CS 423 Midterm
10/18/2022

2:00pm – 3:15pm (75 minutes)

CLOSE BOOK && NO COMPUTER

Tips:

- Give **short, concise answers** rather than long, vague ones (we grade by correctness, not by length).
- There are **many problems** and **you only have 75 minutes**. **Do as much as you can** and **don't get stuck on one problem**.
- **Don't be stressed out** if you don't finish everything. We didn't expect that but we will be impressed if you do :)
- If you find that you have to make assumptions to solve the problem, write down your assumptions (e.g., "I assume this is an x86 architecture" or "I assume a Linux kernel version >2.6.3"). On the other hand, most problems are kept at a high level and do not need such assumptions.

Sections (50 points)	**Points**
1. The OS Interface (8 points)	
2. Process and Thread (12 points)	
3. Synchronization (10 points)	
4. Scheduling (10 points)	
5. Memory management (10 points)	

Name:_____ **NetID:**_____

1. The OS Interface (6 points)

There have been endless debates on what a sane OS kernel should do and not do, and we have discussed in class about different designs, including monolithic kernels (such as Linux) and microkernels (such as Mach). But, oftentimes, kernel hackers and researchers do things just for fun. Jinghao, our TA, wants to add a new function to the kernel to let his favorite Gentoo Linux fry eggs for him. The idea is that his kernel will talk to the smart egg fryer next to his workstation (yes, now is an IoT era). To let user-space applications leverage the new functionality, he wants to introduce a new system call: `` `int fryeggs(int nr_eggs)` ``.

(1) As a first step, he needs to figure out how to add a new system call in his kernel. Could you help him? Please write down the basic steps of adding a new system call. Think about what components need to be revised? (3 points)

(2) Learning from MP1, Jinghao prefers a loadable kernel module (LKM) instead of built-in kernel code. Can he implement the egg-fry functionality in LKM with the **same** system call interface? Why or why not? (2 points)

(3) Jinghao notices that his egg fryer is not working properly. Fortunately, he can retrieve the logs from the fryer into his module. To study the logs, he needs to get the logs into user space from his module. Jinghao decides to implement a read callback just like the one in MP1 so that he can use `cat` to read the logs. However, Jinghao has a special version of `cat` that uses a really small buffer of 64 bytes when issuing a `read()` system call, which is much smaller than the size of the log (generally more than 4KB). How can Jinghao implement the callback so that he can use `cat` to dump all of the logs? (3 points)

Note:

- Assume that `cat` repeatedly issues `read()` calls until there is nothing to read (i.e. the last `read()` returns 0).
- The log is stored in a char buffer called `data` and has global scope in the module; its length is stored in the static variable `data_sz`.
- The data will not change during and between `read()` calls issued by `cat`.
- You can use `memcpy()`, `memset()`, and `copy_to_user()` freely but you don't have to use all of them.

```
ssize_t read_fn(struct file *file,
    char __user *buf, size_t size, loff_t *pos)
{
    // Please fill the code









}
```

2. Process and Thread (12 points)

Process and thread are two different abstractions. Threads are treated as “light-weight processes”. However, as we have seen in the class, both a process or a thread are treated as a “task” in the Linux kernel with the same structure representation.

(1) Can you explain why threads are lightweight compared with processes? Please provide two aspects (2 points)

(2) How does the Linux kernel manage processes and threads? In other words, if one wants to find all the information about a given process (identified by its PID), how to find the information? (2 points)

(3) How to create a process and how to create a thread? Please give the exact system or lib calls you will use. (2 points)

(4) We know that we can use the `kill` command to kill a process. Can you explain how `kill` works? (2 points)

(5) Knowing that threads are more lightweight, Tianyin has a “smart” idea – let’s just replace all the processes into threads so we can make computation faster with smaller overhead. All three TAs think it’s a dumb idea because process provides stronger isolation level than threads. Why is that the case? (2 points)

(6) Tianyin then improves his idea – why not develop another abstraction that provides stronger isolation than thread inside a process? Is it feasible? If not, why? If so, how to develop it and what resources need to be isolated? (2 point)

3. Synchronization (10 points)

We discussed the Thread-Safe Bounded Queue problem discussed in the class, which is a classic producer-consumer problem. Xuhao finds it a good idea and wants to implement it in a wacky OS on a machine from Tianyin.

Learning that condition variables are a handy abstraction to implement a bounded queue (if you forget the implementation, you can find it in the attachment), Xuhao decides to use condition variables for synchronization. He finds that the wacky OS does not have the `pthread` library (which supports condition variables). It only provides mutex.

With the graduate student spirit, Xuhao decides to implement condition variables himself using the available mutex. Recall that condition variables need three operations, `` `wait()` ``, `` `signal()` ``, and `` `broadcast()` ``. Hence, the goal is to implement these three operations.

Xuhao first defines a condition variable to be a data structure of a queue and a lock

```
struct cond_var {
    struct queue thd_queue;
    lock_t qlock;
};
```

Please fill out the implementation.

Notes:

1. We follow the Mesa semantic because we learned in the class that the Hoare semantic is hard to implement.
2. Not every empty space is necessary to fill. We grade by the correctness of your code.

```
void wait(struct cond_var *cv)
{  /* 4 points */
   // Please fill the code




    mutex_acquire(cv->qlock);  /* protect the queue */
    thd_queue.enqueue(current); /* enqueue the current task */
    mutex_release(cv->qlock); /* done with the queue*/

    // Please fill the code









}


void signal (struct cond_var *cv)
{   /* 4 points */
    int tid;
    // Please fill the code









    tid = cv->thd_queue.dequeue();

    // Please fill the code
```

```
}

void broadcast (struct cond_var *cv)
{  /* 2 points */
   // Please fill the code









}
```

4. Task Scheduling (10 points)

Let's take a closer look at one of the simplest scheduling algorithms, Shortest Job First (SJF). SJF selects for execution the waiting process with the smallest execution time. SJF is non-preemptive.

(1) Is the following statement correct? – SJF guarantees to minimize the average amount of time each process has to wait until its execution is complete. If you think it is correct, please prove it. If you think it is wrong, please give a counterexample (i.e., a different schedule that has a smaller average time). (2 points)

(2) SJF has a few variants. For example, its preemptive version is called Shortest Remaining Time First (SRTF). SRTF selects the process with the smallest amount of time *remaining* until completion. SRTF is considered advantageous over SJF. Could you reason out its advantages? (2 points)

(3) Another variant of SJF is Highest Response Ratio Next (HRRN). HRRN is also nonpreemptive, just like SJF. Differently, in HRRN, the next job is not that with the shortest (estimated) run time like in SJF, but that with the highest response ratio defined as:

response ratio = (waiting-time-of-a-process-so-far + run-time) / run-time

What is the advantage of HRRN over SJF? (2 points)

(4) One big problem of SJF is that it needs to estimate the run time of every job, which is non-trivial for real-world computing tasks. Therefore, real-world systems often use algorithms that approximate SJF. Can you describe one approximate algorithm of SJF and also explain why it approximates SJF behavior? (4 points)

5. Memory Management (10 points)

Siyuan has been working on memory management systems for 64-bit machines. Today, he finds an ancient 32-bit machine in the lab. To have fun besides research, Siyuan decides to write a virtual memory system for the 32-bit machine.

(1) How large is the physical memory a 32-bit address space can support? (2 points)

(2) Siyuan decides to make the page size to be 1KB (as the regular page size is 4KB on 64-bit ISA). In this case, how many bits should the virtual page number (VPN) have? (2 points)

(3) What should be the size of a page table entry (PTE)? Note that it's better to make it a power of 2, e.g., if only 3 bits are needed, then let's make it 4 bits; if 5 bits are needed, let's make it 8 bits. Also, let's don't consider other utility bits for now. (2 points)

(4) Since the address space is small ("only" 32-bit), Siyuan plans to just use a linear page table which is indexed by VPNs. Given the PTE size, how much memory would the linear page table take? (2 points)

(5) Learning from the idea of multilevel radix tree design for page tables, Siyuan plans to implement a 2-level radix tree for the page table. With this design, how large will the top-level table be? (Assume top and bottom levels have the same size) (2 points)

/* Intentionally leave as blank pages which can be used for more space or for draft. */

Attachment:

Implementation of Thread-Safe Bounded Queue using condition variables

```
lock_t lock;

struct cond_var full  = { .qlock = &lock };
struct cond_var empty = { .qlock = &lock };

#define MAX 2048U
char *buf[MAX];

u32 front, tail;

char *get(void)
{
    mutex_acquire(&lock);
    while (front == tail) {
        wait(&empty);
    }
    item = buf[front % MAX];
    front++;
    signal(&full);
    mutex_release(&lock);
    return item;
}

void put(char *item)
{
    mutex_acquire(&lock);
    while ((tail – front) == MAX) {
        wait(&full);
    }
    buf[tail % MAX] = item;
    tail++;
    signal(&empty);
    mutex_release(&lock);
}
```