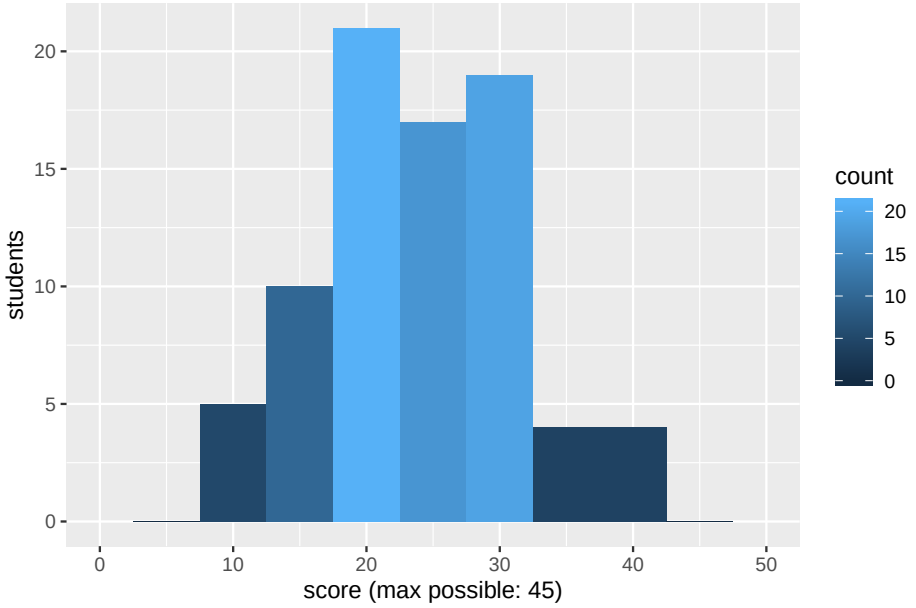
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.S081/6.828 Fall 2019

Quiz I Solutions

Mean 24 Median 23 Standard deviation 7.4



I Buddy allocator

Consider the buddy allocator from the allocation lab. Ben proposes to change the interface from `free(p)` to `free(p, size)` (and similarly for `bd_free`), where `size` is the number of bytes of the block of memory that `p` points to. With this change, the caller of the buddy allocator is responsible for specifying the size of a block when returning the block. Assume that the programmer always passes the correct size (the same size originally requested when allocating).

1. [3 points]: With this change, is it necessary for the buddy allocator to maintain the `split` arrays? (Briefly explain your answer.)

Answer: No. The array `split` is used only to find the size of a block. With the change in interface, the size is supplied to the allocator and so there is not need to compute it. All other uses of `split` are for updating `split` so that `size` returns the correct value. `split` is not used to find out if a buddy is free and to merge with the buddy.

II Page tables

xv6 uses `copyout` to copy data from the kernel to a process's user-space memory. `copyout` translates user-space addresses into kernel addresses. Alyssa suggests that with appropriate mappings in the kernel's page table, `copyout` could be simplified to be as follows:

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  memmove((void *) dstva, src, len);
}
```

She points out that the kernel page table must be updated each time the kernel switches between user programs.

2. [3 points]: How should the kernel page table be set up to make the above `copyout` work correctly for programs smaller than 0x02000000 bytes in size? (Hint: Figure 3-3 of the xv6 text may be helpful.) Briefly explain your answer.

Answer: Modify the kernel page table to map the user program at the bottom of the kernel address space, starting at address 0, with the same mappings as in the user page table. This is possible because the bottom part of the kernel address space below 0x02000000 is unused.

3. [3 points]: How should the kernel page table be set up to make the above `copyout` work correctly for programs greater than 0x02000000 but smaller than 0x80000000 (`KERNBASE`) bytes in size? Briefly explain your answer.

Answer: The kernel page between 0x02000000 and 0x80000000 maps devices. One solution is to modify the kernel to move those devices to unused addresses above 2Gbyte in the kernel address space, instead of using a direct mapping. Then, there is enough space to map the user program as in the previous answer.

III Interrupts

4. [3 points]: Can an executing kernel thread be interrupted by a timer interrupt that causes it to yield its CPU to another kernel thread? (Briefly explain your answer using the relevant xv6 source code.)

Answer: Yes. `usertrap` enables interrupts before calling `syscall` and `kerneltrap` yields its CPU on a timer interrupt.

5. [3 points]: A CPU is handling a disk interrupt and is executing `virtio_disk_intr` just before the call to `acquire`. If a character arrives at the UART, could that CPU interrupt execution of `virtio_disk_intr` and cause xv6 to execute `uartintr`? Briefly explain your answer. (Note: you don't need to understand the details of the disk driver to answer this question.)

Answer: No. The RISC-V hardware turns off the `SIE` bit in `sstatus` when taking an interrupt. xv6 doesn't set the bit when processing an interrupt.

IV System Call Arguments

Alyssa is modifying xv6 to work with user programs written in the new programming language Stop, which passes function arguments on the stack instead of in registers. When a function starts, the `sp` register points to the first argument, the second argument is at address `sp+8`, the third is at `sp+16`, etc.

System call implementations in the xv6 kernel need to use the new convention to read arguments. For example, Alyssa needs to modify `argaddr` in `kernel/syscall.c` to get the argument from the user stack rather than a register.

6. [3 points]: Write kernel code that will fetch the first 64-bit system call argument and put it in the `uint64` variable `x`. Your answer should be one or two lines of code.

Answer: `copyin(p->pagetable, &x, p->tf->sp, 8)` or `fetchaddr(p->tf->sp, &x);`

V set_satp

Many applications could benefit from being able to control their own address spaces, by specifying how user virtual addresses map to the physical memory that the application owns. For example, an application might want to place its stack at a high virtual address, to allow the stack to grow downwards, and the heap to grow upwards. Or an application might want to map some of its data read-only, in order to catch bugs that accidentally try to write that data.

However, the RISC-V hardware prohibits user-mode software from changing the `satp` register, which means that an application cannot directly control its own address space.

7. [3 points]: Explain why it would be a bad idea if RISC-V microprocessors allowed user-mode software to change `satp`.

Answer: That would allow user code to set up mappings to allow it to directly read/write any process's or kernel's memory, breaking isolation.

Ben Bitdiddle is extending xv6 so that it can be used as the operating system on the Athena time-sharing and dial-up machines, in place of Linux. One of Ben's ideas for xv6 is to add a new `set_satp` system call. The call would take one argument, a pointer to a page table. The process would prepare a page table in its memory, and pass the address of the page table to the new system call. In the kernel, the `set_satp` implementation would check that all the PTEs in the page table either refer to physical pages that the process owns, or have `PTE_V` clear. If the page table supplied by the process passes these checks, `set_satp` would put the page table's physical address in `p->pagetable`, so that the trampoline code would install it in `satp` when returning to user space. Ben has in mind that there should also be new system calls that allow a process to find out the physical addresses of the memory it owns.

8. [3 points]: Explain why Ben's `set_satp` idea is a disaster for security, even though it checks that every valid PTE refers to the calling process's own memory.

Answer: A process could change a PTE after calling `set_satp`, making the PTE refer to memory in another process or the kernel. The RISC-V MMU looks at the current content of the page table when translating virtual to physical addresses, not just the contents as of when `satp` was installed.

VI Pre-Emptive Context Switching

Xv6 implements pre-emptive switching in the following way. When a timer interrupt occurs while user code is executing, this code towards the end of `usertrap()` gives up the CPU:

```
// give up the CPU if this is a timer interrupt.
if(which_dev == 2)
  yield();
```

`yield()` switches to the current CPU's scheduler thread *via* `sched()` and `swtch()`, and the scheduler thread picks a `RUNNABLE` process to run and switches to it.

Ben Bitdiddle aims to make pre-emptive switching faster by having `usertrap()` directly return into user-space in a different process, rather than first switching through the scheduler. After all, if `usertrap()` didn't call `yield()`, it would return back to the current process in user space; why not have it return into a different process's user space? Ben replaces the `yield()` in `usertrap()` with this code:

```
if(which_dev == 2){
  extern struct proc proc[];
  struct proc *np;
  struct cpu *c = mycpu();

  for(np = proc; np < &proc[NPROC]; np++) {
    if(np == p)
      continue;
    acquire(&np->lock);
    if(np->state == RUNNABLE) {
      // switch from p to np.
      np->state = RUNNING;
      c->proc = np;         // make myproc() return np.
      p->state = RUNNABLE; // p isn't running any more.
      release(&np->lock);
      break;
    }
    release(&np->lock);
  }
}
```

Ben boots his modified xv6 and runs some commands; for a while it works. However, eventually xv6 panics. Ben tries again a few times, but while the details of the failure differ from run to run, he always gets a panic or kernel page fault. Assume that the code above is a faithful implementation of Ben's idea.

9. [3 points]: Explain what is wrong with Ben's idea of changing the `yield()` into a direct return to user-space in the new process.

Answer: Ben's scheme only works if `np` (the RUNNABLE process) entered the kernel with a timer interrupt and called `yield()`. If, on the other hand, `np` is `RUNNABLE` because it has just been woken up from a `sleep` in the middle of a system call, then Ben's scheme incorrectly skips whatever kernel code in the system call comes after the sleep. Another problem is that Ben's code doesn't save registers in `p->context`; this not an immediate problem, but if `p` is resumed with `swtch` in the future, its kernel thread will run with garbage in its registers. Another problem is that Ben's code sets `p->state` to `RUNNABLE` without holding `p->lock`, so another CPU could start running the process (on its kernel stack) before `usertrap()` has finished.

VII Wakeup

Here's a simplified version of `consoleintr` from `kernel/console.c`. This listing omits much of the function, but you should assume that the version shown here works correctly. Assume that `c`, the character just typed, is a newline; and that there is space for another character in `cons.buf`.

```
void
consoleintr(int c)
{
  // AAA

  acquire(&cons.lock);

  // BBB

  // store for consumption by consoleread().
  cons.buf[cons.e++ % INPUT_BUF] = c;

  wakeup(&cons.r);

  release(&cons.lock);
}
```

10. [3 points]: If the `wakeup()` call were moved to the point marked `AAA`, would xv6 console input still operate correctly? Explain your answer.

Answer: No. The `wakeup()` might wake up a process sleeping in `consoleread()`; that process might execute on a different CPU, acquire `cons.lock`, see that no characters were buffered, and go back to sleep. Then `consoleintr()` might proceed to acquire the lock and add the character to `cons.buf`. The waiting process would never be woken up and might never see the waiting input. This is a lost wakeup.

11. [3 points]: If the `wakeup()` call were moved to the point marked `BBB`, would xv6 console input still operate correctly? Explain your answer.

Answer: Yes. The fact that `consoleintr()` holds the lock at this point means that a concurrent `consoleread()` has to wait for the lock, and thus won't run until the character is in `cons.buf`.

VIII Lab Lazy

For her lazy page allocation lab, Alyssa writes the following kernel code to help handle a page fault caused by a process using a lazily-allocated page.

```
int
demand_page(uint64 stval, struct proc *p)
{
  uint64 va = PGROUNDDOWN(stval);
  void *mem;

  if(va >= MAXVA){
    return -1;
  }

  mem = kalloc();
  if(mem == 0){
    return -1;
  }

  memset(mem, 0, PGSIZE);
  if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, PTE_W|PTE_X|PTE_R|PTE_U) != 0){
    kfree(mem);
    return -1;
  }

  return 0;
}
```

In usertrap(), she inserts the lines marked with +:

```
    } else if((which_dev = devintr()) != 0){
      // ok
+   } else if(r_scause() == 13 || r_scause() == 15) {
+       if (demand_page(r_stval(), p) != 0) {
+         p->killed = 1;
+       }
    } else {
```

Unfortunately, while testing her code, Alyssa's kernel crashes with "panic: freewalk: leaf". The changes she made to `sys_sbrk` are correct and she made no other changes than shown above.

12. [3 points]: What bug in Alyssa's code is causing this panic? How should she fix it?

Answer: Alyssa has forgotten to check that `stval` (or perhaps `va`) is less than `p->sz`. A user access to an address that is too large will result in a page being allocated, but it won't be freed and removed from the pagemap when the process exits, which triggers the panic.

After fixing the above problem, Alyssa is overjoyed to discover that `lazytests` now passes, as well as many of the user tests. However, one user test still causes a crash, printing "panic: remap".

13. [3 points]: Why might this panic occur, and what else does Alyssa need to do fix her code?

Answer: Usertests attempts to write to the "guard" page that's just beneath the stack. The guard page has `PTE_V` set, but not `PTE_U`. Alyssa's code doesn't check for this situation, so her code tries to install a new page into that PTE, which triggers the panic.

IX Lab COW

Ben has implemented a working solution to the copy-on-write lab. As part of his solution, he has an array of reference counts as part of the `kmem` struct, `kmem.refcnt`. He has written a `kdecref` function to decrease the count, and he has modified `kfree` to call `kdecref` and free the physical page only if the new reference count is 0 (the modification is emphasized below). `kfree` is the only function that calls `kdecref`. The C operation `--x` decrements `x` and yields the new value.

```
int
kdecref(void *pa)
{
  int i = PA2INDEX(pa); // index into ref count array
  return --kmem.refcnt[i]; // decrement and return new ref count
}

void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  //BEGIN NEW CODE
  acquire(&kmem.lock);

  int ref = kdecref(pa);
  release(&kmem.lock);
  if (ref > 0) {
    return;
  }
  // END NEW CODE

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

14. [3 points]: Ben wonders if the `acquire(&kmem.lock)` and `release(&kmem.lock)` surrounding the call to `kdecref` are necessary, or if they just add unnecessary overhead. Would it be okay to elide that `acquire`/`release`? Briefly explain why or why not. If not, describe a concrete sequence of events that leads to incorrect behavior if the locking is removed.

Answer: Without the lock, two calls to `kfree` for the same page on different CPUs would race in `kdecref`. Suppose the page started with reference count two. Both calls might read two, subtract one, and write one to the reference count. But the correct final value is zero. The result is that the page won't be freed, even though it should be. The page will no longer be useable; if this happens enough, the system will eventually run out of memory.

X 6.S081/6.828

15. [1 points]: Which parts of the xv6 textbook were most helpful? Which parts did you think were most confusing?

Answer: Helpful: Diagrams x 20, Walkthroughs x5, VM x1, Scheduling x4, Traps x3. **Confusing:** Traps x8, Locking x5, Code explanations x5.

16. [1 points]: Please indicate which of the labs you found to be the most helpful, and which the least.

- Unix utilities **Answer: 10 helpful / 8 unhelpful**
- Simple shell **Answer: 13 helpful / 12 unhelpful**
- Memory allocator **Answer: 11 helpful / 25 unhelpful**
- Lazy page allocation **Answer: 30 helpful / 6 unhelpful**
- Copy-on-write fork **Answer: 34 helpful / 4 unhelpful**
- Uthread and alarm **Answer: 21 helpful / 3 unhelpful**

17. [1 points]: What's the most important thing we could fix about 6.S081/6.828 to make it better?

Answer: More office hours x9, More project guidance x7, Fewer/easier labs x5, Video lectures x5, More GDB x4, Better lecture notes x4, Lab solutions x2, More late days x2

End of Quiz I