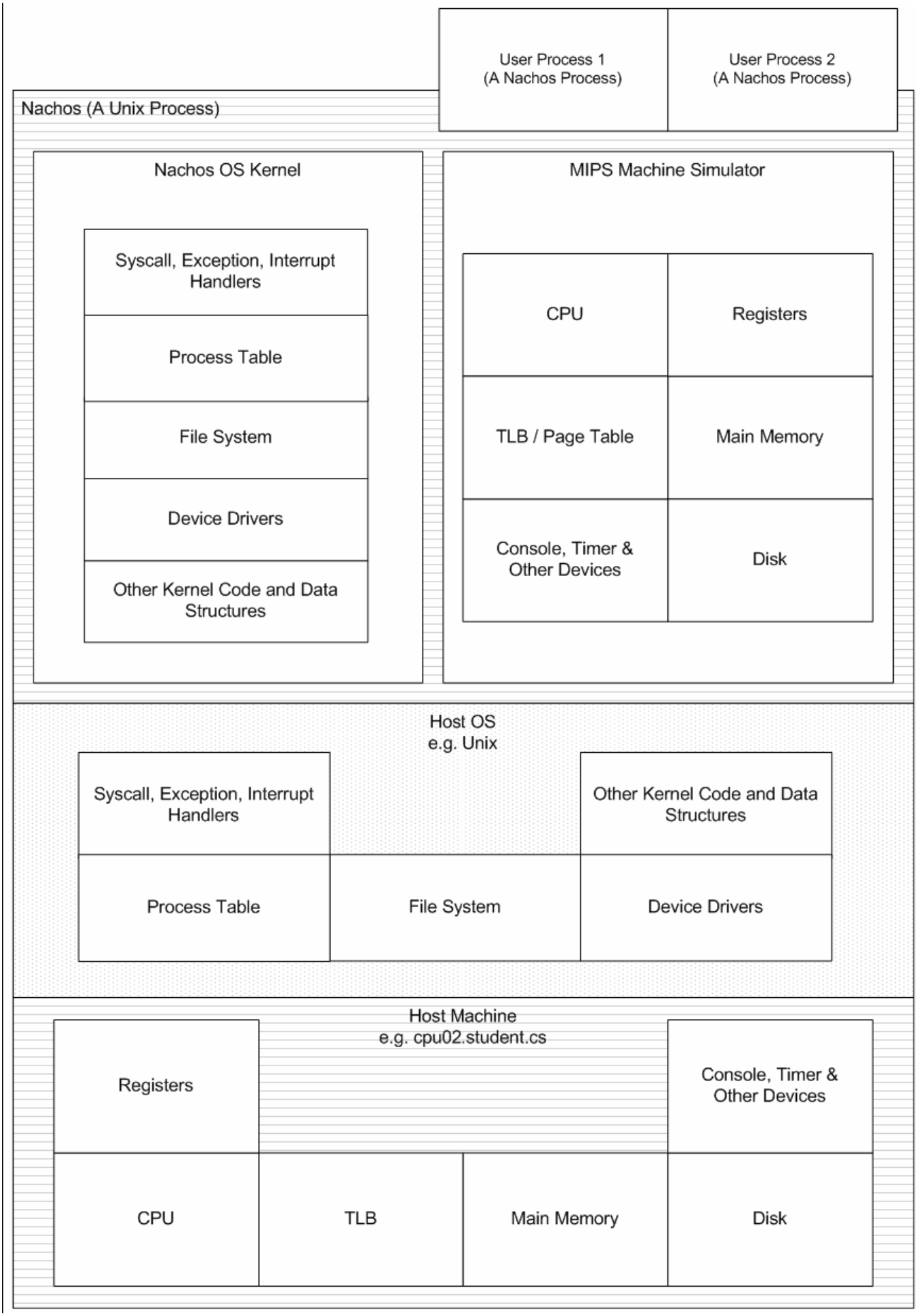
Nachos Tutorial

For CS350

By Richard Jang

Nachos Architecture

- Just a process than runs on top of another OS and host machine
- Process contains the kernel (OS) and mips code machine simulator (virtual machine like Java except mips code instead of bytecode)


User Process 1
(A Nachos Process)
User Process 2
(A Nachos Process)
Nachos (A Unix Process)
Nachos OS Kernel
Syscall, Exception, Interrupt Handlers
Process Table
File System
Device Drivers
Other Kernel Code and Data Structures
MIPS Machine Simulator
CPU
Registers
TLB / Page Table
Main Memory
Console, Timer & Other Devices
Disk
Host OS
e.g. Unix
Syscall, Exception, Interrupt Handlers
Other Kernel Code and Data Structures
Process Table
File System
Device Drivers
Host Machine
e.g. cpu02.student.cs
Registers
Console, Timer & Other Devices
CPU
TLB
Main Memory
Disk

Nachos vs Real OS

- Nachos: OS runs beside machine i.e. on host machine; User processes run on machine simulator
- Real OS: OS and User processes run on same machine
- Nachos: H/W simulated
  - Interact with H/W by calling functions that eventually call underlying host OS library routines
- Real OS: H/W is real
  - Interact with H/W at a lower level
  - Read/write device registers or issue special I/O instructions

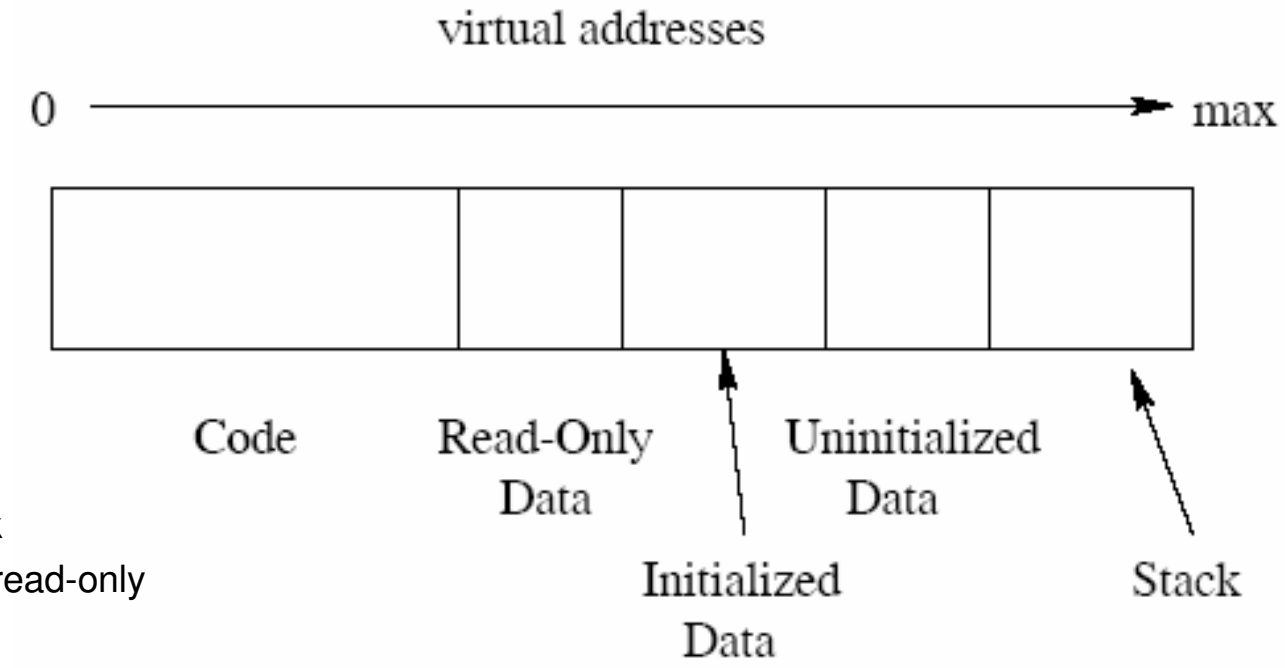
Nachos vs Real OS (2)

- Nachos: Time is simulated (incremented at discrete points in real time)
  - Interrupts can't happen immediately and interrupt anywhere in kernel code where interrupts are enabled
  - Interrupts happen only in places where simulated time gets advanced. When simulated time advances, Nachos will check for interrupts
  - Simulated time advances when interrupts re-enabled, between user instructions, when nothing in the ready queue
  - Incorrectly synchronized code may work in Nachos (e.g. pre-emption may not be possible in certain critical sections of kernel code unless time is advanced), but not in a real OS
- Real OS: Time is continuous
  - Kernel can be pre-empted anywhere where interrupts are enabled

User Process

- Executable in noff format (coff2noff converts from coff to noff format)
- Segments are page-aligned (size is rounded to nearest multiple of page size)

```
int a = 1; // initialized
int b; // uninitialized
main() {
  const int c = 2; // read-only
  int d = 3; // stack
  char *e;  // stack
  char *f = “hello”; // f on stack
                     // “hello” in read-only
  …
}
```



User Process (2)

- Kernel maintains data structures for each process
  - ProcTableEntry indexed by process id
    - contains the exit value of any child processes of the given process
  - Addrspace object: page table, open file table
  - Thread object for saving stuff (see below)
- Thread has 2 sets of registers and 2 stacks (one for each machine)
  - one to keep track of execution of user process on machine simulator (user-level).
  - one to keep track of where in machine simulator/kernel code we are currently at in the simulation of the given user process (kernel-level). Recall simulator/kernel code is executed on host machine and each process may be executing different kernel code, so we need to remember this for each process (or thread in we have multithreading)
- Each process in base version of Nachos has only 1 thread, so each process has only 1 thread object.
- For multithreading, each thread needs a user stack and a thread object, which contains both sets of registers and a kernel stack

Nachos Startup

- Initialize simulated H/W (e.g. timer, console, disk) and initialize kernel data structures (e.g. proc table, scheduler, file system)
- Load initial process from disk, create its addrspace and thread object
- Setup machine simulator to run process by initializing registers and setting the machine's page table to point to process's page table
- Run the process: machine->Run()
  - Fetch-decode-execute cycle. After executing each user instruction, it advances simulated time and checks for interrupts. When executing user instructions, the machine may detect and throw exceptions.

Nachos Startup (2)

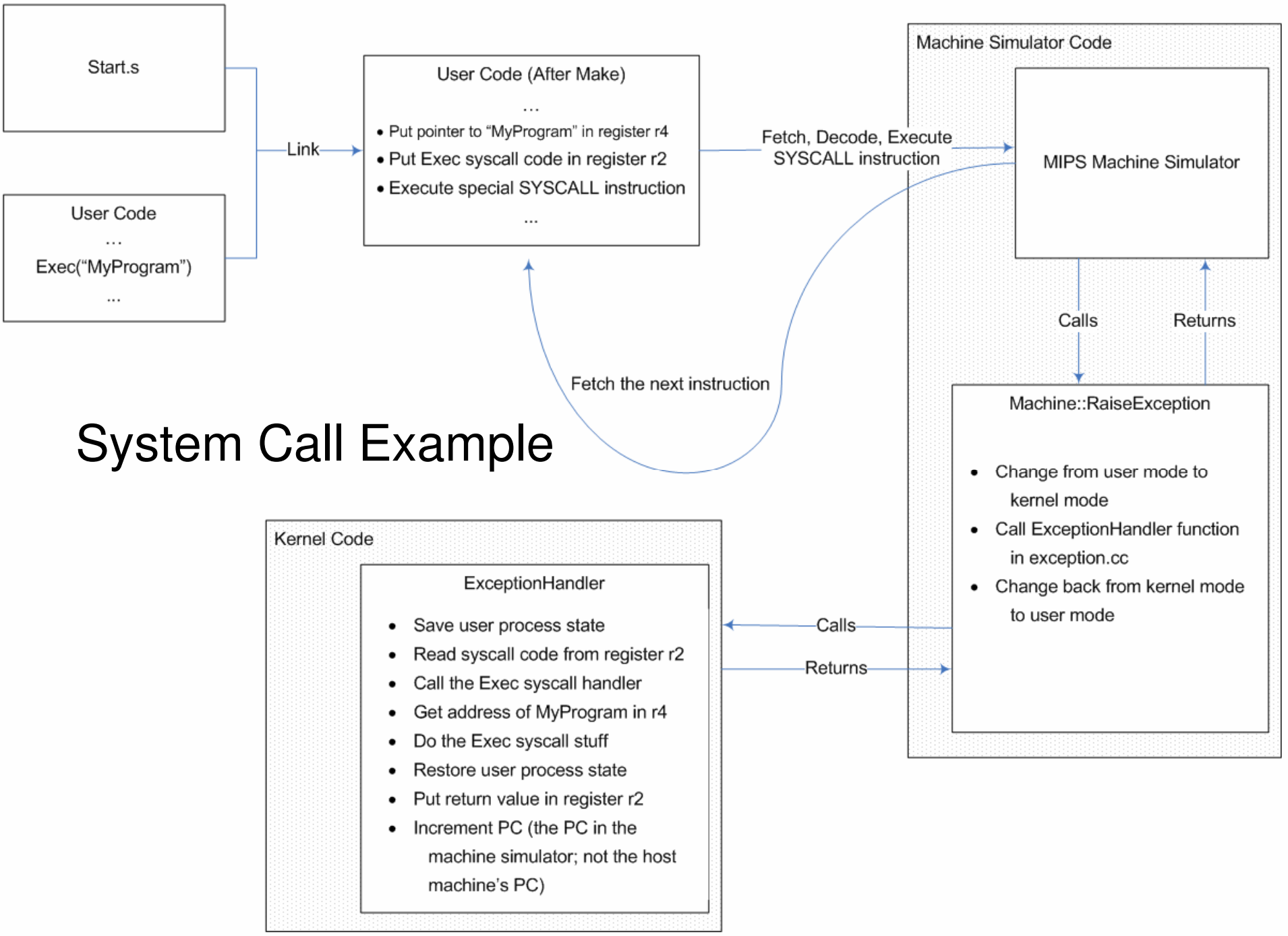
- For initial process, there does not appear to be code that allocates the kernel-level stack. This is because the kernel-level stack already allocated when we run Nachos (initial process uses the stack given to Nachos by Unix).
- For additional processes (via Exec syscall), need to explicitly allocate both stacks.
- Also note that Exec calls Thread::Fork, which sets up the new thread so that the first time it gets the CPU, it calls ProcessStartup.
  - ProcessStartup setups the machine simulator to run the process just like what happens for the initial process
- Thread::Fork is not called for initial process since initial process doesn't have to wait to get the CPU. It will startup right away and the same stuff as ProcessStartup will be executed

More on User Processes

- User program's "main" function is not executed immediately.
- During compilation of user program, start.s is linked in and the _start function gets set to program's entry point rather than main
- _start calls the user program's main function, then it calls the Exit(0) system call, which cleans up the process.
- Note that if you don't include Exit system in your user program, the Exit(0) will still be executed because it got added
- Exit does not delete the Thread object immediately because the current kernel-level execution stack is still being used since we're still in Exit (executing kernel code); i.e. can't delete memory that is still being used
- Instead, the thread is marked for deletion and a context switch happens. After switching threads, the marked thread can be deleted

More on User Processes (2)

- For processes spawned by Exec system call, ProcessStartup (which sets up the machine to run _start) does not get executed immediately.
- First ThreadRoot gets called, which calls ThreadBegin, which basically enables interrupts, then ProcessStartup gets called, then ThreadEnd gets called, which marks the thread for deletion *.
  - For exec'd processes that contain only 1 thread, the thread may appear to get marked for deletion twice - once by Exit(0) and once by ThreadEnd. If you look at the code carefully, you will notice that we never reach the code for ThreadEnd after executing Exit(0)
- For the initial process, ThreadRoot does not get called. However, interrupts are enabled before running the initial process and Exit(0) will mark it for deletion, so the same things happen for the initial process and processes that are exec'd even though it doesn't look like it



System Call Example

System Call Example (2)

- In actuality, in ExceptionHandler, there is no save & restore user process state since OS and user process run on different machines, so the registers for the user process are not affected when running ExceptionHandler
- Real OS would have save & restore process state since OS and process run on same machine
- PC incremented after system call, but not after an exception
- Handling of interrupts (e.g. quantum expires, disk I/O completes) is similar where instead of exception handler there is an interrupt handler

Context switching

```
void Scheduler::Run (Thread *nextThread, bool finishing)  {
    …
    oldThread->SaveUserState();       // save the user CPU registers
    oldThread->space->SaveState();  // currently does nothing
    …
    kernel->currentThread = nextThread;  // keep track of new current thread
    …
    SWITCH(oldThread, nextThread);
    // (kernel-level only)
    // save stack pointer, registers, frame pointer, return address
    // then load new stack pointer, registers, return address
    // then jump to new return address (return address is where in kernel code we will return to;
    // in this case it's immediately after SWITCH)
   …
   oldThread->RestoreUserState();       // to restore user CPU registers
                                        // PC set to where in user code we will return to
   oldThread->space->RestoreState();  // set machine page table to
                                        // this process's page table
}
```

Concurrency

- Scheduler methods disable interrupts to ensure mutual exclusion of ready queue
- You can use synchronization primitives, such as Locks, Semaphores for mutual exclusion, synchronization (see synch.cc)
- Drawback with disabling interrupts is context switching cannot happen even if other threads won't access the thing being protected.
  - Decreases level of multi-programming
- With scheduler, can't use locks or semaphores cause if lock busy, end up calling scheduler again, but lock is busy – get infinite loop
- You will need to test your implementation using multiple processes

Nachos File Systems

- Nachos has 2 file systems – Stub & Real
- Stub uses Unix file system. You will use this in A1
- Real uses a Unix file. You will build a file system on top of this file in A3.

Learning Nachos

- Will likely take you the entire term
- You need a good knowledge of C/C++
  www.student.cs.uwaterloo.ca/~cssystems/
- Don't need to know all the details of machine simulator
- You will not have to write assembly code (except maybe a little for multithreading)
- Use a code browser to navigate around the code
  e.g. www.stack.nl/~dimitri/doxygen/

Learning Nachos (2)

- Other Nachos Tutorials
- For an Introduction
  www.cs.rit.edu/~mpv/course/os2/NachosPrimer.pdf
- Road Map to Nachos (longer)
  www.cs.duke.edu/~narten/110/nachos/main.ps
- Course Notes from some University
  www.dei.isep.ipp.pt/~alex/publico/soii/nachos_study_book.pdf
- The tutorials may not refer to the same version of Nachos that we are using

Implementation Tips

- Design before coding. Come up with preliminary design and some test programs to get an idea of what you will implement.
- Linux, Windows not supported, you still need to make sure it works on Unix
- Use an IDE; e.g. Eclipse with CDT plugin www.eclipse.org
- Use CVS if working in groups (info on course web)
- Make sure group has permission to access repository
- Ensure group id of repository set to your group id. Use something like chgrp -R cs350_### .cvsroot/nachos
- Can import files in CVS on student.cs into Eclipse at home

Implementation Tips (2)

- Use student.cs.uwaterloo.ca for host and /u/<your id>/.cvsroot as the repository path (where your CVSROOT is located). URL looks like :extssh:user_id@student.cs.uwaterloo.ca:/u2/user_id/.cvsroot
- Accessing files on your account from Windows www.cs.uwaterloo.ca/~ctucker/cscf_samba/cscf_samba.html
- Should not touch anything in machine directory except possibly changing size of main memory in machine.h
- Nachos already comes with list, sortedlist, hashtable, and bitmap. See the lib directory

Testing Tips

- Test programs are in C. Nachos is in C++
- Test each requirement mentioned in assignment specs
- Test boundary cases (e.g. invalid arguments)
- Stress testing - Test system limits (e.g. max number of threads)
- Test using multiple processes
- If it's not tested, even if it's implemented, you will not get credit
- Can use debug flags to print things in kernel code. e.g.
  DEBUG(dbgSysCall, "System Call: Exit status=" << status);
- Can make your own debug flags. See debug.h
- Can use Join system call to block parent process until child process finishes
- io_lib.h in test directory contains functions for user programs to print strings and integers

Testing Tips (2)

- Don't put a big array on stack; otherwise it fill overflow. Instead make the array a global variable so it will be in the uninitialized data segment
- Console output is funny. When printing a string, the console will output 1 char at time then block until time advances a bit and then output the next character.
  - When process is blocked, other processes can print
  - Output from multiple process will be scrambled
- Calling Exit only terminates the process and not the machine. Nachos will appear to freeze because it keeps on advancing simulated time.
- To shut down Nachos, call Halt
- GDB debugger on Unix www.student.cs.uwaterloo.ca/~cs350/common/debug.html
- Debugging memory problems - Valgrind (Linux): www.valgrind.org

Testing Tips (3)

- Be careful of naming conflicts. If you include list.h, which contains the List class, in filesys.h, which in turn contains the List method, there will be a naming conflict when you try to use the name List
- In a given scope, the names of member variables, member functions, and included classes should have distinct names otherwise problems may arise (exception: function overloading, using namespaces).

Circular Includes

- Things can't be defined twice; e.g. a.h includes b.h and b.h includes a.h
- This leads to infinite recursion.
- One Solution: Use conditional compilation

[a.h]	[b.h]
#ifndef A_H #define A_H #include “b.h” … #endif	#ifndef B_H #define B_H #include “a.h” … #endif

Circular Includes (2)

- Compilers need to resolve names and determine know how much space to allocate.
- If a.h contains only a pointer to class B then can simply use forward declaration since all pointers take up the same amount of space
- Can get faster compilation if use forward declarations

[a.h]	[a.cc]

#ifndef A_H
#define A_H
class B;
// don't need to include b.h
class A {
  B* b;
}
#endif
	#include "a.h"
	#include "b.h"
	…
	b = new B();
	b->func();
	….

- For template classes, e.g. use: template <class T> class SynchList;

Design Doc

- Read General Assignment Requirements doc under assignments on course web
- Should address each requirement in assignment specs e.g. describe the data structures and how they are used
- Another student should be able to implement your design by reading your design doc
- Do not repeat assignment specs
- Do not describe stuff that TAs already know; e.g. how a system call works (which I have already presented)
- If not mentioned in the design doc, it could be a design decision, which you need to justify. When in doubt, ask on newsgroup.

Testing Doc

- Should describe how you test each requirement in assignment specs
- Should mention which requirement(s) each test program is testing
- Should describe how to run your test program
- Should describe the expected output/behavior
- Good idea to use a table format
- Should mention all bugs and what has not been implemented (you'll lose less marks)

Autotesting

- If you came up with good test cases then you should do well
- We will never reveal the autotesting code

Submitting

- Use submit command
- Only 1 person in group should submit
- Keep a copy of your submission
- When submitting you may get

  FYI: code NOT found as a file in /u/cs350/handin/1/username

  FYI: code/* NOT found as a file in /u/cs350/handin/1/username

  FYI: code/*/*/*/* NOT found as a file in /u/cs350/handin/1/username

  …
- Don’t worry about this. Submit script is looking for nested subdirs under code dir
- Can always verify which files were submitted using: submit –list
- Total 6 slip days, max 3 per assignment except last, where max 2
- You can resubmit as many times as you like within the slip days period
- You don’t need to tell us how many slip days you will use. We will know from the timestamps

Other

- Don’t copy code from previous terms and from other groups. We will run Moss to compare your code to code from previous terms and code from this term
- If nachos crashes because of a bug, you should free server resources using “kill -9 [pid]”. Can use “top” command to find the pid.
- List of CPU servers: www.cs.uwaterloo.ca/cscf/student/hosts.shtml

C/C++

- Pointer = a number = an address in memory
- Use * to refer to the contents at that memory location. Known as dereferencing a pointer. For arrays, can use [element_index] in addition to *
- Use & to refer to the variable's address
- Use "new _type_" to dynamically allocate space for one element of type _type_, and "new _type_[num_elements]" for an array. Initial value(s) for the allocated element(s) are not automatically provided unlike Java
- Use "delete" to deallocate space allocated with "new _type_", and "delete []" to deallocate space allocated with "new _type_[num_elements]"
- Good idea to set pointer to 0 (NULL) after call delete so that pointer ends up pointing to nothing.
- If no value to initialize pointer, use NULL
- void* pointers point to values of no type (need to typecast before can dereference)

C/C++ (2)

- Array examples

  char *a = new char[12];

  a = &a[0] = address of first char

  a[0] = *a = first char

  a[1] = *(a+1) = second char

  char *b = a;

  b = &b[0] = address of first char of a

  &b = address of variable b (&b != b)

  delete [] a;

  a = 0;

C/C++ (3)

- Multidimensional arrays

```
int a[2][3];
// order of increasing address ->
a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2]
int* p = a + 4*sizeof(int); // points to a[1][1]
```

- Strings in C are null-terminated by '\0' = ascii code 0

```
char *s = "abc";
// s[0] = 'a', s[1] = 'b', s[2] = 'c', s[3] = '\0' = 0;
// strlen(s) is equal to 3;
// treat s as read-only, behavior of s[2] = 'z' is undefined
```

- Templates

```
List<Thread *> *readyList = new List<Thread*>();
```

Bad Code

- Returning local variable pointers since space for local variables are deallocated upon return

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

- Declaring a pointer, but not allocating space

```
void f() {
  char *s;
  s[2] = ‘a’;
}
```

Endianess

- Byte-ordering within a word (an int)
- Big-Endian (Sparc) (most significant byte first)
- Little-Endian (x86) (least significant byte first)
- e.g. 66051
- We write/read 00000000 00000001 00000010 00000011
- Little-Endian writes words in opposite direction we write and reads it in opposite direction that we read.

Address	Big-End	Little-End
0	00000000	00000011
1	00000001	00000010
2	00000010	00000001
3	00000011	00000000

Questions?