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 (6 points)	
2. Process and Thread (12 points)	
3. Synchronization (12 points)	
4. Scheduling (10 points)	
5. Memory management (10 points)	

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? (4 points)

- add a new syscall entry to the syscall table (arch/x86/entry/syscalls/syscall_64.tbl):

```
451   common      fryeggs     sys_fryeggs
```

- add the new syscall implementation with `SYSCALL_DEFINE` macro:

```
SYSCALL_DEFINE1(fryeggs, int, nr_eggs)
{
  // some implementation
}
```

- The userspace can use `syscall(__NR_fryeggs, nr_eggs)` to invoke the new syscall.

Rubric
- 1pt if write something
- 1pt if mention updating syscall table
- 1pt if mention syscall implementation

(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)

It would not work, because one cannot add a syscall in a kernel module.

One way to do this is to extend `ioctl`:
- Create a `proc` file entry (or character device) for this module
- Implement the `ioctl` hook in the `proc_ops` struct, with a new command `IOCTL_FRYEGG`
- The userspace can then do

```c
int fd, ret;

fd = open("/proc/fryeggs", O_RDWR | O_NONBLOCK);
if (fd < 0) { // yes, you should always check syscall retval
  perror("open");
  return 1;
}

ret = ioctl(fd, IOCTL_FRYEGG, nr_eggs);
if (ret < 0) {
  perror("ioctl");
  return 1;
}
```

to fry the egg.

Rubric
- 1pt if answered no
- 1pt if mention updating syscall table cannot be done in LKM

Or
- 1pt if answered yes
- 1pt if mention the use of proc fs as in the mps

(3) Jinghao notices that his egg fryer is not working properly. Fortunately, he was able to retrieve the logs from the fryer into his module. To study the logs and debug his fryer, he needs to get the logs in userspace. 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?

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.

```c
ssize_t read_fn(struct file *file,
    char __user *buf, size_t size, loff_t *pos)
{
    // 0.5pt for checking pos and read-size
    // 0.5pt for returning 0 if nothing to read
    unsigned long to_read = min(size, data_sz - *pos);
    int nr_nocpy;

    if (!to_read)
        return 0;

    // 0.5pt for the use of copy_to_user
    // 0.5pt for checking return value
    nr_nocpy = copy_to_user(buf, data + *pos, to_read);

    if (to_read == nr_nocpy)
        return -EFAULT;

    // 0.5pt for updating pos
    // 0.5pt for returning correct size to user
    *pos += to_read - nr_nocpy;
    return to_read - nr_nocpy;
}
```

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)

1. Shared address space
2. less context switch overhead
3. Easier IPC with global variables
4. More efficient creation and destruction
5. Shared fd table
6. Small overhead context

Rubric
- any 2 from above (1 each)

(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)

- `struct task_struct (get_task_by_pid)`
- `PCB / TCB`

Rubric
- any 1 from above

(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)

Process: `fork()`
Thread: ~~`clone(fn, stack, CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | CLONE_SYSVSEM, arg)`~~
Or just `pthread_create()`

Rubric
- 1pt Process: `fork()` / `clone()`
- 1pt Thread: `pthread_create()` / `clone()` <- needs to specify flags if answer `clone()`

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

`kill` works by sending a signal/exception to the target process. To kill a process, send `SIGKILL` (9) to it.

Rubric
- 2pt mentions sending signal/exception to the process

(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)

This is because threads usually share the same address space (because of the `CLONE_VM` flag usually used by thread library calls), where one thread can poke the memory of another.

Rubric
- 2pt mentions the `unsafe` nature of shared memory

(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)

It is feasible. To create such a thread, one can just call clone without cloning
The address space, FS info, fd table, sig handler table, system V semaphore adjustment values.

Rubric
- any 2 from the above list (1 each)
- or 1 with detailed/novel solution
- 1 pt if stated not feasible but mentioned a bunch of stuff to isolate

3. Synchronization (12 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
      thread_sleep(my_pid);
      do_sth();
```

We screwed up the interface here.

So, if there is a thread sleep of the current thread then we give 4pt. If the program is obviously incorrect, e.g., thread sleep before the mutex, then we cut points accordingly.

```
}


void signal (struct cond_var *cv)
{   /* 4 points */
    int tid = -1;
    // Please fill the code


      mutex_acquire(cv->qlock);
      if (!cv->thd_queue.empty()) {
            tid = cv->thd_queue.dequeue();
      }
      mutex_release(cv->qlock);
    // Please fill the code
      if (tid >= 0)
            thread_continue(tid);
```

If there is a correct dequeue logic and continue the thread, then we give 4 points. Points are cut based on the correctness of the pseudo code.

```
}
```

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

      mutex_acquire(cv->qlock);
      while(!cv->thd_queue.empty()) {
            tid = cv->thd_queue.dequeue();
            thread_continue(tid);
      }
      mutex_release(cv->qlock);
    // 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)

Correct. Say we have jobs with time interval length of {t_1, t_2, … t_n} (sorted by increasing order. We want to argue that the arrangement A = {t_a1, t_a2, … t_an} where a_i = i created the smallest average wait time. Given n fixed, it is equivalent to argue that A creates the smallest total wait time which is sum_{j=1}^{n-1} (n-j)*t_aj.

say there's arrangement B that doesn't arrange the shortest job first; more specifically, we have t_bj > t_b{j+1}, and we argue that B can be improved by flipping values of bj = b{j+1}.

The waiting time that contributed by these two jobs are

$$(n-j)t_{b_j} + (n-(j+1))t_{b_{j+1}}$$
$$= (n-j)t_{b_{j+1}} + (n-j)t_{b_j} - t_{b_{j+1}}$$
$$> (n-j)t_{b_{j+1}} + (n-j-1)t_{b_j}$$

Thus, any arrangement B that doesn't follow the shortest job first ordering can be improved by flipping two adjacent job arrangements.

correct: 1pt

proof:

- good proof (without loss of generality): mentioned switching the short task after a longer one results in longer wait time with generality, or give the formula of total wait time and mentioned the first job will be count the most in the total wait time: 1pt
- not good proof (with loss of generality): give a concrete example, but loss of generality: 0.5pt
- bad proof: say other unclear things: 0pt

other reasonable answer (e.g. SRTF, preemptive sched): 2pt

(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)

It runs small jobs faster than SJF because it preemptively schedules incoming jobs that have shorter time than the remaining of the current running job.

explain the advantage, etc. 2pt
say something (which is true) but does not show the advantage: 1pt
say some advantage (which is true but not obvious) without explanation: 1pt
wrong advantage/explanation: 0pt

(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)
1. It helps the starvation problem of SJF. if one waits for too long, it will get arranged first.
mentioned process wait too long can get arranged first: 2pt
mentioned helps starvation: 2pt
say advantage but didn't show the reason: 1pt

(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)

- say the name of the algorithm and explain the algorithm, but they are not match (say RR but explain MLFQ, or explanation is wrong): 2pt
    - if explained why it approximate SJF, +1pt
- only made a correct explanation without say why it approximate SJF: 3pt
- a unreasonable customize algorithm: 0pt
- describe a reasonable customized algorithm and explain why it approximate SJF: 4pt
- say a known algorithm (RR, MLFQ, etc) without explain (or wrongly explain) hot it approximate SJF: 3pt
- say a known algorithm (RR, MLFQ, etc) and explained how it approximate SJF: how shorter task can finish before longer task: 4pt

Answer from OSTEP : it first assumes it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process. In this manner, MLFQ approximates SJF.

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)

2^32
-0.5 if 2^32 -1 smallest

(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)

1KB is 2^10, we have 32 bits in total, so VPN has 32 - 10 = 22bits.
offset correct +1
VPN correct +1

(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)

32bits
since 2^5 is the smallest power of 2 that is larger than 22
need at least 22 bits (+1)

(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)
2^27 bits or 16MB.
Indexed by VPN, so the table has length of 2^22. This multiplied by 32 bits, we have 2^27 bits intotal. (16MB).

-1 know 2^size of VPN * PTE size but miss numbers though they have it correctly before.
+2 know 2^size of VPN * PTE size but miss numbers but use the wrong number from previous question

(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? (2 points)

2^16 bits, or 8 KB.
VPN has a size of 22 bits, two levels has the same size, so the same number of entries is 11 bit. Thus, each page table has 2^11 = 2048 entries. This multiplied by 32 bits (size of PTE), we have 2^16 bits which is 8KB .
number of bits for indexing: +1
number of page table entries: +0.5
multiplied by size of PTE: +0.5

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

/* 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);
}
```