1. In RAID:

a. Mirroring is generally better-performing than striping
b. RAID-4 can survive up to 4 disk failures without losing data
c. For random writes, RAID-0 and RAID-1 are typically better performing than RAID4 or RAID5
d. In RAID-4, if an application decided to write a block, the operating system must always first read at least one other block.
e. RAID-5 dominates RAID-4 so much more that RAID-4 is almost never used these days

C

a. Striping is faster.
b. RAID-4 has nothing to do with disk failures.
c. Correct.
d. Blocks can be cached. Also, overwriting of data with same copy.
e. Not true, RAID5 has more overhead to add a disk drive.

2. Some security questions

a) If prime numbers were quite rare, algorithms like RSA that are based on factoring products of primes would be insecure
b) Traditionally in Unix, the password file /etc/passwd was world-readable, so anybody could read anybody else's password and the system was insecure
c) A larger trusted computing base helps to increase the security of a system
d) Shared-secret cryptography is rarely used nowadays; almost all encryption now uses public keys
e) Principals in GNU/Linux are uid_t values, which are integers

A

a. If they were rare, it would be easy to make and store a list of primes. Correct.
b. Wrong.
c. Wrong. More room for buggy software and loopholes.
d. Wrong. Shared key is faster. Public key used mostly for bootstrapping the process.
e. They are not integers.

3. In NFS

a. Each call to 'open' without a corresponding 'close' corresponds to a distinct NFS file handle.
b. An NFS file handle's generation number helps avoid race conditions when a parent and child process close a shared file descriptor at about the same time
c. The NFS model assumes a smart file server and a dumb client, rather than the reverse
d. In practice, most NFS operations are idempotent; this keeps the protocol simple.
e. Although NFS clients use caches heavily, this does not introduce correctness problems since caches are always validated before use.

D

a. no: can open file twice and get same file handle
b. no: within a kernel, you know all the files that are shared. doesn't help in a single machine
c. no: stateless file server and client smart
d. yes if client retries, it's ok
e. no they don't validate cache before use.

4.

a. On a multi-CPU machine, single and multiqueue schedulers can both make effective use of cache affinity
b. Because hardware is generally agnostic about which OS it should run, a GNU/Linux platform can host a macOS guest about as efficiently as it could host a GNU/Linux guest
c. To maintain flexibility, some virtual-memory systems let an application specify different page sizes for different arrays in the program.
d. In a multilevel page table, two page table entries at different levels cannot map to the same physical page, because the hardware prohibits this.
e. For best performance, a virtual memory manager should typically choose dirty pages as victims, as opposed to clean pages.

A

a. yes. both can know what things are likely to be cached
b. Not a question of hardware, but system calls.
c. Possible but more of a pain. Conventionally, not used. Hardware won't let it happen. Page size fixed
d. can have a shared page in 2 spots in virtual address space. Also, for IPC.
e. choose clean pages – don't have to write them out.

5.

a. The main problem with fsck and similar programs is that they are too unreliable
b. Log-structured filesystems are a mistake for flash devices, since their main goal is to avoid seek overhead for writes, and flash devices don't seek.
c. Checksums are ineffective for data integrity in file systems, since a device that can lose or permute ordinary data can also lose or permute checksum.
d. Unlike 'fdatasync', the 'fsync' system call can arrange for the contents of a symbolic link to be safely synchronized to secondary storage.
e. Although programs like 'git' maintain version histories, these histories are independent of the version histories used to implement filesystems, which means that running 'git' atop a log-structured filesystem will maintain 2 sets of histories

E

a. It's supposed to be reliable.
b. Log-structured filesystems work well with SSDs.
c. Fundamentally missing the point of a checksum. Wrong
d. Since the symbolic link can be stored in the data blocks (path longer than 64 bytes) , both calls can do it. Not just fsync.

6.

a. Every filesystem has a structure called an inode, which stores the metadata for a file.
b. Although filesystems typically store user data in fixed-size blocks, they store directory data in varying length structures instead of blocks.
c. In a Unix-style filesystem, directory entries cannot cache any file metadata, other than inode numbers, since metadata is kept in inode and the cache might become invalid.
d. Security is a good reason why GNU/Linux lacks a system call openi(I, FLAGS) which would work like open(NAME, FLAG) except with an inode number parameter I rather than a filename NAME.
e. Symbolic links are more flexible than hardlinks as they allow links to cross directory boundaries and allow links to files in a parent directory.

D

a. no. not all filesystems have inodes
b. no.
c. file type can't change: can be cached.
d. yes -> permissions!
e. more flexible, but wrong reasons. No restriction on either for directory boundaries, only filesystem boundaries.

7.

a. Hard links can be used to create two different directory entries in the same directory, i.e., entries that have the same name but different inodes.
b. Every Unix filesystem contains at least 2 directory entries, though they may both have the same inode number.
c. Link counts prevents cycles from being created in the directory structure
d. Two different filesystems cannot contain the same inode number
e. The last-modified time of a file cannot decrease; it can only increase or stay the same.

B

a. Wrong. Same inode.
b. yes . and ..
c. no. (Can only be used by programs to detect a cycle.)
d. No.
e. can decrease (touch)

8. In this question, assume that no files are being modified (perhaps by some other process) during the actions being described, and all the system calls succeed.

a. If you call ‘stat’ with two different file names and ‘stat’ reports the same filesystem and inode number (st_dev and st_ino) both times, the link count might also be 1 both times.
b. If you call ‘stat’ with two different filenames and ‘stat’ reports the same inode number and same file system, the last modified time might disagree.
c. If you call ‘stat’ twice with the same filename, it might report differing st_dev, st_ino pairs.
d. The readdir system call is redundant, since you can find the directory entries individually by calling ‘stat’ on each of the directory’s files. The main reason for readdir is efficiency, not functionality.
e. If you call readdir several times in succession on the same file descriptor, readdir could return same filename more than once.

A

a. yes -> Symlinks. rc = 1, but different filenames.
b. nobody is changing the file system -> last modified must agree
c. wrong. if RO then same result by calling stat twice
d. wrong. readdir can find directory entries. stat can only check inode contents.
e. wrong. sequentially iterates through it.

9.

a. The plain elevator algorithm is fairer than circular scan algorithm
b. Anticipatory scheduling is more efficient than eager (work-conserving) scheduling.
c. Batching typically increases throughput and fairness when accessing blocks of a disk
d. Flash-based storage devices are simpler to deal with than hard disks, because if the block size is kept constant, then one can assume all input and output operations have roughly the same cost.
e. GNU/Linux is not a good match for a RT-11-style system because when GNU/Linux creates a file it cannot easily communicate the file’s desired size to the kernel

E

a. not fair. biased to center.
b. anticipatory guesses the correct answer. but so does eager
c. throughput yes, but not fairness
d. input is faster, but rewrite is not.
e. yes

10.

a. Suppose a thread T owns a spin lock L. Then T must unlock L, i.e., no other thread can unlock L.
b. Suppose a thread T has successfully called sem_wait on a semaphore S. Then T must call sem_post on S, i.e., no other thread can call sem_post on S.
c. Because 2 different threads can simultaneously call sem_wait on the same semaphore successfully, race conditions can still occur if semaphores are used, if they are not used carefully
d. Suppose a thread T has successfully acquired a binary semaphore (blocking mutex(B. Then T must release B, i.e, no other thread can release B.
e. It can make sense to have two condition variables for the exact same condition, to improve parallelism when many threads all depend on the same collection.

C

a. not true. thread can hand the lock to someone else.
b. not true. Same as a
c. yes
d. false. Same as a.
e. no. they can just have one. many = not synchronized.

11.

Would it make sense to use SSL/TLS within an operating system on a single, multi-core machine? For example, a CPU might use SSL/TLS when issuing commands to a flash drive. If so, explain the advantages of using SSL/TLS internally and how you would go about setting it up. If not, explain why not, and give a more-conventional approach that would make sense for security and explain why.

12.

Suppose you execute debugfs command 'clri 2'. What bad thing will happen to your ext2 file system? Is this something you can recover from, at least partially? If so, how; If not, why not?

13.

Suppose you have two programs S and M, both of which use a single lock to control access to a large shared array with A elements. The programs are identical except that S uses a spin lock and M uses a mutex. Both programs bottleneck accesses to the array, so you decide to use finer-grained locking and divide the array into N subarrays of size $\frac{A}{N}$, each subarray with its own independent lock. Which program, S or M, will benefit more from this change? If your answer depends on sort of operations that the program is doing, explain that dependency.

14.

Suppose you want to extend GNU/Linux by having conditional symbolic links where the condition is an arbitrary user-defined program. For example, the symlink from 'a' to 'sort –c foo?b:c' would resolve to 'b' if the command 'sort –c foo' exited with status 0, and would resolve to 'c' otherwise. Describe the main challenges you see in implementing this extensions.

15

Why can't the following shell script be converted to simpsh format?

```
#!bin/sh
sort | uniq
```

Give a simple change to lab 1 that would allow simpsh to execute the equivalent of the above script in a relatively natural way.

16. Suppose Intel and IBM announce a new processor, the Ozette Lake processor. It has an unusual feature: in user mode, it executes only unprivileged x86-64 instructions, but in kernel mode it executes only 64-bit z/Architecture instructions (either privileged or unprivileged). The z/Architecture ABI is quite different from the x86-64 ABI; among other things, it is a big endian processor designed for mainframes. GNU/Linux has been ported to the z/Architecture using a C compiler that generates z/Architecture instructions instead of x86-64 instructions

a. Would it be possible to implement a Linux-based OS on this processor without going to extreme lengths such using a software based emulator for the x86-64 architecture or the z/Architecture? If so, explain the most important problems you would need to overcome to support such an OS; if not, explain why it would not be feasible.
b. Assume that the answer to (a) is "yes" or that it's "no" and you're willing to put up with the performance problems of a software based emulator. Explain the relationship between system calls and marshalling/unmarshalling in your OS.

17. Suppose we're worried about the power consumption of spin locks, in that CPU chews up energy while spinning. Implement the lock and unlock function for spinlocks that use the following primitives, while using less power than the implementation discussed in class when code is spinning. The 'pause' primitive tells the code to wait for a bit and to remove speculative compare instructions from its pipeline: this uses less power.

```
void pause(void) { asm volatile("pause\n": : : "memory"); }
unsigned xchgl(unsigned *ptr, unsigned x)
{
    asm volatile ("xchgl %0, %1"
            : "=r" ((unsigned x))
            : "m" (*(volatile unsigned *)ptr), "0" (x)
            : "memory");
    return x;
}
```