My sites / 21S-COMSCI111-1 / Site info / Final Exam

Spring 2021 **- Finals week**

Spring 2021 - COM SCI111-1 - EYOLFSON

Started on	Tuesday, 8 June 2021, 4:45 PM PDT
State	Finished
Completed on	Tuesday, 8 June 2021, 7:40 PM PDT
Time taken	2 hours 55 mins
Grade	**131.50** out of 150.00 (**88**%)

Question **1**

Complete

5.00 points out of 5.00

Interfaces (5 minutes).

Why would you try to never do 1024 `write` system calls (each writing 1 byte) versus a single `write` call writing 1024 bytes?

We would try to call `write` a single time rather than 1024 times in order to avoid the overhead resulting from the `write` system call. Since `write` has to communicate between user space and kernel space each time it's called, which generates a lot of overhead, it's better to establish this communication once using a single `write` call that writes a large amount of data, rather than establish it many times using many `write` calls that each write a small amount of data.

Feedback:

Information

Threads (25 minutes).

Consider the following code:

```
#define NUM_THREADS 4

void* run(void*) {
  fork();
  printf("Hello\n");
}

int main() {
  pthread_t threads[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_create(&threads[i], NULL, &run, NULL);
  }
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_join(threads[i], NULL);
  }
  return 0;
}
```

Assume there's an existing process that begins execution at `main`.

Question **2**

Correct

5.00 points out of 5.00

How many new processes are created when the original process exits?

Answer: 4 ✓

The correct answer is: 4

Question **3**

Complete

4.00 points out of 5.00

Can one of the new processes become a zombie? If so give an example.

Since the child processes spawned by `fork` can potentially finish execution prior to the parent exiting, they can become zombie processes after they finish.

Feedback:

Question **4**

Complete

5.00 points out of 5.00

Can one of the new processes become an orphan? If so give an example.

Since the parent process never calls `wait` after forking, it will never address the child process it spawned. As a result, when the parent process is rejoined and then exits from `main`, the child will be left unaddressed, making it an orphan process since the parent has now exited.

Feedback:

Question **5**

Complete

10.00 points out of 10.00

How many times does `”Hello”` get printed? and most importantly, why does `”Hello”` get printed that many times?

`"Hello"` would be printed 8 times. 4 threads enter the `run()` function, and each fork, resulting in 8 threads about to run the `print` statement. We know that all 4 of the threads in the parent process must print `"Hello"` and rejoin before the process can exit, so there are 4 `"Hello"`s printed from the parent process. When `fork` is called from a thread, only that thread is cloned for the new process, so the 4 child process threads will each print `"Hello"` a single time. Since these are new processes, this printing will occur, regardless of what's happening in the parent process. Therefore, `"Hello"` would be printed 8 times.

Feedback:

Information

Disks (10 minutes).

Assume you have 4× 8 TB hard disk drives, and you want to use RAID. Using this configuration, answer the following questions:

Question **6**

Complete

2.00 points out of 2.00

How much usable space would you have if you used RAID 0, how many drive failures could you recover from?

Using RAID 0, you would have 32TB of usable space. Any drive failures would result in data loss, therefore you couldn't recover from any drive failures.

Feedback:

Question **7**

Complete

2.00 points out of 2.00

How much usable space would you have if you used RAID 1, how many drive failures could you recover from?

Using RAID 1, you would have 8TB of usable space. As long as one drive remained, you could avoid data loss, therefore you could recover from 3 drive failures.

Feedback:

Question **8**

Complete

2.00 points out of 2.00

How much usable space would you have if you used RAID 5, how many drive failures could you recover from?

Using RAID 5, you would have 24TB of usable space. If one drive fails, the others can be used to reconstruct each block in the drive, therefore you could recover from 1 drive failure.

Feedback:

Question **9**

Complete

2.00 points out of 2.00

How much usable space would you have if you used RAID 6, how many drive failures could you recover from?

Using RAID 6, you would have 16TB of usable space. Since the parity is duplicated, we can now recover from 2 drive failures.

Feedback:

Question **10**

Complete

2.00 points out of 2.00

You want to use your RAID to be able to recover from disk failure. Besides space and how many failures you could recover from, what is one other factor your should consider when choosing a RAID configuration?

Another factor that should be taken into consideration is the write performance of the RAID configuration.

Feedback:

Information

Page Tables (20 minutes).

Consider a three level page table using the Sv39 format. That means each level of page table uses 9 index bits, and there are 12 offset bits. Each PTE is 8 bytes, and physical addresses are 56 bits.

Question **11**

Complete

1.00 points out of 1.00

How large is a page?

Since the offset of the virtual address is 12 bits, that means the page size is `2^12` bytes.

Feedback:

Question **12**

Complete

2.00 points out of 2.00

Does each level of the page table fit on a page? Why?

No, each level of the page table doesn't fit in a single page. Since there are 9 index bits for each level of the page table, we know that there must be `2^9` PTEs in each page table. This means page tables are `2^9 * 2^3 = 2^12` bytes. This tells us page tables take up exactly 1 page. Therefore, the top level (level 2) of the page table can fit on a single page, as it consists of a single page table. However, level 1 of the page tables in this problem would consist of `2^9` page tables, which would require `2^9 * 2^12` bytes of space - clearly much more than a single page. Level 0 would require even more space. Therefore, level 2 fits on a single page, while levels 0 and 1 do not.

Feedback:

Question **13**

Complete

10.00 points out of 10.00

Assume that virtual address `0x00404038FF` maps to physical address `0x1118FF` explain how you'd use a three level page table to do this translation. For each level of the page table, give the indices you would use. For everything except the level 0 page table, you may assume the entry is an abstract page table. The virtual address in binary is `0b000_0000_0100_0000_0100_0000_0011_1000_1111_1111`.

We would start by taking the first 9 bits of the virtual address, which are `0b000000001`. This gives us the index into our level 2 page table (`1`). We then take the next 9 bits of the virtual address, which are `0b000000010`. This gives us the index into our level 1 page table (`2`). We finally take the next 9 bits of the virtual address, which are `0b000000011`. This gives us the index into our level 2 page table (`3`). We then use the index into level 2 (`1`) to access the correct level 1 page table. Then, we use the index into level 1 (`2`) to access the correct level 0 page table. Finally, we use the index into level 0 to access the PPN. Finally, we take the last 12 bits of the virtual address (the offset), which translates to `0x8FF`. We then append this onto the PPN to get our physical address: `0x1118FF`.

Feedback:

Question **14**

Complete

4.00 points out of 7.00

One variation is to use a gigapage (which is 1 GiB). The page tables still fit within the original page size. What modification would you have to make to resolve `0x00404038FF` as a gigapage?

We would have to consider that the offset of the virtual address/physical address is now 30 bits long. Therefore, the address would have to be modified accordingly, bringing its offset from 12 bits to 30 bits. Assuming the same virtual page number and offset, the new address would look like `0x0010100C00008FF`.

Feedback:

Question **15**

Complete

10.00 points out of 15.00

Locking (40 minutes).

You're tasked with implementing a bank account transfer that works with multiple threads. You create a bank account structure with a lock and an amount representing the dollar amount of the bank account. This bank makes no cents (get it?) and only tracks whole dollar amounts. You remember how Java implements monitors and write transfer to lock the entire function. Your initial implementation is as follows:

```
struct bank_account {
  pthread_mutex_t lock;
  int amount;
};

void transfer(struct bank_account *this,
              int amount,
              struct bank_account *that) {
  pthread_mutex_lock(&this->lock);
  if (this->amount >= amount) {
    this->amount -= amount;
    that->amount += amount;
  }
  pthread_mutex_unlock(&this->lock);
}
```

Give an example of a data race that could occur. You may explain it using abstract bank accounts such as: bank account A, B, and C.

Imagine a situation where there are 2 threads. One is trying to transfer from `A` to `B` and the other is trying to transfer from `B` to `A`. The first thread is about to increment the amount of `B` by `amount` (second line of `if`-statement) and the second thread is about to decrement the amount of `B` by `amount` (first line of `if`-statement). Since both threads are simultaneously writing to the value of `B->amount`, we have a data race.

Feedback:
You properly identified the data race and gave an abstract example, you should've given a concrete example with the actual memory writes and reads showing the race.

Question **16**

Complete

7.50 points out of 15.00

You change the code to the following:

```
void transfer(struct bank_account *this,
              int amount,
              struct bank_account *that) {
  pthread_mutex_lock(&this->lock);
  pthread_mutex_lock(&that->lock);
  if (this->amount >= amount) {
    this->amount -= amount;
    that->amount += amount;
  }
  pthread_mutex_unlock(&that->lock);
  pthread_mutex_unlock(&this->lock);
}
```

Your code now does not have any data races, but can deadlock. What are two things you could do to prevent the deadlock? You do not have to write any code. Explain what you would do to implement each strategy (in different paragraphs please).

One possible solution is to replace calls to `lock` with calls to `trylock`. This allows us to continue execution if the current thread is unable to secure the lock. From there, we can then unlock all held locks, effectively removing the thread's tendency to "hold and wait", which means deadlock can no longer occur.

A second possible solution is to force a strict ordering on how threads acquire locks. For instance, we can force threads to acquire locks for the `bank_account` with a lower memory address first, and then acquire locks for the `bank_account` with a higher memory address after. This makes it impossible for a situation where Thread 1 is holding resource `A` and Thread 2 is holding resource `B`, but Thread 1 is waiting on `B` and Thread 2 is waiting on `A`. Instead, both threads would have to compete for the resource with the lower memory address before trying to get the lock on the other resource. In other words, this method removes the ability for threads to enter a "circular wait" scenario, making deadlock impossible.

Feedback:
Simply replacing lock with trylock isn't going to work. You can only continue if you got both locks, and you need to give up your single lock if you only got one.

Question **17**

Complete

10.00 points out of 10.00

Someone suggests using two variables is wasteful, you can just use a semaphore that keeps track of the amount of money in each account.They provide the following implementation that has no data races:

```
struct bank_account {
  sem_t amount;
};

void transfer(struct bank_account *this,
              int amount,
              struct bank_account *that) {
  for (int i = 0; i < amount; ++i) {
    sem_wait(&this->amount);
    sem_post(&that->amount);
  }
}
```

Explain how you could STILL effectively deadlock a process running 2 threads using this code. Please provide an example.

The function `sem_wait()` will wait until the semaphore has a value of greater than `0`. Imagine that both threads were trying to transfer from `A` to `B`. The `amount` in `A` is `1`, and both threads are attempting to transfer `5`. The first thread that acquires the semaphore will decrement `A->amount`, making it `0`. Now, both threads are attempting to acquire the semaphore, however, it is `0`, so neither can. Since it is assumed no other threads are running, `A->amount` will remain `0`, it is impossible for either thread to acquire `A->amount`, and both threads are deadlocked.

Feedback:

Question **18**

Complete

6.00 points out of 6.00

File Systems (20 minutes).

In Lab 4, using a block size of 1024, you created a symbolic link (or soft link) to the name `hello-world`. Why were you able to store the content of the symbolic link within the inode itself? Explain why this optimization only works for names less than or equal to 60 bytes.

The only data that symbolic links store is the name of their target. As a result, they don't need to be allocated data blocks if the name of the target is small enough. We can therefore use the extra space in the inode that would usually be reserved for pointers to hold the name of the target. inodes have 12 direct pointers, 1 direct pointer, 1 doubly-indirect pointer, and 1 triply-indirect pointer. That makes 15 total pointers, which are each 4 bytes. `15 * 4 = 60`, so 60 bytes is the amount of space freed up if we don't use pointers. Therefore, in order for this optimization to work, we can have a target name with a maximum of 60 bytes.

Feedback:

Question **19**

Complete

7.00 points out of 7.00

Regular files are allocated across given a number of blocks. The inode structure records the number of 512 byte blocks used and an explicit size record. Someone claims that you could just calculate the size using the number of 512 byte blocks. Explain why you need to explicitly record the size.

Regular files are not required to be divided exactly into 512-byte blocks. A 200 byte file and a 500 byte file both require one 512-byte block. As a result, it's impossible to calculate the exact size of a file just from the number of data blocks it is allocated, since there may be unused bytes within a block.

Feedback:

Question **20**

Complete

7.00 points out of 7.00

The inode (generally) points to the content of a file. Explain the condition(s) required for the kernel to delete an inode and free the blocks it points to.

In order for the kernel to delete an inode, all hard links to that inode must be deleted first. If any hard links point to an inode, it isn't safe for the kernel to delete it.

Feedback:

Information

Memory Allocation.
Assume you have a buddy allocator that initially has a single 1024 byte free block. You receive an allocation of 120 bytes.

Question **21**

Complete

2.00 points out of 2.00

What is the size of block used for the preceding allocation?

The free block would be divided into two 512-byte blocks. One of the 512-byte blocks would be divided into two 256-byte blocks. One 256-byte block would be divided into two 128-byte blocks. This is the smallest size we could go to and still fit a 120-byte allocation, so the size of the block used is 128 bytes.

Feedback:

Question **22**

Complete

3.00 points out of 3.00

What are the sizes of the free blocks after the allocation and how many are there?

There is a 512-byte free block, a 256-byte free block, and a 128-byte free block. There are 3 total free blocks.

Feedback:

Question **23**

Correct

2.00 points out of 2.00

How many bytes are lost due to internal fragmentation for the 120 byte allocation?

Answer: 8 ✓

The correct answer is: 8

Information

Assume you have a single 1024 byte free block again, and you receive 180 byte sized allocations.

Question **24**

Incorrect

0.00 points out of 2.00

How many TOTAL 180 byte allocations could a single 1024 byte free block hold?

Answer: 5 ✗

The correct answer is: 4

Question **25**

Complete

7.00 points out of 7.00

What is the TOTAL amount of internal fragmentation for all your 180 byte allocations with the initial 1024 byte free block. Why is this amount of fragmentation acceptable?

When using a buddy allocator, only 4 180-byte allocations can be made in a 1024-byte free block, as opposed to the 5 180-byte allocations that could theoretically fit. The size of the free block for these allocations is 256 bytes, which results in 76 bytes of internal fragmentation per allocation. `76 * 4 = 304`, so there are 304 bytes of total internal fragmentation.

This fragmentation is acceptable because of the design of the buddy allocator. By taking on this fragmentation, we ensure all our free blocks have sizes that are powers of 2, which simplifies the process of searching for free blocks and merging them back together when they become free again. Therefore, we're willing to give up this space inefficiency.

Feedback:

Question **26**

Complete

4.00 points out of 4.00

Assume we only had 128 byte allocation, causing no internal or external fragmentation with the buddy allocator. Why would we still want to use a slab allocator for only 128 byte allocations?

We'd still want to take advantage of the ease of tracking free blocks using the bitmap of the slab allocator, rather than the linked lists of the buddy allocator. This improves the time efficiency of the allocator, as bitmap modification can be done faster than linked list traversals.

Feedback:

Question **27**

Complete

2.00 points out of 2.00

Virtual Machines (10 minutes).
Assume you're thinking about implementing a type 2 hypervisor. What CPU mode does the guest kernel execute in?

A type 2 hypervisor has the guest kernel execute in user mode.

Feedback:

Question **28**

Complete

8.00 points out of 8.00

You're now tasked with implementing the type 2 hypervisor. The CPU you're implementing this for has few privileged (kernel) instructions, and no non-privileged (user) instructions behave different in kernel mode. What is the best implementation strategy? Explain how it would work at a high level.

Under these circumstances, we'd use trap-and-emulate. This method's biggest weakness of not being able to detect the difference between privileged and non-privileged instructions is negated by the fact that there are no non-privileged instructions that behave differently in kernel mode. In addition, the inefficiency brought on by the trap behavior is negated by the face that there are very few privileged instructions.

This method works by catching all privileged instructions executed by the guest kernel with a trap, which redirects control to the VMM, which can then emulate the effects of the privileged instruction to update the VCPU. Afterwards, control is then returned to the guest kernel. This process helps us avoid unintentional side effects that the privileged instruction may have on the host.

Feedback:

◄ Midterm

Jump to...

Lab 1B Week 1 ►