CS372H: Spring 2008 – Midterm 1

Instructions

- This midterm is closed book and notes.
- If a question is unclear, write down the point you find ambiguous, make a reasonable interpretation, write down that interpretation, and proceed.
- State your assumptions and show your work. **Write brief, precise, and legible answers.** Rambling brain-dumps are unlikely to be effective. **Think before you start writing** so that you can crisply describe a simple approach rather than muddle your way through a complex description that "works around" each issue as you come to it. **Perhaps jot down an outline** to organize your thoughts. And remember, a **picture can be worth 1000 words.**
- For full credit, show your work and explain your reasoning.
- There are three problems on this exam
- Question 2 has four parts.
  Answer any **three** of the **four** parts of question 2.
- Write your name on this exam

1 Virtual memory (20)

Suppose you need to implement the OS for a “smart” phone that provides a supervisor mode and facilities to enter supervisor mode on interrupt, exception, and trap. The hardware implements a RISC-style load/store instruction set. However, the hardware only has primitive virtual memory support—it has a two base registers and two bounds registers for applications and a separate pair of base and bounds registers used when in supervisor mode. The hardware has no paging support. Unfortunately, all of your application software relies on 8 segments spread across a sparse address space. How could you modify the OS to support applications that require multiple segments per address space using this hardware?

Solution: Load code base and bounds and also one data segment base and bounds into base and bounds registers; if an application accesses data from a different data segment, you will get a seg fault (exception). On such an exception, the OS should unload the current data base/bounds and load the ones needed for the current instruction. Slow, but it will work.

2 Project (40)

This problem has four parts.
Answer any **three** of the **four** parts of this problem.

1. In boot/main.c, here is bootmain:

```
1  void
2  bootmain(void)
3  {
4     struct Proghdr *ph, *eph;
5
6     // read 1st page off disk
7     readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
8
9     // is this a valid ELF?
10    if (ELFHDR->e_magic != ELF_MAGIC)
11         goto bad;
12
13    // load each program segment (ignores ph flags)
14    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
15    eph = ph + ELFHDR->e_phnum;
16    for (; ph < eph; ph++)
17         readseg(ph->p_va, ph->p_memsz, ph->p_offset);
18
19    // call the entry point from the ELF header
20    // note: does not return!
21    ((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))();
22
23 bad:
24    outw(0x8A00, 0x8A00);
25    outw(0x8A00, 0x8E00);
26    while (1)
27          /* do nothing */;
28 }
```

Why in line 21 do we call the loaded kernel's entry point function using address (ELFHDR->e_entry & 0xFFFFFF)) rather than calling the function using address (ELFHDR->e_entry))?

Solution: The link address (ELFHDR→e_entry = 4GB - 256MB + 1MB) does not match the load address (1MB); the and operation shifts the jump address to jump to the load address (where the code actually is) rather than the link address

Explain how the kernel code is written to ensure that the code invoked in this way works.

Solution: The kernel code begins with entry.S, assembly code that is location independent (so load != link isn't a problem). entry.S sets up basic segmentation so that once it longjumps to set the code segment register, the code appears to have been loaded it the link address

2. Recall your implementation of mon_backtrace() in lab 1, where the instructions said:

   The backtrace function should display a listing of function call frames in the following format:

   ```
   Stack backtrace:
     ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031\\
     ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061\\
     ...
   ```

   The first line printed reflects the currently executing function, namely mon_backtrace itself, the second line reflects the function that called mon_backtrace, the third line reflects the function that called that one, and so on. You should print all the outstanding stack frames. By studying kern/entry.S you'll find that there is an easy way to tell when to stop.

   How were you able to determine "when to stop"?

   Solution: the "saved" ebp == 0 for the final frame

3. A question in lab 2 asks: *After check_boot_pgdir(), i386_vm_init() maps the first four MB of virtual address space to the first four MB of physical memory, then deletes this mapping at the end of the function. Why is this mapping necessary? What would happen if it were omitted and why?*

   Answer these questions

   Solution: When we turn on paging but before we turn off segmentation, segmentation maps VA KERNBASE to LA 0. We need to make sure LA 0 still maps to the desired physical address, so we install some page mappings from LA 0 to kernel memory. Then, we turn segmenting off, and va KERNBASE maps to la KERNBASE, and all is good –¿ we can remove the low page mappings
   If it were omitted, then crash b/c when paging turned on, no mappings for la in which kernel is executing code

4. The file inc/memlayout.h defines KSTACKTOP to be 0xefc00000 (= 4GB - 256MB - 4MB = KERNBASE - 4MB). KSTACKTOP is not the kernel stack address upon entry to i386_init().

   - Why couldn't we use KSTACKTOP as the kernel stack when the kernel was first invoked?

     Solution: Before page tables are set up, physical mememory is mapped starting at KERNBASE, so KSTACKTOP would not map to any allocated memory

   - How was the kernel's stack pointer address set before entry to i386_init and how/where was the memory region for this initial stack allocated?

     Solution: bootstack is a region of meory defined by linking instructions so the binary image includes some pages for kernel stack at location boostack...boostacktop. entry.S changes stack pointer to point at bootstacktop to initialize the kernel stack.

   - When and how is the kernel's stack address changed to KSTACKTOP?

     Solution: The kernel stack pointer is set to KSTACKTOP in trapentry.s – this code is invoked on a trap, and so the kernels stack is changed to point to KSTACKTOP for the frist time after a trap. Memory at KSTACKTOP is same memory as bootstack (but remapped via paging)

3 Multi-threaded programming (40)

Jurassic Park consists of a dinosaur museum and a park for safari riding. There are m passengers and n single-passenger cars. Passengers wander around the museum for a while, then they line up to take a ride in a safari car. When a car is available, it loads the one passenger it can hold and rides around the park for a random amount of time. If the n cars are all out riding passengers around, then a passenger who wants a ride waits in a FIFO line; if a car is ready to load but there are no waiting passengers, then the car waits in a FIFO line.

Thus, the main function for a visitor thread is

```
void visitor_main(...){
  int carId;
  museum->wander();           // You don't need to write this
  carId = ride->waitForCar(); // Returns id of car I will ride 0..n-1
  park->sightsee();         // You don't need to write this
  ride->getOutOfCar(carId); // Postcondition: car can get in line
}
```

and the main function for a car thread is

```
void car_main(...){
  ride->waitForPassenger(myId); // Postcondition: passenger in car
  ride->cruiseUntilPassengerGetsOut(myId); // Postcondition: can get in line
}
```

Implement the ride object using the framework below. **Your solution must follow the coding standards specified in class.**

List the member variables for a ride object:

Solution:

```
Lock mutex;
Cond carReady;
Cond passengerIn;
Cond passengerOut;
const int n = NCARS;
int carState[n]; // Each entry initialized to FREE
int nextTicketToGive = 0;
int nextTicketReady = 0;
int carsWaiting = 0;
List waitingCars;
int frontCarLoaded = 0;
```

int ride::waitForCar() // Postcondition: I am at head of line; Returns id of car I will ride 0..n-1

Solution:

```
mutex->lock();
int myTicket = nextTicketToGive++;
while(carsWaiting == 0 OR myTicket != nextTicketReady){
    carReady.wait(&mutex);
}
int myCar = waitingCars->getHead();
frontCarLoaded = true;
carState[myCar] = BUSY;
passengerIn.broadcast(&mutex);
mutex->unlock();
return myCar;
```

ride::getOutOfCar(int carId) // Postcondition: car can get in line

Solution:

```
mutex->lock();
carState[carId] = FREE;
passengerOut.broadcast(&mutex);
mutex->unlock();
```

ride::waitForPassenger(int carId) // Postcondition: passenger in car

Solution:

```
mutex->lock();
carsWaiting++;
waitingCars.add(carId);
carReady->broadcast(&mutex);
while(!frontCarLoaded && waitingCars->getHead() != carId){
      passengerIn.wait(&mutex);
}
frontCarLoaded = 0;
carsWaiting–;
waitingCars.remove(carId);
nextTicketReady++;
carReady.broadcast(&mutex);
mutex->unlock();
```

ride::cruiseUntilPassengerGetsOut(int carId) // Postcondition: can get in line

Solution:

```
mutex->lock();
while(carState[carId] != FREE){
      passengerOut.wait(&mutex);
}
mutex->unlock();
```