cs5460/6460: Operating Systems

Final recap, sample questions

Anton Burtsev

April, 2025

1. Stacks and calling conventions

Assume you have the following source code:

PollEv.com/antonburtsev

```
int func3(int d, int e, int f) {
      return d * e * f + 5;
}
int func21(int b, int c) {
      if (b == c) {
            return func3(b, c, 0xCCC);
      }
      return b * c;
}
int func22(int b, int c) {
      return func3(b, c, 0xDDD);
}
int func11(int a) {
      return func21(a, a) * func3(a, 0xBA, 0xBB);
}
int func12(int a) {
      return func22(a, a) * func3(a, 0xBA, 0xBB);
}
int main() {
      return func11(0xA) + func12(0xB);
}
```

Your best friend (who already passed this class last year) sets a breakpoint in one of the functions, runs the program, hits the breakpoint, and dumps the stack getting the following:

```
0xffd6c380: 0xffd6c398
0xffd6c384: 0x565c8567
0xffd6c388: 0x0000000a
0xffd6c38c: 0x000000ba
0xffd6c390: 0x000000bb
0xffd6c394: 0x00000000
0xffd6c398: 0xffd6c3a8
0xffd6c39c: 0x565c85ae
0xffd6c3a0: 0x0000000a
0xffd6c3a4: 0x00000000
0xffd6c3a8: 0x00000000
```

(a) (5 points) Find out in which function the breakpoint was triggered (explain your work). A hint from a friend: stack grows down.

1. Stacks and calling conventions

Assume you have the following source code:

```
int func3(int d, int e, int f) {
      return d * e * f + 5;
}
int func21(int b, int c) {
      if (b == c) {
            return func3(b, c, 0xCCC);
      }
      return b * c;
}
int func22(int b, int c) {
      return func3(b, c, 0xDDD);
}
int func11(int a) {
      return func21(a, a) * func3(a, 0xBA, 0xBB);
}
int func12(int a) {
      return func22(a, a) * func3(a, 0xBA, 0xBB);
}
int main() {
      return func11(0xA) + func12(0xB);
}
```

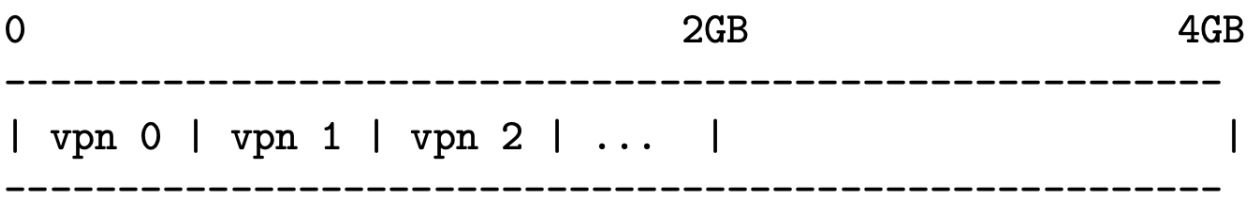
Your best friend (who already passed this class last year) sets a breakpoint in one of the functions, runs the program, hits the breakpoint, and dumps the stack getting the following:

```
0xffd6c380: 0xffd6c398
0xffd6c384: 0x565c8567
0xffd6c388: 0x0000000a
0xffd6c38c: 0x000000ba
0xffd6c390: 0x000000bb
0xffd6c394: 0x00000000
0xffd6c398: 0xffd6c3a8
0xffd6c39c: 0x565c85ae
0xffd6c3a0: 0x0000000a
0xffd6c3a4: 0x00000000
0xffd6c3a8: 0x00000000
```

(b) (5 points) Explain each value on the stack (you can annotate next to the stack drawing and/or provide some explanation here)

2. xv6 address space

The following figure represents an address space of an xv6 process:



The page with virtual page number 0 (vpn 0) is mapped to physical address `0x100_0000` (`16_777_216`).

(a) (5 points) If the physical page at address (`0x100_0000`) is mapped at some other virtual addresses, what are those virtual addresses (explain your answer)?

PollEv.com/antonburtsev

(b) (5 points) Assume the process text is less than 1 pagesize. The stack of the process above is in vpn 2. Which flags are set in the page table for the pages vpn 0, vpn 1, and vpn 2 (explain your answer)?

	PTE_P	PTE_U	PTE_W
VPN 0:			
VPN 1:			
VPN 2:			

3. Interrupts

The following is the listing of the `release()` function in the xv6 kernel.

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

(a) (5 points) What is the role of the `popcli()` function?

3. Interrupts

The following is the listing of the `release()` function in the xv6 kernel.

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

(b) (10 points) What happens if you comment out invovcation of `popcli()`.

4. xv6 initialization

   (a) (5 points) When the `swtch` function in the xv6 kernel executes for the very first time, where does it return?

4. xv6 initialization

   (a) (5 points) When the `swtch` function in the xv6 kernel executes for the very first time, where does it return?

   (b) (5 points) In a two-CPU system is it possible that the very first process executes on the second CPU?

   (c) (5 points) In a two-CPU system xv6 creates two GDTs, can you explain why one is not enough?

5. Isolation

Recall that in one of our homework assignments when we configured execution of the system call, we configured the Interrupt Descriptor Table (IDT) to be following:

```
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
```

(if you don't remember your homework, xv6 does the same)

(a) (5 points) Explain why we set the entry to be `DPL_USER`(3), what will happen if it is 0?

(b) (5 points) Recall we also added entries in global descriptor table (GDT) with `DPL_USER`. What are the purposes of those entries?

(c) (5 points) In order to go from kernel to user, scheduler must modify system state to run in user mode. How is it done?

6. System call interface

Alice adds the following program to xv6

```
#include "types.h"
#include "stat.h"
#include "user.h"


int
main(void)
{
  int n;
  char *argv[] = { "echo", "hello", 0 };
  printf(1, "start\n");

  for(n = 0; n < 10; n++) {
    fork();
    exec("echo", argv);
  }

  printf(1, "end\n");
  exit();
}
```

(a) (5 points) What are the possible outputs of this program?

CS 5965 - Advanced OS Implementation

cs5965 teaches advanced topics in operating systems through a hands-on engineering approach. As a student in this class **you will build a version of a small but functional operating system**. In contrast to previous years this is an implementation-heavy class. We will use Rust (although other programming languages are ok too) to boot into Rust and implement core pieces of the operating system: memory allocator, ELF loader, page table and address spaces, processes and finally context switching and scheduling.

Note: This class is different from CS 6465 2024. In the future the idea is to have two versions offered despite sharing the same name as the 2024 class The idea is that you can learn enough about modern commodity operating systems like Linux, hypervisors like KVM, Xen and VMware, container technologies and a range of cutting edge research topics that you can either get a job in this area or start working on competitive research.

You will study, in detail, organization of modern commodity kernel like Linux, commodity hypervisors like KVM and Xen, understand performance and security problems of modern operating systems. You will further understand major research directions in the areas of security, performance and reliability of operating systems. We will cover ideas of microkernels, exokernels, unikernels and library kernels, ideas of retrofitting isolation into commodity kernels, modern hardware isolation mechanisms, software fault isolation and WASM, basics of software verification and its potential for development of formally correct operating systems.

CS 5965 - Advanced OS Implementation

cs5965 teaches advanced topics in operating systems through a hands-on engineering approach. As a student in this class **you will build a version of a small but functional operating system**. In contrast to previous years this is an implementation-heavy class. We will use Rust (although other programming languages are ok too) to boot into Rust and implement core pieces of the operating system: memory allocator, ELF loader, page table and address spaces, processes and finally context switching and scheduling.

CS 5965 - 001 ADV OS Implementation

Class Details

Class Number: 20728 | Instructor: BURTSEV, ANTON | Component: Special Topics | Type: In Person | Units: 3.0
Requisites: Yes | Wait List: No | View Feedback

Learning how a modern operating system really works by reading, understanding, and modifying the source code for an OS kernel. Topics include scheduling, virtual memory, file systems, traps and interrupts, device drivers, concurrency control. Students will complete a number of programming assignments and also a more significant final project. Prerequisite: CS 5460

Days / Times
MoWe/03:00PM-04:20PM

Locations
JTB 120

Meets With
- CS 6465 001

WASM, basics of software verification and its potential for development of formally correct operating systems.

Thank you!