->Release;
Lock3->Release();

P3
Lock2->Acquire();
Lock3->Acquire();
...
Lock3->Release;
Lock2->Release();

Why hold a lock if you can not use it ?
Condition Variables (Wait)

- Wait
  - Release the lock
  - Add the thread to a queue of threads waiting on this condition
  - Go to sleep
  - Acquire the lock
Condition Variables (Signal)

• Signal
  – Take one thread from the list
  – Run it (scheduler->ReadyToRun(that thread))

• Broadcast
  – Wake up all the threads and let them compete for the lock
Readers & Writers

- Given a buffer
- Two kinds of processes can access that buffer:
  - Readers
  - Writers
- Correct operation:
  - several readers can read simultaneously
  - Only one writer is allowed to access the file
Readers & Writers

- int counter_for_read;
- Lock *lock_for_counter;
- Lock *lock_for_readers_writers;
- Condition *c_counter;
- Condition *c_write;
Lab 1: Preparation

- Read and understand the partial thread system (nachos-3.4/code/threads)
- Remember: **module add prog/gcc/2.8.1**
- Run the test program in the debugger and follow the state of each thread object

- Compile in the **code/threads** directory
Lab1: Assignment 1

- Implement locks and conditions - when used together provides the same functionality as monitors
- Write code for the Lock and Condition classes (in the file ..:/code/threads/sync.cc)
  - Interfaces (class definition) for Lock and Condition are already defined (in ..:/code/threads/sync.h)
  - No changes to the public interfaces should be done (you can add private variables)
Lab 1: Assignment 2

• Test your implementation
  – Write a class **BoundedBuffer** (in the file code/threads/threadtest.cc) and use it for producer/consumer communication. (Silberschatz, p. 172)
  – Don’t forget border conditions: empty buffer, full buffer
  – Try several producers and/or consumers
class BoundedBuffer {
public:
    BoundedBuffer(int bufsize);
    ~BoundedBuffer();
    int Read();
    void Write(int value);
private:
    // some variables you need
};