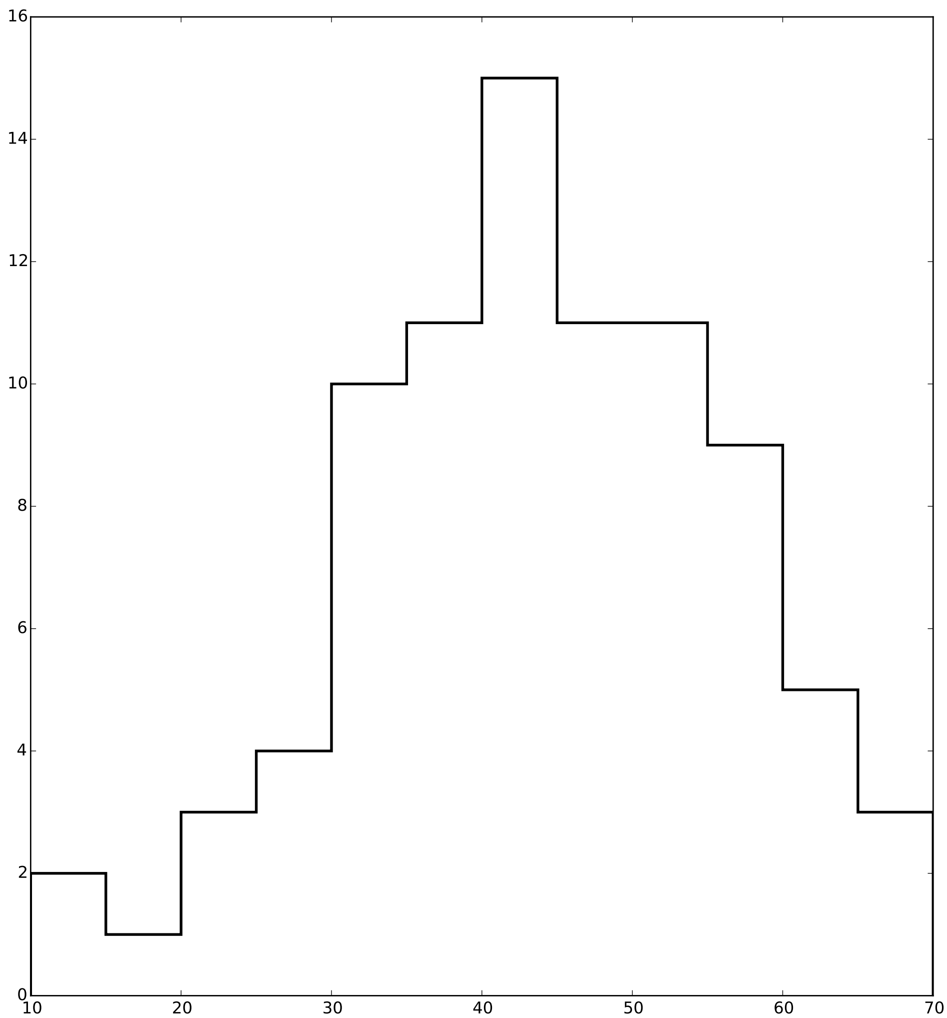
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2014

Quiz I Solutions

Mean 42.8 Median 43.0 Standard deviation 12.1 Kurtosis 2.7



I Shell

A solution in the xv6 shell for setting up a pipe for filters commands such as `ls| wc` is as follows:

```
case '|':
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0) {
    perror("pipe");
    exit(-1);
  }
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->left);    // run left side of pipe and exit
  }
  if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);  // run right side of pipe and exit
  }
  close(p[0]);
  close(p[1]);
  wait(&r);
  wait(&r);
  break;
```

1. [5 points]: Why does the child process that runs the left-side of the pipe close file descriptor 1 and why does the child process that runs the right-side of the pipe close file descriptor 0?

Answer: dup must replace stdout for the left-end of the pipe, because that child's output goes into the pipe. dup must replace stdin for the right-end of the pipe, because that child gets input from the pipe.

2. [5 points]: It is possible that the child process that runs the left-end of the pipe finishes before the child process that runs the right-end of the pipe is created. Why is the above implementation still correct?

Answer: It can only happen if the left-end generates less than a pipe-buffer full of data (since if it generated more, the write into the pipe would block). In this case, the OS will internally buffer the data and deliver it to the first read call. The pipe will not be destroyed prematurely because the parent does not close its file descriptors for the pipe until after both children have been created.

3. [5 points]: Why is it important that the child that runs the left-end closes `p[1]`?

Answer: The child that runs the right-end of the pipe should receive end-of-file (instead of blocking forever) if the left-end is done writing and exits.

II Upcalls

Ben explores the following implementation of the `alarm` system call in xv6:

```
int sys_alarm(void)
{
  int ticks;
  void (*handler)();

  if(argint(0, &ticks) < 0)
    return -1;
  if(argptr(1, (char**)&handler, 1) < 0)
    return -1;
  proc->alarmticks = ticks;
  proc->alarmhandler = handler;
  return 0;
}
```

He modifies `trap` as follows:

```
    if(proc && (tf->cs & 3) == 3 && proc->alarmhandler) {
      proc->ticks++;
      if (proc->ticks >= proc->alarmticks) {
        proc->ticks = 0;
        (*proc->alarmhandler)();
      }
    }
```

He uses the same code for registering an alarm handler as given in the alarm homework. He tests this solution with `alarmtest`:

```
void periodic();

int
main(int argc, char *argv[])
{
  int i;
  printf(1, "starting␣alarmtest\n");
  alarm(10, periodic);
  for(i = 0; i < 50*500000; i++){
    if((i++ % 500000) == 0)
      write(2, ".", 1);
  }
  exit();
}

void
periodic()
{
  printf(1, "alarm!\n");
}
```

When running `alarmtest` Ben observes that the kernel invokes `alarmhandler` but he doesn't observe the output of `alarmtest`'s print statement (i.e., "alarm!") that a correct solution would print. He also observes that the kernel doesn't crash and `alarmtest` terminates. In fact, he can run `alarmtest` several times.

4. [5 points]: Alice points out that Ben's solution is very broken, even though it doesn't crash the kernel and seems to work mostly, modulo the missing output "alarm!". Explain briefly what is wrong with Ben's solution?

Answer: User code (`alarmhandler`) runs with kernel privileges. The `alarmhandler` could invoke privileged instructions, which user processes shouldn't be able to do.

5. [5 points]: Explain briefly why Ben's solution works so well? That is why doesn't Ben observe a kernel crash, panic, or something else bad? Why can the kernel invoke `periodic`?

Answer: The user code is mapped in the bottom half of the address space and the kernel in the top half. The kernel can invoke code that lies in the bottom half, because the user page directory is still active when running in kernel mode. The system call in `periodic` works, because xv6 allows the kernel to be interrupted.

6. [5 points]: Why does the `printf` in `periodic` print nothing?

Answer: The x86 will not push `esp` to the stack, because a change in privilege level did not occur. Thus the kernel sees a garbage value when it looks at `proc->tf->esp`. This will fail the bounds check in `fetchint`.
A second answer was also accepted: if you did not notice that `proc->tf->esp` is garbage, and assumed that the inner trap frame was correct, then `proc->tf->esp` would have pointed to the kernel stack. This would also fail the bounds check in `fetchint`.

III Using virtual memory

In xv6, each process maps the kernel into its address space using `setupkvm`:

```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP␣too␣high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

Alyssa wants to modify `setupkvm` so that user processes share the x86 page tables for kernel mappings (i.e., the kernel part of the address space) and still work correctly. She modifies `setupkvm` as follows (changes are marked with "New code"):

```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (kpgdir) {                    // New code (note that kpgdir is set by kvmalloc)
    return copykpgdir(pgdir);      // New code
  }                                // New code
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP␣too␣high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

7. [5 points]: Complete the function `copykpgdir`:

```
copykpgdir(pde_t *pgdir)
{






   return pgdir;
}
```

Answer:

```
copykpgdir(pde_t *pgdir)
{
  int i;
  for(i = PDX(KERNBASE); i < NPDENTRIES; i++){
    pgdir[i] = kpgdir[i];
  }
  return pgdir;
}
```

IV The dreaded UVPT

Ben Bitdiddle made a terrible mistake in his solution to lab 2, and has accidentally set the `PTE_W` and `PTE_U` bits in every single page directory entry and page table entry. In a futile attempt at debugging, he writes the following user program:

```
void
umain(int argc, char **argv)
{
    // Pointer to the page directory entry for UVPT
    uint32_t *addr = (uint32_t *) 0x????????;

    // Physical address of the page directory
    uint32_t x = PTE_ADDR(*addr);
}
```

Ben chooses `addr` such that it points to the page directory entry for UVPT (regardless of the layout of physical memory). Therefore `x` contains the physical address of the user program's page directory.

Recall that the UVPT region begins at virtual address `0xef400000`.

8. [5 points]: What is the value of `addr`?

Answer: `0xef7bdef4 = UVPT + (UVPT >> 10) + (UVPT >> 20)`. The page directory page serves as the page directory, page table, and physical page.

Upon further reflection, Ben realizes that the page directory entry for UVPT should not have the `PTE_W` bit set. He attempts to clear it with the following code:

```
void
umain(int argc, char **argv)
{
    // Pointer to the page directory entry for UVPT
    uint32_t *addr = (uint32_t *) 0x????????;

    // Physical address of the page directory
    uint32_t x = PTE_ADDR(*addr);

    // Clear all of the permission bits
    *addr = x;

    // Set permissions to read only, user
    *addr |= PTE_P | PTE_U;
}
```

Ben is very confused to find that the last line of code triggers a page fault, even though the second to last line works fine.

9. [5 points]: Why does the page fault occur?

Answer: When we set `*addr = x`, we immediately clear the `PTE_P` bit of the page directory entry for UVPT. On the next line, we attempt to write to a virtual address which is no longer marked as present!

Alyssa P. Hacker points out that the page fault can be avoided by using a different virtual address to access the page directory entry for UVPT. She mumbles something about the physical memory map at virtual address `0xf0000000`, and makes the following addition to Ben's code:

```
void
umain(int argc, char **argv)
{
    // Pointer to the page directory entry for UVPT
    uint32_t *addr = (uint32_t *) 0x????????;

    // Physical address of the page directory
    uint32_t x = PTE_ADDR(*addr);

    // Avoid potential page fault
    uint32_t offset = 0x????????;
    addr = (uint32_t *) (x + offset);

    // Clear all of the permission bits
    *addr = x;

    // Set permissions to read only, user
    *addr |= PTE_P | PTE_U;
}
```

10. [5 points]: For what value of `offset` will this code work as intended?

Answer: The page directory is located at physical address `x`, so the page directory entry for UVPT is located at physical address `x + 0xef4`. To convert this physical address to a virtual address, we just add `0xf0000000`. Therefore the final offset is `0xf0000ef4`.

V The stack and calling conventions

The calling convention determines how arguments are passed, which registers get saved, and how values are returned when a function is called. JOS uses the standard C calling convention, which is called *cdecl*. cdecl defines eax, ecx, edx as caller saved and all other registers as callee saved.

Consider the function below, which uses cdecl. (As usual, we use AT&T assembly syntax.)

```
prologue:
0:      push    %ebp
4:      mov     %esp,%ebp
8:      push    %ebx
C:      sub     $0x04,%esp
body:
10:     lea     -0x08(%ebp),%ebx
14:     push    %ebx
18:     call    3c
1C:     mov     (%ebx),%eax
epilogue:
20:     add     $0x8,%esp
...
```

11. [5 points]: This listing is incomplete. The prologue and body are provided, but some of the epilogue is missing. Complete the epilogue so that it restores the stack to its initial state and then returns.

Answer:

```
24:     pop     ebx
28:     mov     ebp,esp
2C:     pop     ebp
30:     ret
```

or

```
24:     pop     ebx
28:     pop     ebp
2C:     ret
```

VI Kill, sleep, and file systems

Ben observes that in many sleep loops in xv6 test for `proc->killed`, except in a few places. For example, the sleep loop in `iderw` doesn't check for if a process has been killed while sleeping. Ben modifies the loop in `iderw` to check for `proc->killed` and to return out of `iderw` if set (the changes are marked by comments):

```
void
iderw(struct buf *b)
{
  struct buf **pp;
  int isread = !(b->flags & B_VALID);

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  {
    ;
  }
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    if (proc->killed) {      // Code added
      goto done;             // Code added
    }                        // Code added
    sleep(b, &idelock);
  }

done:
  release(&idelock);
}
```

12. [5 points]: Is this change safe (i.e., will xv6 continue to function correctly)? (Briefly explain if so or if not so.)

Answer: No. Ben is wrong, because `bread` may receive a buffer without the `B_VALID` bit set. A process that calls `bread`, which invokes `iderw`, may be killed before the disk interrupt is delivered, and thus return out of `iderw` without a valid buffer. Since such a buffer could contain arbitrary data, functions such as `ialloc` may corrupt the file system or panic because the data indicates that there are no free inodes (even though there may be free inodes on disk).
Interestingly, there is no problem for `bwrite`, which is used only by the logging system. All writes are performed inside of a transaction, executed in order, and all will complete or none of them. The file system won't get into an inconsistent state.

VII Processor exceptions

Ben Bitdiddle wants to optimize JOS performance. He determines that system call overhead (generating a processor exception via **`int`** `$T_SYSCALL`, executing the system call stub in `IDT[T_SYSCALL]`, executing `trap()`, and finally dispatching the corresponding system call) is unacceptable. He plans to avoid it by loading the system call service code directly into user environments, enabling the user environment to call the system call service code (e.g., `sys_cputs`) directly without any interaction between the environment and the kernel.

13. [5 points]: What is one major downfall of Ben's design?

Answer: User environments must be given access to all kernel data, thus there is no isolation between user environments.

14. [5 points]: x86 processors will automatically switch stacks when a user environment invokes, directly or indirectly, an exception handler with a higher privilege level. Suppose the x86 didn't switch stacks when handling an exception generated while executing a user environment – what specifically could go wrong?

Answer: The state of the stack is arbitrary, thus the processor may fail to push important things like the IP, CS, SS, etc.

VIII 6.828

We'd like to hear your opinions about 6.828, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

15. [2 points]: This year we posted an improved draft of the xv6 commentary at the beginning of the semester. Did you find the chapters useful? What should we do to improve them?

Answer: Most students who read the xv6 book found it useful. Those who didn't know about it agreed that such a thing would have been very useful.

16. [2 points]: Did you find code reviews useful? What should we do to improve them?

Answer: Code reviews should be assigned sooner, and guidelines regarding the content of the reviews should be provided. Some students still have not received one or both of their code reviews for lab 2.

17. [1 points]: We slowed the pace of the 6.828 labs a bit through a combination of hacking days, spreading the labs, and having more lectures directly focused on the labs. How was the pace this year? Too slow? Still too fast? About right?

Answer: Most students thought the pace was about right, with equal numbers saying "too fast" and "too slow".

End of Quiz I