Supporting User Programs

Phase 2 of Nachos Project

Xiangru Chen

ACM Honored Class 06
Shanghai Jiao Tong University

October 28, 2010

Outline

1. Knowledge
2. Tasks
3. Selected Topics

Outline

1. Knowledge
2. Tasks
3. Selected Topics

Important Classes

`nachos.machine.Processor`

- simulator of a MIPS CPU
- supports a subset of R3000 instruction set
- includes the “main memory”

Important Classes

`nachos.machine.Processor`

- simulator of a MIPS CPU
- supports a subset of R3000 instruction set
- includes the “main memory”

`nachos.machine.FileSystem` and `nachos.machine.OpenFile`

- basic interfaces for file system
- *Abstract Factory* pattern

Important Classes

`nachos.userprog.UserKernel`

- `extends ThreadedKernel`
- a kernel that support multiple user processes
- contains global algorithm and data for the OS

Important Classes

`nachos.userprog.UserKernel`

- extends `ThreadedKernel`
- a kernel that support multiple user processes
- contains global algorithm and data for the OS

`nachos.userprog.UThread`

- extends `KThread`
- can execute user code inside a user process

Important Classes

`nachos.userprog.UserKernel`

- `extends ThreadedKernel`
- a kernel that support multiple user processes
- contains global algorithm and data for the OS

`nachos.userprog.UThread`

- `extends KThread`
- can execute user code inside a user process

`nachos.userprog.UserProcess`

- contains local algorithm and data for a process
  - page tables, file tables, etc.
- much work need to do here

Code Overview

Booting

- a kernel thread does the initialization
- then calls `UserProcess.execute(shellProg, args)` to start the shell program

Code Overview

Booting

- a kernel thread does the initialization
- then calls `UserProcess.execute(shellProg, args)` to start the shell program

Launching a Process

- load binary code from file
- run instructions on `Processor`

Code Overview

Address Translation

- use page table in phase 2
- page table is an array of `nachos.machine.TranslationEntry`
- call `Processor.setPageTable()` before process runs

Code Overview

Address Translation

- use page table in phase 2
- page table is an array of `nachos.machine.TranslationEntry`
- call `Processor.setPageTable()` before process runs

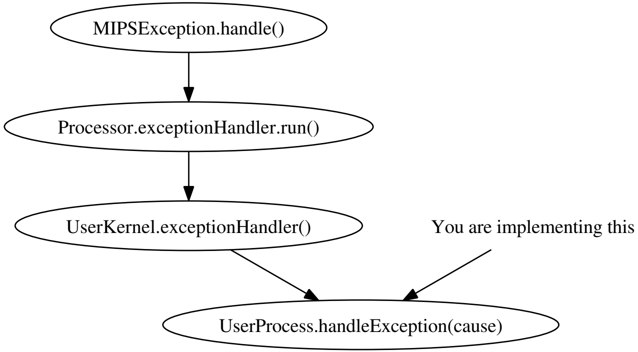
Context Switch

- `UThread.saveState()` is called before context switches
- `UThread.restoreState()` is called after context switches

Code Overview

When exception occurs

- exception handler in `UserKernel` is invoked



MIPSException.handle()

Processor.exceptionHandler.run()

UserKernel.exceptionHandler()

You are implementing this

UserProcess.handleException(cause)

Outline

1. Knowledge
2. Tasks
3. Selected Topics

Task 1

Implement syscalls for file management.

- `creat`, `open`, `read`, `write`, `close`, `unlink`
- see **syscall.h** for details

Make use of `nachos.machine.StubFileSystem` .

- it's a wrapper of your real file system
- through `StubFileSystem` , you can access files in the "test" directory by the `OpenFile` interface

Task 2

Implement simple paging using page table.
Modify `UserProcess.readVirtualMemory()` and `UserProcess.writeVirtualMemory()` .

- they are widely used & important method
- they are used to copy data between kernel and user's virtual address space
- better to make their code independent with address translation

Task 3

Implement syscalls for process management.

- `exec`, `join`, `exit`
- see **syscall.h** for details

Bullet-proof all the syscalls.

- i.e. there should be *nothing* a user program can do to crash the operating system

Task 3

Take `join` as an example:

`handleJoin(int pid, int addrStatus)`

1. check whether *pid* is a child of `currentProcess`
2. call `join` on the child's `KThread` object
3. check whether the child exited normally
4. get the return value of the child
5. write the return value to memory address *addrStatus*
6. do some cleanings
7. return a value

Task 4

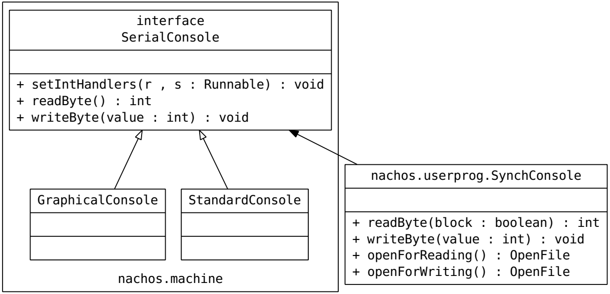
Implement a lottery scheduler.

- if you did make a good design before implementing `PriorityScheduler` , this will be very easy

Outline

1. Knowledge
2. Tasks
3. Selected Topics

Consoles



interface
SerialConsole

```
+ setIntHandlers(r , s : Runnable) : void
+ readByte() : int
+ writeByte(value : int) : void
```

GraphicalConsole

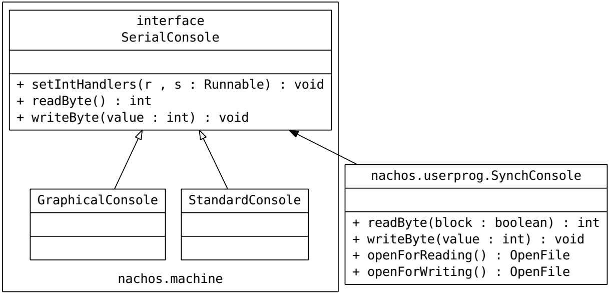
StandardConsole

nachos.machine

nachos.userprog.SynchConsole

```
+ readByte(block : boolean) : int
+ writeByte(value : int) : void
+ openForReading() : OpenFile
+ openForWriting() : OpenFile
```

Consoles



interface
SerialConsole

+ setIntHandlers(r , s : Runnable) : void
+ readByte() : int
+ writeByte(value : int) : void

GraphicalConsole

StandardConsole

nachos.machine

nachos.userprog.SynchConsole

+ readByte(block : boolean) : int
+ writeByte(value : int) : void
+ openForReading() : OpenFile
+ openForWriting() : OpenFile

- Make use of `openForReading()` and `openForWriting()` to provide standard input/output to user program.
- Consider whether your readings need to be blocked or not.

Consoles

A difference between nachos console and your real console:

- by default, your real console program will buffer your input until a line break
- for `*nix` users who want to simulate more accurately:

```
#!/bin/sh

onexit () {
  stty $OLDSTTYSTATE
}

OLDSTTYSTATE=`stty -g`
trap onexit 0
stty -icanon min 1 -echo
java -cp bin nachos.machine.Machine $*
```

Debugging

Use debug flags properly will help a lot in debugging.

- invoke nachos with `'-d <debug flags>'`
- see `nachos.machine.Lib.debug()`

Debugging

Use debug flags properly will help a lot in debugging.

- invoke nachos with `’-d <debug flags>’`
- see `nachos.machine.Lib.debug()`

Use different random seeds may help you find bugs.

- invoke nachos with `’-s <seed>’`
- the default seed is 0
- use various random seeds to test your system
- without human intervention, the same random seed will lead to the same result
- see `nachos.machine.Lib.random()`

Debugging

Use debug flags properly will help a lot in debugging.

- invoke nachos with `'-d <debug flags>'`
- see `nachos.machine.Lib.debug()`

Use different random seeds may help you find bugs.

- invoke nachos with `'-s <seed>'`
- the default seed is 0
- use various random seeds to test your system
- without human intervention, the same random seed will lead to the same result
- see `nachos.machine.Lib.random()`

Use `GraphicalConsole` to seperate standard input/output from debug output.

Cross Compiling

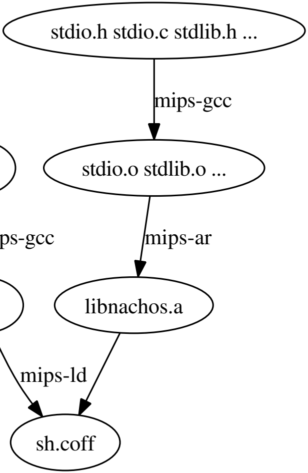
First, edit your `Makefile` to make it work properly

- tell it where is your cross compiler
    - i.e. `ARCHDIR= "../mips-x86.linux-xgcc"`

Cross Compiling

First, edit your `Makefile` to make it work properly

- tell it where is your cross compiler
  - i.e. `ARCHDIR= "../mips-x86.linux-xgcc"`



Integrating with Tiger Compiler

Need libraries to support tiger programs.

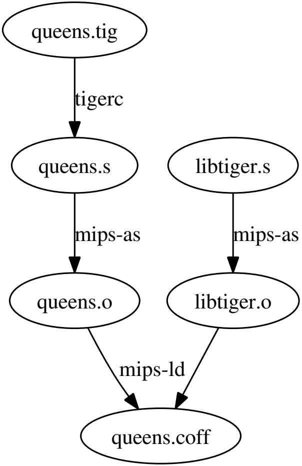
- remember the `runtime.s` in your tiger project?

Integrating with Tiger Compiler

Need libraries to support tiger programs.

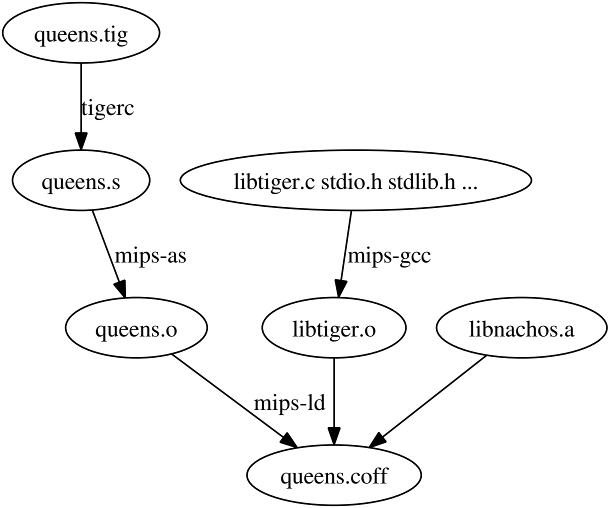
- remember the `runtime.s` in your tiger project?

Approach 1:



Integrating with Tiger Compiler

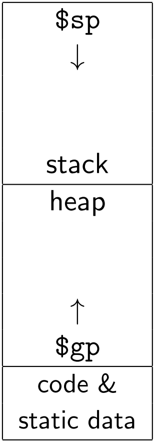
Approach 2:



Integrating with Tiger Compiler

Tiger programs need dynamic memory allocation.

- we can use $gp to allocate memory
- give $gp a initial value in `UserProcess.initRegisters()`
- write memory allocation programs in your library, for example:
  - `move $2, $28`
  - `addu $28, $4`
  - Think: will this code work?



$sp

↓

stack

heap

↑

$gp

code & static data

Q & A