HoH: CSL373/CSL633 Labs

What's the best way for learning OS? Create one!

Feb 19, 2015

Also available in pdf, beamer, slides.

Introduction

Introduction

Hello! I'm Alice. I am your TA for this course.

In this series, you will join forces with me, and together, we will build a *kernel from the scratch*. We both will be working on this kernel.

I'll do some coding in a branch, and ask you to implement some functionality. You can get my code by merging the branch with yours, and implement the functionality I asked. Once you implement it and commit the changes in your repository, I'll again work on the kernel on some other branch..

Status so far - our kernel boots into C code

So far, I have managed to write: See osdev barebones

1. x86/boot.S : containing seven lines of 32-bit x86 assembly instructions to:
   - set the stack pointer,

     ```
     movl  $tmpstack_bottom, %esp
     ```
   - clear flags,

     ```
     pushl $0
     popf
     ```
   - call the C function

     ```
     call  core_boot
     ```

make

- Syntax:

```
bash$ make <target> B=<release/debug>
where target =
  iso         : create boot cd
  exe         : build kernel (default)
  qemu        : run qemu
  qemu-gdb    : qemu with gdb
  qemu-direct: directly run the kernel
  qemu-direct-gdb: directly run the kernel
```

- Usage: make iso / make qemu / make qemu-gdb B=debug

On Boot

- CPU sets cs:ip to 0xffff:0x0000 and starts executing code from this location(BIOS ROM is memory mapped at this location. When CPU tries to load the instruction from this location, cache and memory will be bypassed, and instructions will be directly loaded from ROM).
- *CPU starts executing BIOS code directly from ROM.*
- BIOS code initializes cache, RAM and other peripherals
- BIOS code installs its handlers by modifying Interrupt descriptor table(IDT) to provide services for bootloader
- BIOS loads the boot loader(grub2) code from the boot disk at

Analyzing tracefile

I've enabled qemu's instruction tracing. So after executing 'make qemu', a trace file created named qemu.log in the current working directory.

When looking at the tracefile(qemu.log), please skip the initial bios instructions

```
----------------
IN:
0xfffffff0:  ljmp   $0xf000,$0xe05b
```

Our kernel's instruction trace

Towards the end you can see our kernel's instruction trace. For example:

```
----------------
IN:
0x00100050:  mov    $0x104080,%esp
0x00100055:  push   $0x0
0x00100057:  popf
0x00100058:  call   0x1040a0
----------------
IN: some_host
```

Boot our kernel from your laptop

Optional: Multiboot specification specifies the interface between boot loader(eg: grub) and the kernel. You can also boot our kernel from your laptop, by using any multiboot combatible boot loader.

For example: On grub2, I press 'c' to enter command prompt, and type:

```
(grub2) multiboot (hd0,msdos5)/home/alice/hohlabs/_tmp
(grub2) boot
```

Setup

Tools

Please ensure you have latest version of:

- qemu (package: qemu qemu-system)
- g++ (package: g++-multilib >=4.7)
- git (package: git-all)
- grub2 (package: grub2 grub-pc-bin)
- boost library (package: libboost-all-dev)
- xorriso (to create iso image. Otherwise you'll get a warning that)
- coreutils(for makefile)

Clone the repository

Since we both will work on this kernel, we need to have a version control system. We'll use git as our version control system. Please clone the repository to your local directory

```
user@host:~$ git clone ssh://<user>@palasi.cse.iitd.ac
user@host:~$ cd hohlabs
user@host:~/hohlabs$
```

Procedure

For each parts, do

1. Please get the changes done by Alice by merging the corresponding branch to your master branch

   ```
   user@host:~/hohlabs$ git pull
   user@host:~/hohlabs$ git merge origin/<branch_name>
   ```

   For example, to get first part, do:

   ```
   user@host:~/hohlabs$ git pull
   user@host:~/hohlabs$ git merge origin/vgatext
   ```

2. *Modify the files under the directory "labs" only* to add the

Submission

- To submit the assignment, from palasi, do:

```
user@host:~/hohlabs$    git commit -a -m "your log me
user@palasi:~/hohlabs$ os-submit-lab <labid>
```

- Can be submitted from palasi only.
- Resubmissions are allowed.

HoH: Primitives - CSL373/CSL633 Lab1

Overview

In this first part, we'll look into basic primitives required for writing an OS.

- Evaluation:
  - Code component:
    - *NOTHING : 0* Not working
    - *PARTIAL : 1* Partial/buggy - TA is able to find atleast one bug in your code
    - *TYPO : 1.5* Code is not clean
    - *CORRECT : 2* Working code
  - Viva component:

MMIO

MergeRequest

I've added few more code in origin/vgatext branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/vgatext
```

Aim

In this part, we'll program a memory mapped device while enhancing our kernel by adding the functionality to display "Hello, world!".

Information

In VGA text mode, 16 bit (2 bytes) of information is stored for each screen character and is stored in row-major order. First byte(MSB) is the ASCII code of the screen character and the next byte(LSB) encodes background(4 bit: msb) and foreground color(4 bit: lsb). Color: 0x0 corresponds to black pallete, 0x7 corresponds to white pallete, 0x1 corresponds to blue pallete.

Usage

I've added few lines of C code in x86/main.cc:

```
for(i=0;i<sizeof mesg;i++){
  vgatext::writechar(i, mesg[i], bg_color, fg_color
}
```

Define

You need to define the following functions in labs/vgatext.h

```
void writechar(int loc, uint8_t c, uint8_t bg, uint
```

Arguments of vgatext::writechar:

- loc: location of screen character to be written,
- c: ascii code of the character to be written(8 bit)
- bg: background color(4 bit)
- fg: foreground color(4 bit)
- base: the memory mapped address of the vga text buffer

Given

To help you with mmio, I also added util/io.h which has following functions:

```
mmio::read8(base,byte_offset)
mmio::write8(base,byte_offset,8 bit value)
mmio::read16(base,byte_offset)
mmio::write16(base,byte_offset,16 bit value)
mmio::read32(base,byte_offset)
mmio::write32(base,byte_offset,32 bit value)
```

Tip

- You might find mmio::write16/mmio::write8 useful for implementing vgatext::writechar.
- Note that both mmio::write8 and mmio::write16 takes byte offset as an argument.
- If you're using mmio::write16, please take care of endianness - x86 is little endian.
- When using bit shift operations, we recommend you to use unsigned integer types

Turn in

You're required to implement vgatext::writechar() in labs/vgatext.h

Check

The kernel shall print ‘Hello, world!’ in the top left corner of the screen.

Note

- Expected: 1-2 line of C++ code. If you find yourself adding more than 10 lines of code in this part, please raise an alarm. After 10 logical lines of code, each logical line of code you add, 10% of mark will be substracted.
- Optional: Boot our kernel from a PC/laptop instead of qemu.

Note

- Endianness is a property of CPU - it's about what should be the memory contents "when a CPU executes Write instruction to memory" or what is the value of register if we execute read instruction from memory.
  When we say: MSB: char(8 bits) and LSB: bgfg (8 bits) - it's independent of endianness.
  It means: first byte should be char. and next byte is bgfg.
  It specifies what should be the memory contents after you execute the CPU instruction. And depending on the target CPU's (in which your OS is written for) endianness, you need

Demo Tip

Be prepared to answer following viva questions:

- How to program with memory mapped devices?
- What happens between 'programming from cpu' to 'device recieving the command/data' (Refer: Computer Architecture course)
- How to boot your kernel into C/C++ code?

PMIO

MergeRequest

Now it's my turn. I've added few more code in origin/serial branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/serial
```

Aim

In this part, we'll program an I/O mapped device while enhancing our kernel by adding debugging routines which will print debug messages to serial port.

Information

Serial port aka pc16550d uart(universal asynchronous receiver transmitter). In pc16550d uart,

Registers:

- the "transmitter holding" register of size 8 bits(1 byte) is I/O mapped at zeroth offset, and
- the "line status" register of size 8 bits(1 byte) is I/O mapped at fifth offset.

The line status register has several fields (in lsb order):

Usage

I've added hoh_debug macro in util/debug.h, which will convert the arguments into string and call serial::print for each character in the string. Usage:

In x86/main.cc: I've added the following line.

```
hoh_debug("Hello, serial!");
```

hoh_debug macro will expand to a call to serial::print()

I also added serial::print function in util/debug.cc:

```
void serial::print(char c){
```

Define

You need to define the following functions in labs/serial.h

```
bool is_transmitter_ready(io_t baseport);
void writechar(uint8_t c, io_t baseport);
```

Given

To help you with I/O(in and out asm), I had added following functions in util/io.h:

```
io::write8(baseport, offset, 8 bit value)
io::write16(baseport, offset, 16 bit value)
io::read8(baseport,offset)
io::read16(baseport,offset)
```

Tip

- You may find: io::read8(baseport,offset) and io::write8(baseport, offset, value) defined in util/io.h useful.
- When using bit shift operations, we recommend you to use unsigned integer types

Turn in

You're required to implement serial::is_transmitter_ready() and serial::writechar() in labs/serial.h

Check

The kernel shall print ‘Hello, serial!’ in your terminal.

Note

- Expected: 2-4 line of C++ code. If you find yourself adding more than 20 lines of code in this part, please raise an alarm. After 20 logical lines of code, each logical line of code you add, 5% of mark will be substracted.

Demo Tip

Be prepared to answer following viva questions:

- How to program with io mapped devices?
- What happens between ‘programming from cpu’ to ‘device recieving the command/data’ (Refer: Computer Architecture course)

Abstract mmio/pmio

MergeRequest

I've added few more code in origin/keyboard branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/keyboard
```

Aim

In this part, we'll look at one way of abstracting out details of mmio::read8 vs io::read8 while enhance our kernel by adding a simple keyboard driver.

Information

In Keyboard(8042, name=lpc_kbd), there are two main registers

- status register: size="8 bits" The status register has several fields

```
name="perr",     size="1 bit", description="Parity
name="timeout",  size="1 bit", description="General
name="aobf",     size="1 bit", description="Auxilia
name="is",       size="1 bit", description="Inhibit
name="cd",       size="1 bit", description="Command
name="sf",       size="1 bit", description="System
name="ibf",      size="1 bit", description="Input_b
```

Usage

```
core_loop_step():
    if(!has_key(dev)){
      return;
    }
    input=get_key(dev);
    hoh_debug("Got key: "<<input);

core_loop():
    repeat core_loop_step
```

Define

You need to define the following functions in labs/keyboard.h

```
bool has_key(lpc_kbd_t& dev);
uint8_t get_key(lpc_kbd_t& dev);
```

Given

Following functions are defined in generated/lpc_kbd.h(generated from spec/lpc_kbd.spec using modified mackerel):

```
lpc_kbd_status_rd()              : return the value of "st
lpc_kbd_status_obf_extract()     : extract "obf" field fro
lpc_kbd_input_rd()               : return the value of "in
```

Tip

Trivial.

Turn in

You're required to implement the required functions in labs/keyboard.h

Check

Kernel shall print scancode of each key pressed in your terminal(hoh_debug).

Note

- Expected: 2-4 line of C++ code. If you find yourself adding more than 10 lines of code in this part, please raise an alarm. After 10 logical lines of code, each logical line of code you add, 10% of mark will be substracted.

Demo Tip

Be prepared to answer following viva questions:

- Is keyboard memory mapped(mmio::read8) or io mapped(io::read8)?
- What's the offset of status register and input register from basemem/baseport?
- Which bits corresponds to obf field in status regiser? How to extract those bitfields from value of status register?
- Endianness?
- Is knowing answer to above questions necessary while using the

Credits

Device interface functions in generated/lpc_kbd.h are generated by a modified version of mackerel.

kShell

MergeRequest

I've added few more code in origin/shell branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/shell
```

Aim

In this part, we'll look at one design approach while implementing a toy shell supporting builtin functions only.

- You need to implement the shell by implementing the given interfaces(in labs/shell.h and labs/shell.cc).
- You are *not* allowed to modify the interface and it's usage in x86/main.cc.
- You are *not* allowed to use any global variables or static variables in your functions.
- To make sure we have a personalized UI for each student, exact user interface is open. So be creative!

Information

Reuses previous parts of this series to create a shell.

Usage

```
core_loop_step():
    if user has pressed key, get the key and do:
        shell_update(ro: key, rw: shell_state);

    // execute shell for one time slot to do some compu
    shell_step(rw: shell_state);

    // shellstate -> renderstate: compute render state
    shell_render(ro: shell_state, wo: render_state);
```

Define

You need to define the following structures in labs/shell.h

```
// state for shell
struct shellstate_t{
};
// state required to render( for ex: intermediate resu
struct renderstate_t{
};
```

You also need to define the following functions in labs/shell.cc

```
void shell_init(shellstate_t& state);
```

Given

NA.

There're several helper functions given in the labs/shell.cc. When you execute, you'll be seeing a simple menu based interface. You may or may not use those functions. Please feel free to create your own interface.

Tip

- See the comments inside labs/shell.cc
- shell_step:
  - you may have to have a statemachine to know whether computation is in progress or not etc. (store the state in shellstate_t. pass the state to renderstate - if you want to enable/disable the menu item)
- Prefer iterative over recursive - stack size is limited to 4KB
- Use integer arithmetic instead of floats.

Turn in

You're required to define the structures in labs/shell.h and implement the required functions in shell.cc

Check

A simple shell with several builtin commands including a “long computation task” and a status bar showing the “number of key presses” so far.

Note

Have you noticed that:

- Select long computation task
- Press a key
- Status bar will get updated only after the long computation task is finished?

ie. System latency to keyboard events is high - we'll improve this in next part.

Demo tip

Be prepared to answer following viva questions:

- What are the advantages and disadvantages of this design? How to improve? What are other alternative approaches?
- What happens between you pressing a key in keyboard and it appearing on screen(if it appears).

Stackless Coroutine

MergeRequest

I've added few more code in origin/coroutine branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/coroutine
```

Aim

In this part, we'll learn about "asymmetric-stackless coroutines" while enhancing our kernel to make it responsive to key presses while long computation task is running.

- You shall implement the long computation task as a stackless coroutine using the given APIs and add a new menu item/builtin command for the same.
- On key press, the status bar shall be updated with 'the number of keys pressed so far' while this long computation task is running(not after it finishes).
- If we select older menu item, shell still take seconds to respond

Information

Coroutines are a generalization of coroutines which allows explicit suspend and resume operations(yield and call). Coroutines can be used for nonpremptive multitasking(fibers), event loop, and light weight pipes(producer consumer problem).

Definition of coroutine from Coroutines: A Programming Methodology, a Language Design and an Implementation(1980):

```
For the purposes of this thesis, the following will be rega
the fundamental characteristics of a coroutine:
(1) the values of data local to a coroutine persist betwee
```

Define

You need to define the following structures in labs/coroutine.h

```
// state for your coroutine implementation:
struct f_t{
};
```

You also need to define the following functions in labs/coroutine.cc

```
shell_step_coroutine(shellstate_t&, coroutine_t&, f_t&)
```

You also need to enhance your shell implementation in labs/shell.h

Usage

```
core_loop_step():
    if user has pressed key, get the key and do:
        shell_update(ro: key, rw: shell_state);

    // execute shell for one time slot to do some compu
    shell_step(rw: shell_state);

    // execute shell for one time slot to do some compu
    shell_step_coroutine(rw: shell_state, f_coro, f_loca
```

Given

Following functions are defined in util/coroutine.h:

```
coroutine_t          : internal data structure to save th
coroutine_reset()    : initialize/reset coroutine_t

h_begin()            : begin coroutine ( jump to saved st
h_yield()            : yield           ( save the state,
h_end()              : end             ( infinitely call
```

Example usage of coroutines

```
//
// state of function f to be preserved across multiple
//
struct f_t{
 int i;
 int j;
};

//
// first time you call f(), it'll
```

Tip

- void f(T* px) === void f(T& x)
- Stackless => No recursion!

Turn in

- You shall implement the long computation task as a stackless coroutine using the given APIs.
- Add a new menu item/builtin command for calling it.

Check

- On key press, the status bar shall be updated with 'the number of keys pressed so far' while the long computation task is running(not after it finishes).
- Result of both the menu items should be same.

Note

- You're required to initialize the coroutine from shell_step_coroutine(). You may have a statemachine (DEAD,START,READY), and on state transition from DEAD->START, you may want to initialize the coroutine.

Note

- Have you noticed that we need to save value of local variables in a structure and that stack is not preserved? In the next part, we'll implement a stack for each coroutines, and let local variables stored on stack instead of new structure.

Fiber

MergeRequest

I've added few more code in origin/fiber branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/fiber
```

Aim

In this part, we'll learn about "fibers" while enhancing our kernel to make it responsive to key presses while long computation task is running.

- You shall implement the long computation task as a fiber using the given APIs and add a new menu item/builtin command for the same.
- On key press, the status bar shall be updated with 'the number of keys pressed so far' while this long computation task is running(not after it finishes).
- Result of all three menu items should be same

Information

Usage

```
core_loop_step():
    if user has pressed key, get the key and do:
        shell_update(ro: key, rw: shell_state);

    // execute shell for one time slot to do some
    shell_step(rw: shell_state);

    // execute shell for one time slot to do some
    shell_step_coroutine(rw: shell_state, rw: f_co
```

Define

You need to define the following functions in labs/fiber.cc

```
shell_step_fiber(shellstate_t&, addr_t& main_stack, addr
```

You also need to enhance your shell implementation in labs/shell.h

```
update shellstate_t and renderstate_t structure:
  for handling fiber state, and
  new menu item for long computation task as fibers
```

You also need to enhance your shell implementation in labs/shell.cc

Given

```
stack_reset(f_stack,f_array,f_arraysize,f_start,f_a
stack_resetN(f_stack,f_array,f_arraysize,f_start,f_
stack_saverestore(from_stack,to_stack)
```

Example usage of fibers

```
void f(addr_t* pmain_stack, addr_t* pf_stack, int*
  addr_t& main_stack = *pmain_stack; // boilerplat
  addr_t& f_stack    = *pf_stack;
  int& ret           = *pret;
  bool& done         = *pdone;

  int i;
  int j;
```

Extra information

```
//
// Switch stacks.
//
// Algo:
//   1. Save _c's context to stack,
//   2. push ip of _c's restore handler
//   3. switch stacks
//   4. execute ip of _n's restore handler to reste
//
```

Tip

NA

Turn in

- You shall implement the long computation task as a fiber using the given APIs.
- Add a new menu item/builtin command for calling it.

Check

- On key press, the status bar shall be updated with 'the number of keys pressed so far' while the long computation task is running(not after it finishes).
- Result of all the three menu items should be same.

Note

- To achieve responsiveness, we've to add yield points explicitly. Sometimes, it may not be easy - can we trade efficiency and implment pre-emptive scheduling? Yes, But Pre-emption requires support for timers. To use timers, we need to have support for interrupts. which means we need to write interrupt handlers and program Interrupt Descriptor Tables(IDTs)
- Before we do so, let's first implement support for multiple non-premptive threads.

Non-preemptive scheduling

MergeRequest

I've added few more code in origin/fiber_schedular branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/fiber_schedul
```

Aim

In this part, we'll learn about non-preemptive sheduling while enhancing our shell to support mulitple pending long computation task.

- You shall support atleast two additional long computation tasks
- For these additional long computation tasks:
  - You shall support multiple pending long computation tasks
  - Add menu item/builtin command for calling additonal tasks(Retain previous menu items).
  - Same command/menu item may be entered multiple times
  - Each command may be queued at max 3 times.

Information

NA

Usage

```
core_loop_step():
    if user has pressed key, get the key and do:
        shell_update(ro: key, rw: shell_state);

    // execute shell for one time slot to do some
    shell_step(rw: shell_state);

    // execute shell for one time slot to do the c
    shell_step_coroutine(rw: shell_state, rw: f_co
```

Define

You need to define the following functions in labs/fiber_schedular.cc

```
shell_step_fiber_schedular(shellstate_t&, addr_t stacks
```

You also need to enhance your shell implementation in labs/shell.h

```
update shellstate_t and renderstate_t structure:
  for handling schedular state, etc
```

You also need to enhance your shell implementation in labs/shell.cc

```
atleast two long computation tasks.
```

Given

NA

Tip

This is the goal: So far, we have the capability to ru

1. G:: GArg -> GResult
2. H.:: HArg -> HResult

We also want to support multiple invocations of these

Now, we have to store 3*(GArg,GResult) and 3*(HArg,HRe

What should be a good data structure for storing these

Turn in

- You shall support multiple pending long computation tasks
- Add few more menu item/builtin command for calling it.

Check

Note

Optional Design check

To test how good is your design:

- commenting out shell_step_fiber:
  - is it equivalent to take fiber computation taking infinite amount of time
- commenting out shell_step_coroutine():
  - is it equivalent to take coroutine computation taking infinite amount of time

etc.

Preemption (threads)

MergeRequest

I've added few more code in origin/preemption branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/preemption
```

Aim

In this part, we'll learn about "preemption" while enhancing our kernel to make it responsive to key presses while long computation task is running.

- You shall enhance the fiber implementation by adding preemption.
- You need to write a part of trap handler - ring0_preempt - which should switch stack to 'main_stack'
  - We would like to reuse shell_step_fiber_schedular to do the scheduling.
- You shall program one shot LAPIC timer to raise an interrupt

Information

- Lecture videos:
  - Trap handlers
  - Context switch
- FXSAVE and FXRSTOR assembly instructions: To save and restore FPU/SIMD registers

Usage

NA

Define

You need to define the following structures in labs/preempt.h

```
// preempt_t : State for your timer/preemption handler
struct preempt_t{
};
```

You also need to define the following functions in labs/preempt.h

```
//
// _name: label name
// _f   : C function to be called
//
```

Given

- lapic.reset_timer_count(N); to generate a timer interrupt after N timer ticks (N=0 to stop)

Tip

NA

Turn in

Check

- On key press, the status bar shall be updated with 'the number of keys pressed so far' while the long computation task is running(not after it finishes).
- Result of all the three menu items should be same.
- On demand timer ticks: No timer ticks if there're no fibers running.

Note

NA

SPSC Queue: Execute task on remote core

MergeRequest

I've added few more code in origin/multicore branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/multicore
```

Aim

In this part, we'll learn about multicore programming while enhancing our kernel to schedule a task on another core

- You'll have to implement Leslie Lamport's portable lock-free single-producer single-consumer circular buffer
- Size of buffer will always be a power of 2.

Information

- Leslie Lamport's Proving the Correctness of Multiprocess Programs
- gcc atomic intrinsics
- C11/C++11 atomics

Usage

Define

Given

Tip

Turn in

Check

Note

User mode

MergeRequest

I've added few more code in origin/ring3 branch. Please merge it with your master branch

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/ring3
```

Aim

In this part, we'll learn about ELF headers, page table handling and user mode switching while enhancing our kernel to load arbitary user program and execute it.

- ELF headers
- Page table
- Long computation task in user mode
- We'll implement exit system call later. We'll verify correctness by looking at qemu's instruction trace.

Information

Please see lecture videos:

- ELF headers
- Page table
- First user program

Usage

Define

Given

`switch_toring3`

Tip

Turn in

Check

- You need to verify user program execution by looking at qemu.log

Note

Ring3 Preemption

MergeRequest

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/ring3_preemption
```

Aim

In this part, we'll learn about preempting user program/ process context switch while enhancing our kernel to make it responsive to key presses while long computation task is running in ring3/user mode.

Information

Please see following lecture videos: - Process context switch

Usage

Define

Given

Tip

Turn in

Check

Note

Exit system call

MergeRequest

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/to_kernel
```

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

Signals - Interrupting user mode program

MergeRequest

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/from_kernel
```

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

Schedular

MergeRequest

```
user@host:~/hohlabs$ git pull
user@host:~/hohlabs$ git merge origin/schedular
```

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

End of lab one

```
Please make sure you submit the feedback form
--
Regards,
Alice H
Hoh labs
```

Systemcalls: create_process and page_remap system calls
User IPC: send_message
File system server in user mode
Virtual memory in user mode

II

HoH: Kernel Design - CSL373/CSL633 Lab2

Overview

In this lab, we'll use the components we implemented in previous lab to make a fully functional kernel. We'll first enhance our kernel by implementing:

- Sending a message to kernel
- Receiving a message from kernel
- Sending and recieving between two apps

We'll test our kernel by having two applications: Another thing to think about is why there are no problems with PIC code using the small code model. The reason is that the GOT is always located in

Systemcalls: create_process and page_remap system calls

MergeRequest

Systemcalls: create_process and page_remap system calls
User IPC: send_message
File system server in user mode
Virtual memory in user mode

Aim

Information

Usage

Systemcalls: create_process and page_remap system calls
User IPC: send_message
File system server in user mode
Virtual memory in user mode

Define

Systemcalls: create_process and page_remap system calls
User IPC: send_message
File system server in user mode
Virtual memory in user mode

Given

Tip

Systemcalls: create_process and page_remap system calls
User IPC: send_message
File system server in user mode
Virtual memory in user mode

Turn in

Check

Note

User IPC: send_message

MergeRequest

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

File system server in user mode

MergeRequest

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

Virtual memory in user mode

MergeRequest

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

End of lab two

```
Please make sure you submit the feedback form
--
Regards,
Alice H
Hoh labs
```

Systemcalls: create_process and page_remap system calls
User IPC: send_message
File system server in user mode
Virtual memory in user mode

III

HoH: Advanced OS - CSL633 Lab3

Overview

- .
- .

kernel: Verification of User's Long computation task(Trust by verification)

MergeRequest

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

Uniform schedular : tasks, coroutines, threads, user pgm

MergeRequest

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

Shell in Ring 3 : User level Schedular

MergeRequest

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

H: Introduction to H

MergeRequest

Aim

Information

Usage

Define

Given

Tip

Turn in

Check

Note

End of lab three

```
Please make sure you submit the feedback form
--
Regards,
Alice H
Hoh labs
```

IV

*