CS 111 Final exam

Kevin Zhang

TOTAL POINTS

87.5 / 100

QUESTION 1

1 Linked extents for file systems 10 / 10

✓ **- 0 pts Advantage and Disadvantage explained correctly**

- 5 pts either advantage or disadvantage not explained

- 10 pts Entirely incorrect description of advantage and diadvantage

QUESTION 2

2 Internet cryptography 10 / 10

✓ **- 0 pts Correct**

- 2.5 pts What is symmetric cryptography

- 2.5 pts What's asymmetric cryptography

- 2.5 pts symmetric crypto usage

- 2.5 pts asymmetric crypto usage

QUESTION 3

3 Stateless protocols 10 / 10

✓ **- 0 pts Correct - mention of the resulting reliability or scalability**

- 5 pts partially correct answer about saving time/ space

- 10 pts incorrect

QUESTION 4

4 Redirect on write for flash 10 / 10

✓ **- 0 pts Correct**

- 5 pts Partially correct answer

- 10 pts Incorrect answer

QUESTION 5

5 File descriptors and nodes 7.5 / 10

- 0 pts Correct

- 5 pts Incorrect answer for open file instance descriptors

- 5 pts Incorrect answer for in-memory inodes

✓ **- 2.5 pts Partial credit for open file instance descriptors**

- 2.5 pts Partial credit for in-memory inodes

OFIDs are used to keep track of specific instances of opens of a file by processes, and the characteristics of those opens.

QUESTION 6

6 First fit memory allocation 10 / 10

✓ **- 0 pts Correct**

- 4 pts Incorrect answer for advantage

- 4 pts Incorrect answer for disadvantage

- 1 pts Incorrect/missing explanation why the strategy has the advantage

- 1 pts Incorrect/missing explanation why the strategy has the disadvantage

QUESTION 7

7 Event loops 10 / 10

✓ **- 0 pts Correct**

- 5 pts Signals would be received, but could not be handled till the block is cleared. Issue isn't correctness or progress, but poor performance.

- 10 pts Issue is that it's single threaded, so blocking for one event prevents handling other events, killing performance.

- 2 pts Not deadlock, just serialized performance.

- 8 pts Event loop synchronization is usually single threaded.

- 8 pts Event loop synchronization is not sender/receiver based. Issue isn't lost events, it's poor performance.

- 10 pts Event loop code does yield and can be interrupted. The issue is its own performance.

- **1 pts** There might be a preemptive scheduler, but that only helps other processes, not the one running the event loop.

- **2 pts** Not really CPU waste, but serialized handling of the events. CPU might be running other processes not part of the event loop.

- **8 pts** Issue isn't lost events, but poor performance.

- **10 pts** Not a correct description of event loop synchronization.

- **2 pts** Since event loop synchronization is usually single threaded, blocking one thread IS blocking all threads.

- **4 pts** Not just a delay for particular events, but poor throughput of the overall work of the process.

- **2 pts** Not really about locking. There is not necessarily any lock associated with this method.

- **3 pts** Entire core won't block, just the event loop code, which is usually single threaded.

QUESTION 8

8 Cooperative switching 10 / 10

✓ **- 0 pts Correct**

- **7 pts** Core problem is that yielding is under programmer control. They might not yield properly.

- **5 pts** Reason is we can't rely on application writers to yield when they should.

- **10 pts** Completely wrong. (See chapter 6, pages 6-7.)

- **1 pts** Processes can yield in the middle and be rescheduled later.

- **8 pts** Not about taking turns. About allowing processes to determine when they will hand over control to the OS.

- **5 pts** Oddly, no explanation of the approach, but the right reason for not using it.

- **8 pts** Mostly wrong, but some elements of reasons not to use it correct.

- **4 pts** Doesn't mean there are no system calls or hardware protection, just that OS can't preempt the running process.

- **5 pts** Kind of backwards. User code is not preempted and is trusted to voluntarily yield to the OS.

QUESTION 9

9 Convoys 0 / 10

- **0 pts** Correct

- **3 pts** Service time vs. interarrival time is key for causing convoys.

- **5 pts** Not an accurate description of the problem or its results.

- **10 pts** No answer

✓ **- 10 pts Totally wrong. (See lecture 9, slides 24-27.)**

- **4 pts** Core bad effect is serialization of processes.

- **2 pts** Queue of waiting processes is major element of the problem.

- **2 pts** Problem isn't resource throughput, since convoyed resource is heavily used. Problem is serialization of jobs.

- **2 pts** Generally convoyed resources are locked and resource can't be taken away from them until complete.

- **7 pts** Not about fairness, but about bottlenecks.

QUESTION 10

10 Dining philosophers solution 10 / 10

✓ **- 0 pts Correct**

- **10 pts** Solution is correct. Total ordering in acquisition of forks.

- **8 pts** In your scenario, A would acquire fork 0 first, then fork 4. B blocks waiting for 0. C gets 2, then gets 1, since B is blocked and doesn't ask for 1. D can't get 2. E gets 3. Either A or E get 4. No deadlock. Two philosophers eating, and when they are done, two others can eat.

- **9 pts** At least one philosopher picks up left first and one picks up right first, due to total ordering of fork acquisition. So impossible for all five philosophers to have a fork in the same hand.

- **10 pts** Not proper description of semaphore behavior in context of described system.

- **0 pts** According to solution, no philosopher ever puts down a fork if he has acquired it.

- **9 pts** In your scenario, E grabs the fork to his right, while others grab fork to their left. Either he gets it or the other philosopher needing it gets it, and the other one gets no forks at all. Some philosopher ends up with two forks. No deadlock as a result of resource ordering.
- **2 pts** It's the circular dependency condition that does not hold.
- **3 pts** Need better explanation.
- **3 pts** It's possible for interspersals of semaphore operations, but that won't lead to deadlock in this solution.
- **10 pts** Solution is correct. Your description doesn't match your picture, since A isn't next to fork 0, D isn't next to fork 2, etc.
- **8 pts** Problem you outline isn't a deadlock. A philosopher might never end up eating due to dynamics, but some philosopher is always able to eat.

- **8 pts** Not a deadlock. A is eating, and when he finishes eating, the forks he releases will allow at least one other philosopher to eat. Processes holding locks that get stuck aren't a deadlock problem.
- **8 pts** Not necessary for two philosophers to eat at once. If one philosopher is currently eating, and when he finishes at least one other philosopher will be able to eat, and so on, solution is sufficient. This one is, by avoiding circular dependencies.
- **7 pts** Not a deadlock. Problem doesn't require a fair solution, just one with no deadlocks. If someone is always able to eat, no deadlock.
- **0 pts** Click here to replace this description.

Final Exam
CS 111, Principles of Operating Systems
Summer 2018

Name: Kevin Zhang

Student ID Number: 104939334

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 contiguous space had been allocated to the file. The last few 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?

Linked variable extents allow files to have a variable size that fits the particular files request and makes requesting more space contiguously easy by finding the next free block and pointing the current extent to that file. Variable sized partitions decrease internal fragmentation and is easier to store in a file's control block. However linked variable extents make random accesses to a file extremely difficult to implement efficiently. One must traverse through the linked list in order to find the offset desired. Furthermore linked lists with variable length causes a significant amount of external fragmentation - Blocks of data are strewn throughout memory in variable length so the small segments between allocations accumulate.

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 to send a public key to the client authenticate the server with the client. The public key along with a 3rd party certificate is distributed to users/clients who verify this certificate in order to validate the public key. Once the public keys are authenticated, a session can be negotiated and a symmetric key is then established using Diffie-Hellman key exchange to establish a joint symmetric key that makes session communication more easily performed. This symmetric key is encrypted w/ popular hashing/encryption algorithms like AES. Both parties now know a secret symmetric key that no one else knows so they communicate in a session using this private symmetric key. Asymmetric key encryption allows the client to establish this session and symmetric key encryption

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

Stateless protocols like Https allows servers and distributed systems to resume its operation after a crash or restart without the saved state of a previous execution. It doesn't require any type of session tracking or saving and restoration mechanism and because of this advantage, makes resuming service as well as daily routine tasks much more efficient. Not only does Https not require extra storage/ I/o to store session states, It also allows for faster restarts and clearer recovery providing better availability.

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

Flash devices have a requirement where first of all if a block is written to, it must be erased and in order to erase and rewrite a block, one must erase that entire block. Redirect on write is a good strategy because it prevents the need to erase on every write but instead writes in long contiguous buffered writes to the next available free space, eliminating the need to perform 2 I/os (erase first, then write) Furthermore if we don't use redirect, we might even have to erase data that is already being used which results in another I/o where the data needs to be copied Redirect on write allows us to make long efficient buffered writes, save I/o operations, and bundle the large erases to the end with garbage collection which drastically increases the efficiency of flash file systems

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?

In-memory inodes are inodes that are read into memory that contain the information of a particular node (file or directory). Reading it into memory gives the user fast access to file information including file type, size etc without needing to access the disk (slower)

File instance descriptors on the other hand are per process data structures that contain pointers to inodes. These file descriptors are created by a process when the process opens a file and they serve as an individual process' resource handler for that particular inode. All file operations including read and write can be performed on the file descriptor and the descriptor is allocated/deallocated by the process.

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

First fit memory allocation is very efficient because it doesn't require a full traversal of the memory freelist every time a new request is made to memory. It works particularly well when memory has a lot of free blocks. Other algorithms such as best fit would need to traverse the entire list to find the best fit. However the disadvantage of this allocation strategy is the amount of fragmentation that arises as the system grows older. There is internal fragmentation because the first fit very rarely fits the requested size exactly resulting in unused free space inside each allocated block. There is also external fragmentation as blocks are allocated and deallocated that resides between allocated blocks that are free but aren't fit to the requested size resulting in more and more wasted space, a huge problem and undesirable scenario for memory allocation.

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

Event loops or event based concurrency systems are designed to only use 1 thread. In this way there are no concurrency issues and the loop can process events as they come in. However, because there is only 1 thread, performing a synchronous I/O that blocks will cause this entire event loop to hang, waiting for the I/O to finish, wasting resources and missing requests that come in. Event loops therefore require asynchronous I/O such that they can perform I/O without blocking and wasting CPU cycles. If there were blocking (besides necessary blocking like page faulting), event loops would constantly hang.

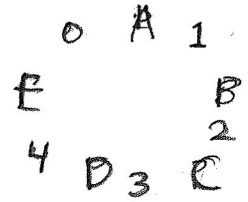
8. How does the cooperative approach to switching between running processes and running OS code work? Why is this approach not used in most operating systems?

The cooperative approach is the approach where the OS does not have the ability to terminate processes and assume control and relies on each side voluntarily yielding control to the other to allow it to run. This is not used because it is highly undesirable to allow the system to run on voluntary yielding. Many scheduling mechanisms and fail recovery mechanisms including round-robin scheduling and preemption will be nullified resulting in processes making their own determinations as to the time they will take. This can cause massive CPU starvation for processes that cannot obtain the CPU due to other malignant processes holding the CPU/memory hostage and can lead to all kinds of starvation problems. The OS is required to manage processes correctly so it cannot rely on processes to cooperate.

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?

The convoy of a resource mainly relates to resource allocation packet transportation by bundling resources and resource allocation and is usually a result of high number of requests for that resource. The usual effect of this convoy is that it is distributed in processing and the resource restricted to the highest priority requester.

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:

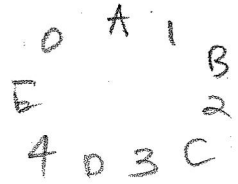



```
void getforks() {
if (left(p) < right (p))
{
     sem_wait(forks[left(p)]);
     sem_wait(forks[right(p)]);
}
else
{
     sem_wait(forks[right(p)]);
     sem_wait(forks [left(p)];
}

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

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

Assume that the philosophers are arranged around the table in this fashion



This is the correct solution since regardless of how the philosophers pick, there will always be a a fork left after the end of the first round of each philosopher picking a fork. After All philosophers pick, there will be one philosopher waiting for a fork and one untouched fork, that can be picked up by the nearest philosopher. This philosophers finishes eating, drops the fork and this allows the next adjacent philosopher to complete his set eventually returning to the original Waiting philosopher who picks his 1st fork up and then the second and finishes eating. Therefore no deadlock can occur so the algorithm is correct. On the other hand, if any philosopher were to pick up both forks before any other, then he/she will finish eating so the two forks will be available for 2 other philosophers to take eventually resulting in all philosophers eating.