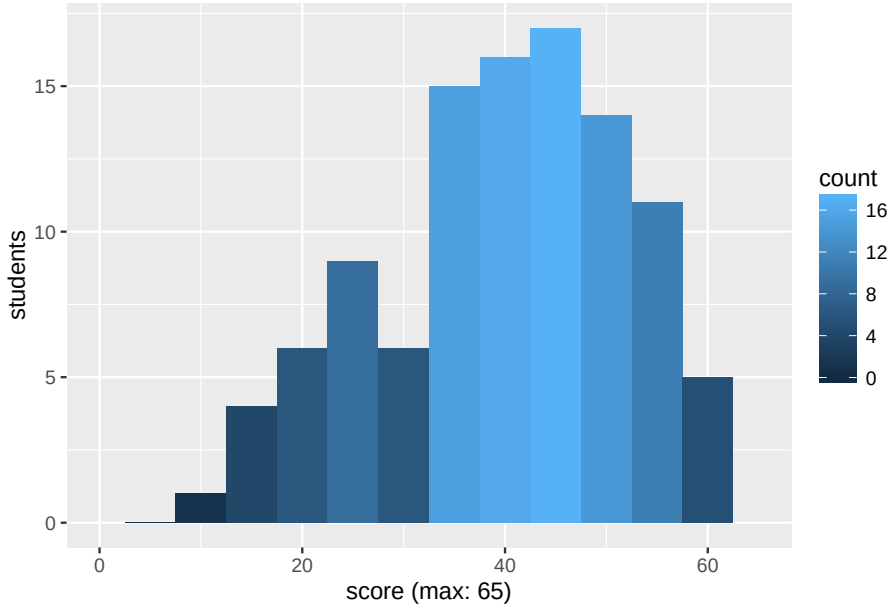
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2016

Quiz I Solutions

Mean 39.9 Median 40.0 Standard deviation 12.3 Kurtosis -0.71



I Context Switch

Homework 8 (user-level threads) uses this structure to hold information about each thread:

```
struct thread {
  int        sp;                /* curent stack pointer */
  char stack[STACK_SIZE];       /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
```

Ben Bitdiddle wonders why the `sp` element of the struct is needed. He plans to delete `sp` from `struct thread`:

```
struct thread {
  char stack[STACK_SIZE];       /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
```

He plans to have `thread_switch()` push the `%esp` register onto the "current" stack and pop it from the "next" stack. And he plans to modify `thread_create()` appropriately.

1. [5 points]: Explain why Ben will find it difficult or impossible to make this idea work.

Answer: `thread_switch()` must be able to find the correct place in the target thread's stack to pop registers from. Without the saved `sp` in `struct thread` it won't know where to look.

II xv6 Shell, Sleep, and Wakeup

Recall from Homework 2 that the shell uses the `wait()` system call to wait for each command to exit, at which point the shell prints another prompt. Look at the implementation of xv6's `wait()` system call in `proc.c`. Here's an abbreviated version of the xv6 code (with lots of lines omitted):

```
wait(void) {
  for(;;){
    if there is an exited child {
      clean it up ...;
      return pid; // return the child pid
    }

    if(this process has no children || proc->killed){
       ...
       return -1; // error
    }

    // wait for a child to exit.
    sleep(proc, ...);
  }
}
```

2. [5 points]: Is it possible for the `sleep()` to return even if there is no exited child? If yes, how can that happen?

Answer: `kill()` might wake up the process.

Suppose we re-arranged the code in the `for` loop to first check if the process has children, then sleep, and then check if there is an exited child:

```
wait(void) {
  for(;;){
    if(this process has no children || proc->killed){
       ...
       return -1; // error
    }

    // wait for a child to exit.
    sleep(proc, ...);

    if there is an exited child {
      clean it up ...;
      return pid; // return the child pid
    }
  }
}
```

This modification will cause `wait()` to act incorrectly in some circumstances.

3. [5 points]: Explain what can go wrong as a result of the modification.

Answer: If the process has just one child, and it has already exited (but the parent hasn't yet waited for it), the `sleep()` will wait forever. But `wait()` should return in that situation.

III xv6 Locking

Ben Bitdiddle thinks that xv6 kernel threads should be able to yield the CPU while holding locks. To test whether this works, he places an `acquire`/`yield`/`release` at the beginning of `syscall()` where it will be called a lot:

```
void
syscall(void)
{
  int num;

  acquire(&test_lock);
  yield();
  release(&test_lock);

  // the rest of syscall()...
}
```

No other code uses `test_lock`. Ben also deletes the panic calls in `sched()` (in `proc.c`), and deletes the `holding()` panics in `acquire()` and `release()`.

Ben boots his modified xv6 on qemu with a single core (`CPUS:=1` in the Makefile). xv6 prints `init: starting sh` but doesn't print a shell prompt; it does nothing, as if in a deadlock. When Ben looks at a backtrace with gdb, he sees the CPU is looping in the `acquire()` call he added to `syscall()`.

4. [5 points]: Explain a sequence of events in Ben's modified xv6 that could cause this to happen.

Answer: One process makes a system call, acquires `test_lock`, and yields. A second process runs, makes a system call, and loops in `acquire()` waiting for the lock. Interrupts are disabled by `acquire()`, so no timer interrupt will occur and thus the first process won't run and release the lock. This is a deadlock.

Now Ben tries his modified kernel on a two-core qemu (`CPUS:=2`). He gets a prompt this time and types a few commands.

5. [5 points]: Can Ben's modification cause a deadlock situation with two cores? Or does using a two-core machine make his modification safe? Explain your answer.

Answer: Yes, the deadlock can still occur, but not with two processes. With two processes and two CPUs, the CPU that is not looping in acquire with interrupts turned off will eventually take a timer interrupt and schedule the process that acquired the lock before yielding. This allows the lock-holding process to release the lock, which prevents deadlock.

With three processes, deadlock can arise if process 1 makes a system call, acquires the lock, and yields. Then processes 2 and 3 simultaneously make system calls on the two cores, and they both spin

in acquire with interrupts turned off. Again, process 1 will never run again (because all CPUs have interrupts disabled) and never release the lock, resulting in deadlock.

IV JOS Paging (1)

Ben Bitdiddle is finding himself thoroughly confused by page tables and how they work in JOS. He finds that much of his confusion stems from all the bit fiddling that goes on with the `pde_t` and `pte_t` types. Ben decides to rewrite the code to make `pde_t` and `pte_t` (defined in `inc/memlayout.h`) structs with separate `uint32_t` fields for the PDE index and the PTE index, as well as individual fields for the permission bits. He changes all the bit manipulation macros so that they instead modify the appropriate struct fields. Here is Ben's PTE struct definition; his PDE definition is similar:

```
struct pte {
  uint32_t physical_page_number;
  uint32_t available;
  uint32_t dirty;
  uint32_t accessed;
  uint32_t user;
  uint32_t writeable;
  uint32_t present;
};
```

Satisfied with this much more readable code, Ben compiles his kernel and tries to run it. It immediately crashes.

6. [5 points]: Why is it not okay for Ben to modify the `pde_t` and `pte_t` types?

Answer: These types need to match the memory layout the x86 hardware expects for PDEs and PTEs, otherwise the virtual memory translation performed by the MMU will not work correctly.

Ben has taken 6.858 (Computer Security), and finds it suspicious that `pgdir_walk` allocates new PDEs with all permissions set (`PTE_W|PTE_U`). He worries that this would mean that all pages will be writeable by user processes.

7. [5 points]: Why is this not the case?

Answer: The permissions are the intersection of the permissions in the PTE and the PDE, so JOS gets a chance to further limit user permissions when it sets up each PTE.

Ben is having unexpected issues with lab 3. His `load_icode` contains the following code

```
elf = (struct Elf *)binary;
ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
eph = ph + elf->e_phnum;
for (; ph < eph; ph++) {
  if (ph->p_type != ELF_PROG_LOAD) continue;
  region_alloc(e, (void *) ph->p_va, ph->p_memsz);
  memcpy((void *) ph->p_va, binary + ph->p_offset, ph->p_filesz);
  memset((void *) ph->p_va + ph->p_filesz, 0, ph->p_memsz-ph->p_filesz);
}
```

Ben has carefully traced through his code, and finds that a triple fault occurs in `memcpy`, despite the write going to the address that was just allocated with `region_alloc`.

8. [5 points]: Why is Ben's call to `memcpy` not doing what he expects? Assume that his implementation of `region_alloc` is correct.

Answer: `load_icode()` is run by process A to set up process B's memory. Thus it initially runs with process A's page table. `region_alloc()` adds a PTE to B's page table, but in the above code, the `memcpy()` is run with A's memory mapping. It will therefore write to whatever physical page `ph->p_va` points to in A, or fault (as in Ben's case) if that page is not mapped in A. One way to fix this is for the code to use `lcr3()` to load B's page table just before for loop.

V JOS Paging (2)

Ben has a bug in his JOS Lab 3. He boots his kernel and runs a userspace process with the following code at the start:

```
_start:
  800020:  cmpl $USTACKTOP, %esp
  800026:  jne args_exist
  800028:  pushl $0
  80002a:  pushl $0
args_exist:
  80002c:  call libmain
  800031:  jmp 800031
```

Ben gets a page fault in his program, with the trap-frame looking like this:

```
TRAP frame at 0xf01c0000
  edi  0x00000000
  esi  0x00000000
  ebp  0x00000000
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0xeebfdffc
  err  0x00000006 [user, write, not-present]
  eip  0x00800028
  cs   0x----001b
  flag 0x00000046
  esp  0xeebfe000
  ss   0x----0023
```

9. [5 points]:

What is Ben's bug? What evidence supports your answer?

Answer: In this case, the page-fault error code indicates that an attempted write failed when trying to execute the push instruction (which the eip points to). The push instruction writes a value on the stack, which corresponds to the write operation that faults. This suggests that Ben didn't set up the user stack mappings correctly. As further confirmation, the cr2 value (the faulting address) is close to the esp value.

VI Booting

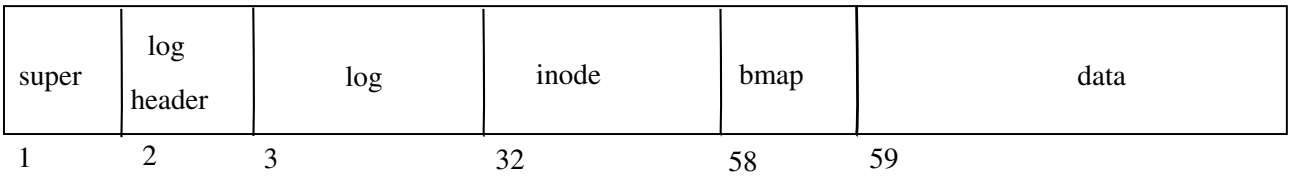
Ben wants to set a break-point inside kern/entry.s because he is unsure if his bootloader is loading and jumping into the kernel correctly. He opens up obj/kern/kernel.asm and sees the address of entry is f010000c, so he sets a breakpoint at 0xf010000c right when GDB starts. However, the breakpoint never hits and Ben is stumped.

10. [5 points]: Why doesn't the breakpoint at 0xf010000c ever trigger?

Answer: The processor executes this instruction before paging is enabled (specifically at address 0x10000c), and uses a physical address, so gdb never observes an %eip equal to f010000c.

VII Xv6 file system

Xv6 lays out the file system on disk as follows:



Block 1 contains the super block. Blocks 2 through 31 contain the log header and the log. Blocks 32 through 57 contain inodes. Block 58 contains the bitmap of free blocks. Blocks 59 through the end of the disk contain data blocks.

Ben modifies the function `bwrite` in `bio.c` to print the block number of each block written. He boots xv6 with a fresh `fs.img` and types in the command `echo > x`. This command creates the file `x` but does not write anything into `x`. This command produces the following trace:

```
$ echo > x
write 3
write 4
write 2
write 34
write 59
write 2
$
```

11. [5 points]: Briefly explain what block 59 contains in the above trace.

Answer: The root directory, with the new entry for "x".

12. [5 points]: Briefly explain what block 4 contains in the above trace.

Answer: A logged copy of block 59.

13. [5 points]: Consider the first 3 writes in the trace. Could the correctness of the xv6 file system be violated if the disk driver sent the writes to the disk hardware in the following order? (Briefly explain your answer)

```
write 4
write 2
write 3
```

Answer: Block 2 contains the log header, whose length field indicates whether the transaction has committed (is complete). When block 2 is written, it indicates to a future post-crash recovery that block 3 should be replayed to block 34, and block 4 should be replayed to block 59. If a crash happens after 4 and 2 are written, but before 3 has a chance to be written, recovery will still replay block 3 to block 59. However, since block 3 wasn't written to the disk before the crash, the replay will copy some garbage (whatever happened to be in block 3 from the previous transaction) to block 34 (which contains i-nodes). This garbage will cause xv6 to crash later on, or yield incorrect results from future file system operations.

End of Quiz I