CS 111: Operating System Principles

Final Exam Spring ’21

Instructor: Jon Eyolfson

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?

> *There is considerable overhead switching between user and kernel mode. You should reduce the number of calls to get the best performance. Calls that only write 1 byte at a time will be mostly overhead.*

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`.

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

> *4, one for each thread.*

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

> *Yes, the parent process does not `wait` for any process. Any child process that finishes before the parent will be a zombie until it gets re-parented to a subreaper.*

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

> *Yes, the parent process only waits that all of its threads are finished. Depending on the scheduling, it can exit before any of the new processes.*

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

> *It gets printed 8 times. Each of the 4 threads in the original process spawn a new process each. The new process will only have a single thread, that copies the thread executing `fork`. So, each of the 4 threads prints "`Hello`", then each of the 4 new processes also print "`Hello`".*

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:

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

> *You'd have 32 TB of usable space, but can't recover from any failures.*

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

> *You'd have 8 TB of usable space, but can recover from 3 drive failures.*

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

> *You'd have 24 TB of usable space, and can recover from 1 drive failure.*

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

> *You'd have 16 TB of usable space, and can recover from 2 drive failures.*

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?

> *You should also consider read and write performance. For example, RAID 6 provides more redundancy at the cost of reduced read and write performance as well as less usable space.*

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.

How large is a page?

> *4 KiB.*

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

> *Yes, the page table is 2^{12} bytes, because there are 2^{9} entries and each entry is 2^3 bytes. We do this to reuse the PTE, otherwise we may have to store an entire physical address. Our machine also expects pages, so it will be faster. We can index the correct page by just using the offset directly.*

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 will have a single level 2 page table. The first index we use is `0x01`, which points to a level 1 page table. We then use the second index of `0x02` to find our level 0 page table. Within our level 0 page table we'd look up the PTE at index `0x03`, which contains the physical page number `0x111`. We'd then add that to our `0x8FF` offset.*

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?

> *For a gigapage, we'd essentially only have one level 0 page table. Because we have a 1 GiB page, the lower 30 bits would be the offset. We'd simple look up the PTE at index `0x01` and use that as our page. We'd assume our process only uses gigapages, so there's no overlap or anything we have to handle.*

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.

> *Assume that account A transfers $20 to account C, and account B transfers $20 to account C. Account C initially has $60. Thread A locks account A and reads C's current amount ($60). Thread A pauses. Thread B locks account B and reads C's current amount (also $60). After, it doesn't matter which thread executes, they'll both update C's new amount to $80 when it should be $100.*

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).

> *The first strategy would be to enforce a total order on the locks. We could use the memory address of the bank account, or create an account number for each bank account. We would then always try to lock the bank account with the lowest memory address or lowest account number first. This puts an ordering on our locks.*
>
> *The second strategy would be to try to acquire the second lock only once. If we can't acquire it, we release the original lock we had and try again. This eliminates the "hold and wait" requirement of deadlocking.*
>
> *There's another problem, this and that can be the same bank account. In this case the code tries to lock the same mutex twice, which isn't allowed. You'd have to add a check to see if the 2 accounts are the same, if they are we can return immediately. Transferring funds to yourself, if allowed, would not result in a gain or loss anyways.*

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.

> *Assume bank account A initially has $0. One thread tries to transfer $1 from account A to B. Another thread tries to transfer $1 from account A to B. Both threads are now waiting for account A to have a non-zero amount, and will never make any progress.*

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 name was small enough that we could store it in the space reserved for pointers to blocks. For Lab 4, we had 15 addresses (12 direct, 1 indirect, 1 double-indirect, and 1 triple-indirect) and each were 4 bytes (60 bytes total). We couldn't use more bytes because all the other inode fields are used.*

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.

> *The blocks indicate how much space is currently allocated for the file not how many are USED. When displaying a file, we only want to display the number of bytes used, and not uninitialized values until we match the block size. We could also use the difference between the allocated space and used space to calculate the internal fragmentation of the file system.*

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.

> *To delete an inode, there must be no hard links pointing to it. Hard links are just names within a directory that point to an inode. If there are no hard then there's no way for a user to access the file. At that point the kernel my delete the inode and free blocks.*

Memory Allocation (20 minutes).

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

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

> *128 bytes.*

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

> *There is one 512 block, one 256 block, and one 128 block.*

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

> *8.*

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

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

> *4.*

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?

> *The internal fragmentation across all allocated blocks is 304 bytes. This is acceptable because there is no external fragmentation. When we free the memory we can coalesce the blocks back to a single contiguous block.*

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?

> *The buddy allocator uses free lists, which is slower than simply using a bitmap with the slab allocator.*

Virtual Machines (10 minutes).

Assume you're thinking about implementing a type 2 hypervisor. What CPU mode does the guest kernel execute in?

> *The guest kernel operates only in user mode.*

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.

> *Since we have a clean interface, we can implement everything using the trap-and-emulate approach. We would receive a trap every time our guest kernel tries to execute a privileged instruction. We would emulate it in user mode continue executing the guest kernel.*