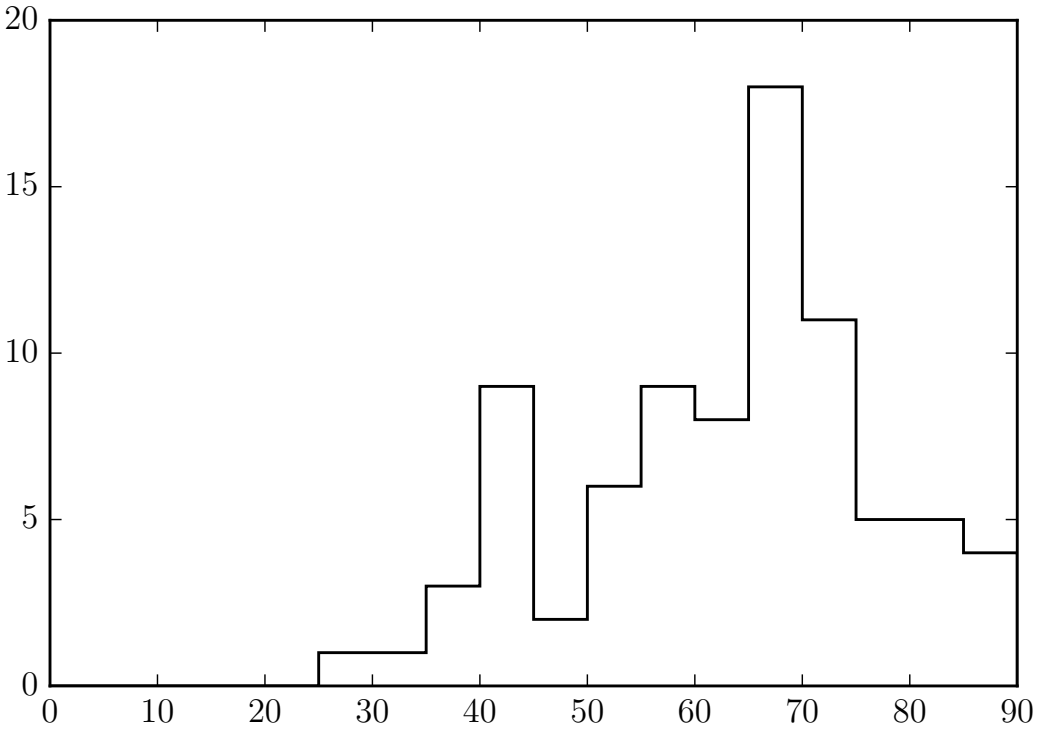
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2014

Quiz II Solutions

Mean 62.1 Median 65.0 Standard deviation 14.0 Kurtosis 2.6



I Ext3

1. [5 points]: Ext3 supports several modes, including journaled and ordered. Consider the workload of an application overwriting the same file n times: each iteration opens the file, writes all the blocks in the file, closes the file, and syncs the file. The file is b blocks large. For this workload, estimate how many disk writes of data blocks Ext3 will perform in journaled mode? How many in ordered mode? (Explain your answer briefly.)

Answer: In journaled mode, ext3 will write each block twice (once to the log and once to its actual location), so the number of data writes is $2bn$. In ordered mode, each block is only written once and the total number of writes is bn.

2. [5 points]: Describe an application that would want to use ext3 in journaled mode, and explain briefly what would go wrong if the application were to use ordered mode.

Answer: An application that requires failure-atomicity for its data blocks would want to use journaled mode. For example, consider a database which uses a file to implement its transaction log. In ordered mode, the metadata of the log might be updated but the data blocks could be missing after a sync concurrent with a power failure.

II OS organization

This question explores the differences between 3 OS organizations: monolithic, microkernel, and exokernel.

3. [5 points]: Identify a subsystem out of JOS that provides a good example of an exokernel organization and briefly explain why it is a good example.

Answer: User-level copy-on-write fork is a good example of exokernel organization because it uses the user-space library operating system to provide system-level functionality.

4. [5 points]: Identify a subsystem out of JOS that provides a good example of a microkernel organization and briefly explain why it is a good example.

Answer: The file server is a good example of microkernel organization because it is a user-space environment, the disk is mapped into only that environment, and other environments interact with it using IPC.

5. [5 points]: Identify a subsystem out of JOS that provides a good example of a monolithic-style subsystem and briefly explain why it is a good example.

Answer: The network device driver is a good example of monolithic organization because it is implemented entirely in the kernel.

III Singularity

In Singularity only one process can have a pointer to an object in the exchange heap. Ben proposes to modify Singularity to maintain a reference count for each object in the exchange heap and collect the object when the reference count is zero. When a process dies, he arranges to decrease the reference counter for each object in the exchange heap to which the dying process has a pointer.

In Ben's design several processes can share an object in the heap. This would allow Ben to use the exchange heap, for example, for a shared buffer cache.

6. [5 points]: What problem does Singularity try to solve by disallowing shared objects in the exchange heap? (Briefly explain your answer.)

Answer: Disallowing shared objects provides stronger fault isolation. If shared objects were allowed, one process could scribble over a block in the buffer cache after it had sent the block to another process via IPC.

7. [5 points]: How would you implement a shared buffer cache in Singularity as described in the paper (i.e., without Ben's modification)? (Briefly explain your answer.)

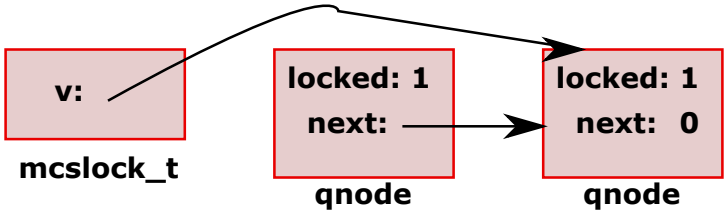
Answer: You would write a user-level file server that maintains the buffer cache internally. Since objects cannot be shared, the file server would need to copy a buffer before sending it to a consuming process.

IV Scalable locks

The following code is an implementation of MCS locks, which was handed out in lecture:

```
1  static inline void
2  mcs_init(mcslock_t *l)
3  {
4          l->v = NULL;
5  }
6
7  static inline void
8  mcs_lock(mcslock_t *l, volatile struct qnode *mynode)
9  {
10         struct qnode *predecessor;
11
12         mynode->next = NULL;
13         predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
14
15         if (predecessor) {
16                 mynode->locked = 1;
17                 asm volatile("":::"memory")    // memory barrier
18                 predecessor->next = mynode;
19                 while (mynode->locked)
20                         nop_pause();
21         }
22 }
23
24 static inline void
25 mcs_unlock(mcslock_t *l, volatile struct qnode *mynode)
26 {
27         if (!mynode->next) {
28                 if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
29                         return;
30                 while (!mynode->next)
31                         nop_pause();
32         }
33         ((struct qnode *)mynode->next)->locked = 0;
34 }
```

This code creates the following structure when two cores are waiting for a given lock:



8. [5 points]: Using the picture, explain how MCS locks avoid collapse when a lock is contended by many cores.

Answer: Collapse is avoided because the unlocking node writes to `mynode->next->locked`, which is only being read by a single CPU. Thus only one CPU issues a request for the recently invalidated cache line, and the cost of releasing the lock is independent of the number of CPUs waiting for the lock.

The instruction `asm volatile("":::"memory")` (line 17) is a memory barrier, which tells the compiler to not reorder memory accesses past this barrier.

9. [5 points]: Describe a race if this instruction was omitted and the compiler was allowed to reorder instructions past the memory barrier.

Answer: The race happens when the instruction on line 16 is ordered past line 18. In this case, the acquirer is inserted into the list while `locked` remains unset. If the lock holder calls `mcs_unlock` before the acquirer executes the reordered line 16, the releaser will set `locked` to 0, and then the acquirer will set `locked` to 1 and spin on it forever.

V Virtual machines

10. [5 points]: Extended page tables translate guest physical to host physical addresses. Suppose we have two kernels (K_1 and K_2), which each map guest virtual addresses 0 to n to guest physical addresses 0 to n, where n is 4 Mbyte. What mapping should a virtual machine monitor setup using extended page tables for K_1 and K_2 so that the monitor can multiplex and run both kernels?

Answer: One solution is the following: the EPT for K_1 is the identity map, and the EPT for K_2 adds n to each guest physical address.

11. [5 points]: In IX, when a network application calls `run_io` how many protection boundary changes (e.g., ring 3 to ring 0, non-root to root) happen? (Briefly explain your answer.)

Answer: There are two protection boundary changes: from ring 3 non-root to ring 0 non-root when `io_run` is called, and back to ring 3 non-root when it returns.

12. [5 points]: In IX, when IX reads or writes the NIC queue, how many protection boundary changes happen? (Briefly explain your answer.)

Answer: There are no protection boundary changes, because the NIC queues are memory-mapped into IX's address space.

VI Multitasking and Locks in JOS

Alyssa P Hacker has successfully implemented fine-grained locks for her lab 4 challenge in 6.828. The page allocator, the console driver, the scheduler, and the inter-process communication (IPC) code each have their own spin lock.

Later, when testing her fork code with 4 CPUs, Alyssa writes the following code fragment:

```
if((a = fork()) == 0) {
    ipc_recv(&a, 0, 0);
    cprintf("Process␣One!\n");
    exit();
}
if((b = fork()) == 0) {
    ipc_recv(&b, 0, 0);
    cprintf("Process␣Two!\n");
    exit();
}
ipc_send(a, 0, 0, 0);
ipc_send(b, 0, 0, 0);
cprintf("Process␣Three!\n");
```

13. [5 points]: Assume that there are no other environments running. In what order will the strings be printed? (Explain briefly why.)

Answer: There is a race condition: process one and process two will likely be picked up by the spare CPU cores when they are awakened (at the same time) by a timer interrupt. We can't say anything about the order.

VII Disk block cache

Consider the following block cache code in JOS:

```
1  #define PTE_SYSCALL     (PTE_P | PTE_W | PTE_U)
2
3  // Write the cached block at virtual address addr back to disk
4  void
5  flush_block(void *addr)
6  {
7          uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
8
9          if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
10                 panic("flush_block␣of␣bad␣va␣%08x", addr);
11
12         if (!va_is_mapped(addr) || !va_is_dirty(addr))
13                 return;
14
15         // Write the disk block and clear PTE_D.
16         addr = ROUNDDOWN(addr, BLKSIZE);
17         ide_write(blockno * BLKSECTS, (void*) addr, BLKSECTS);
18         sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL);
19 }
20
21 // Sync the entire file system.  A big hammer.
22 void
23 fs_sync(void)
24 {
25         int i;
26         for (i = 1; i < super->s_nblocks; i++)
27                 flush_block(diskaddr(i));
28 }
29
30 static void
31 bc_pgfault(struct UTrapframe *utf)
32 {
33         <Sanity checks on faulting address and the super block...>
34         <Allocate new page...>
35
36         if ((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
37                 panic("in␣bc_pgfault,␣ide_read:␣%e", r);
38
39         if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
40                 panic("in␣bc_pgfault,␣sys_page_map:␣%e", r);
41 }
```

Recall that the x86 automatically sets the “dirty bit” in the page table entry of any page that is written by software.

Ben Bitdiddle is working on the block cache for lab 5, but doesn't know that his implementation of `sys_page_map()` is buggy. When Ben's `sys_page_map()` maps a page, the page table entry is not modified if the physical address of the mapped page is unchanged.

14. [5 points]: For testing purposes, Ben writes an program that simply reads every file in the file system. Ben notices that, after running his test program, a call to `fs_sync()` takes several seconds to complete even though no data was written. Why does `fs_sync()` take so long?

Answer: Because of the buggy `sys_page_map()`, the dirty bit remains set on many file mappings and `fs_sync()` writes all of them back to disk.

VIII File descriptors in JOS

Ben Bitdiddle would like to implement a new function, called `reset_fds()`, which will close all of a JOS environment's open file descriptors.

15. [5 points]: How should this function be implemented in JOS? What functionality should be added to the kernel (e.g. via a system call) and what modifications should be made to user space?

Answer: No modifications should be made to the kernel, because an exokernel doesn't know anything about file descriptors. The new function `reset_fds()` should be added to the library operating system, probably in `lib/fd.c`.

Ben's implementation of `reset_fds()` contains the following line of code:

```
memset(FDTABLE, 0, MAXFD*PGSIZE);
```

(Recall that the arguments to `memset` are the start address, fill value, and length to be filled.)

Ben writes a program to test `reset_fds()` and runs it from the shell. Unfortunately, two bad things happen when the test program calls `reset_fds()`: the program page faults, and console input and output stop working in the shell.

16. [5 points]: Why does the test program page fault?

Answer: Not every FD is in use, and if FD `n` is not in use then no memory is mapped at virtual address `FDTABLE+n*PGSIZE`.

17. [5 points]: Why do console input and output stop working in the shell?

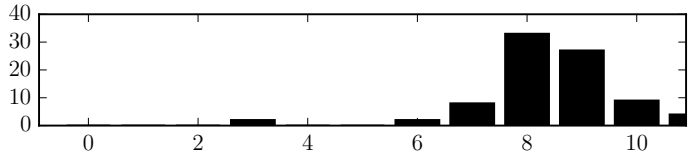
Answer: The pages representing open file descriptors are zeroed out. Because these pages are shared (with `PTE_SHARE`), the **`struct`** `Fd`s for standard input and output are zeroed out for every user environment launched by init. The correct solution would have been to use `sys_page_unmap` or `fd_close` instead.

IX 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.)

18. [1 points]: Grade 6.828 on a scale of 0 (worst) to 10 (best)?

Answer:



19. [2 points]: Any suggestions for how to improve 6.828?

Answer: Issue code reviews more promptly. Include more papers. Include fewer papers. Implement VGA. Better classroom. More lectures. Final project topic selection meetings. Discussion of lab solutions. Better online notes. The final is too early in the morning. More tests. More lecture time on labs. Homework could be more meaningful. Encourage more students to do lab 7. Instructions on what parts of the papers are important. Better communication about test time/format. More time on gdb, tools, etc. Less time on xv6. More portable toolset. Post hw solutions. More design in the labs. Group meetings to discuss labs. Provide less code. Hold office hours closer to lab deadline. Relate papers more to other things, and each other. Add recitations. Move final into in-class exam. Only 1 quiz. Make it harder. Fewer papers, but in more-depth discussion. Add diagrams to lecture notes. Lower the pace. More time with staff. More Linux. Microquizzes. Lecture videos. Split labs and give more deadlines. Less stepping through code.

20. [1 points]: What is the best aspect of 6.828?

Answer: Labs. Writing an operating system.

21. [1 points]: What is the worst aspect of 6.828?

Answer: Debugging. Quizzes.

End of Quiz II — Enjoy the break!