Final Exam
CS 111, Principles of Operating Systems
Summer 2018

Name: _____

Student ID Number: _____

This is a closed book, closed note test. Answer all questions.

Each question should be answered in 2-5 sentences. DO NOT simply write everything you remember about the topic of the question. Answer the question that was asked. Extraneous information not related to the answer to the question will not improve your grade and may make it difficult to determine if the pertinent part of your answer is correct. Confine your answers to the space directly below each question. Only text in this space will be graded. No question requires a longer answer than the space provided.

1. One approach to keeping track of the storage space used by a particular file on a device would be linked variable length extents, in which the file descriptor would point to a section of the device where some variable amount of space had been allocated to the file. The last two words of that space would be a pointer to the next extent used to store more of the file's data and a length of that extent. What advantages would this approach have over the Unix-style block pointers stored in an inode? What disadvantages?

This approach could support arbitrarily large files. Also, it could do so with relatively few linked segments, assuming memory had not yet been fragmented, since each of the variable length extents could be quite large. One disadvantage is that traversing the file will require some extra I/Os. How many would depend on how many extents were in the file, which would depend on how the file was created (many small writes might lead to many small extents) and fragmentation issues. A second disadvantage is the it would be impossible to support sparse files with this particular approach. [Linked extents for file systems are discussed in lecture 13, slides 47-51. The approach is also discussed in Chapter 40, page 8. Fragmentation related to variable length extents is discussed in lecture 5, slides 28-30.]

2. A typical secure session over the Internet uses both symmetric and asymmetric cryptography. What is each used for, and why is that form of cryptography used for that purpose?

Asymmetric cryptography is used for initial authentication of one or both parties, and for secure session key establishment. Asymmetric crypto works well for this since distribution of public keys allows authentication of an otherwise unknown partner, which, with Diffie-Hellman key exchange, allows secure session key establishment. Symmetric cryptography is used for bulk transport of data once a session key is established. It is not as well suited for bootstrapping authentication and key establishment, but is computationally cheaper than asymmetric crypto, and thus is a better choice once a session key has been established. [Lecture 15, slides 53 and 54 goes over an example. The chapter on distributed system security discusses combined use of these approaches. The chapter on cryptography and lecture 15, slides 42-43 discuss the relevant advantages and disadvantages of the styles of cryptography.]

3. What is an advantage of using a stateless protocol in a distributed system?

A stateless protocol has at least two advantages. First, it has better reliability properties. Since the remote partner in a stateless exchange does not try to maintain information about the state of the exchange, failure of the local partner causes no problems. The local partner needs to expend more effort, perhaps, for each communication, but he also need not try to maintain any state about the remote partner, so failure of that machine will be easier to recover from. Second, for servers handling large numbers of clients, stateless protocols have important scaling advantages. Since the server need not remember information about its hundreds or thousands of clients, it can potentially handle more clients. Since each client is likely to work with only a few servers at once, on the other hand, there is no scaling issue offloaded onto them. A possible third advantage is that stateless protocols allow more ready horizontal scaling of systems. Each request can be sent to a different machine, so the load balancing problem is easier. Other advantages might be acceptable, depending on details. [Stateless protocols and systems are discussed in lecture 16, slides 9, 12, 13, and especially slide 41. Related issues are also discussed in the Distributed Systems: Goals and Challenges reading.]

4. Why is redirect on write a good strategy to use in file systems that are to be run on flash devices?

Flash memory can only be written once without requiring a large and performance-expensive erase cycle. Thus, a file system that does not perform writes in place, but instead writes new data in a new location and updates pointers, will fit better into the flash paradigm. Since there are no overwrites of existing data, the flash's write-once disadvantage can be largely ignored. Since flash memory has equal performance for sequential and random access, redirect-on-write's resulting scattering of data throughout the memory will not impose performance penalties. [Lecture 14, slide 48 specifically covers this issue. Chapter 44, section 44.7 does not discuss this by the name of redirect on write, but covers the important concepts.]

5. Unix-style operating systems (such as Linux) keep two types of in-memory kernel-level data structures to keep track of activities involving open files: open file instance descriptors and in-memory inodes. What is the purpose of each?

The in-memory inode serves as a kernel data structure to keep track of the fact that the file is open, to provide access to its inode metadata, and to provide information about which of its blocks are in memory and which are still on the disk. It is shared by all processes that have the file open, and thus is used to record things relevant to all processes using the open file. The open file instance descriptors are used to keep track of specific instances of opens of a file performed by processes and the particular characteristics of that open, such as whether it is open just for read or for read/write, whether read-ahead is requested, etc. While multiple processes can share such an instance, they only do so when sharing things like the mode of the open and the current pointer into the file are intended. [Lecture 13, slides 38-42.]

6. Describe an advantage of a first fit memory allocation strategy and a disadvantage of such a strategy, including why the strategy has that characteristic.

First fit tends to find suitable allocations more quickly than best or worst fit strategies, since it does not need to examine each free memory segment to determine its suitability. Once any suitable segment is found, first fit chooses it. A disadvantage is that the front of the free list tends to get fragmented quickly, since first fit will carve allocations out of the first suitable segment it finds, starting from the head of the list. In addition to obvious fragmentation problems, this characteristic will tend to cause searches to get longer as time goes on, reducing one of first fit's advantages. [Lecture 5, slide 38. Chapter 17, page 12.]

7. Why does code based on event-loop synchronization need to avoid blocking?

Event loop synchronization does not typically use multithreading. Instead, it waits for events to occur and calls management code to handle events as they happen. If any of the management code blocks, then, the system will not check for other events that might occur until the code unblocks. Performance will thus be poor. [Chapter 33, page 5.]

8. How does the cooperative approach to switching between processes and OS code work? Why is this approach not used in most operating systems?

In the cooperative approach, processes are expected to voluntarily yield every so often, which will allow the operating system to regain control without requiring a mechanism (such as a timer interrupt) to involuntarily halt the processes. This approach requires both careful thought and a certain amount of public-spiritedness on the part of all application programmers, since if they don't figure out when they should yield, or choose not to yield for selfish reasons, the OS won't get to run. Since we cannot count on application programmers to always do the right thing, we run the risk of highly unfair behavior. Worse, an infinite loop in a program might only be solvable by rebooting. [Chapter 6, pages 6-7.]

9. In an operating system context, what is meant by a convoy on a resource? What causes it? What is the usual effect of such a convoy?

A convoy in this context refers to a set of processes (or other computing entities) that all need to make exclusive use of a shared service, device, or object. If multiple entities are queued waiting for the object, and the service time to handle one entity exceeds the interarrival time for another non-queued entity to need access to the object, it will increase the length of the queue. In effect, a convoy kills all parallelism between participating entities, since generally all but one of them is queued waiting for the resource. [Lecture 9, slides 24-27.]

10. Consider the following proposed solution to the Dining Philosophers problem. Every of the five philosophers is assigned a unique letter A-E, which is known to the philosopher. The forks are numbered 0-4. The philosophers are seating at a circular

table. There is one fork between each pair of philosophers, and each fork has its own semaphore, initialized to 1. `int left(p)` returns the number of the fork to the left of philosopher p, while `int right(p)` returns the number of the fork to the right of philosopher p. These functions are non-blocking, since they simply identify the desired fork. A philosopher calls `getforks()` to obtain both forks when he wants to eat, and calls `putforks()` to release both forks when he is finished eating, as defined below:

```
void getforks() {
if (left(p) < right (p))
{
     sem_wait(forks[left(p)]);
     sem_wait(forks[right(p)]);
}
else
{
     sem_wait(forks[right(p)]);
     sem_wait(forks [left(p)];
}

void putforks() {
     sem_post(forks[left(p)]);
     sem_post(forks[right(p)]);
}
```

Is this a correct solution to the dining philosophers problem? Explain.

Yes. In brief, the solution leverages avoidance of circular waiting to avoid deadlock. In this solution, no philosopher will ever hold a fork while waiting for a lower numbered fork. If any philosopher holds two forks, he can eat and we don't have deadlock. The only ways possible for philosophers not to be able to obtain two forks when they want them are:
1). no philosopher can obtain any fork; not possible here since if a fork is free and a philosopher wants it, he gets it
2) all philosophers hold one fork; not possible here since that implies that some philosopher holds fork X, but needs to obtain fork Y (Y<X) to eat, which can't be possible since the lower numbered fork is always obtained first
In essence, we are using the resource ordering solution to avoid deadlock. To get full points, the answer must say something more than "resource ordering," offering some explanation of why this prevents circular waiting. [Lecture 10, slide 30.Chapter 32, pages 7-8.]