CS 423 Midterm (Close Book; No Cheat Sheet)

10/23/2025 (Thursday), In Class

Tips:

- **The exam duration is 1 hour.**
- **You are expected to do the exam independently.** No discussion is allowed.
- **You are not allowed to use unauthorized materials.** Textbook, class notes, cheat sheets, and electronic devices are prohibited.
- **Please write clearly and legibly.** If we can't understand your writing, then we won't give points. We won't guess anything beyond what you wrote.
- Give **short, concise answers** rather than long, vague ones (we grade by correctness, not by length).
- If you find that you have to make assumptions to solve problems, write them down (e.g., "I assume this is an x86 architecture" or "I assume a Linux kernel version >2.6.3"). Most problems are kept at a high level and do not need specific assumptions.

Sections (20 Points + 2 Bonus Points)	**Points**
1. Kernel Interface (5 points)	
2. Memory Management (7 points)	
3. MP-1 Continued (5 points)	
4. MP-2 Continued (5 points)	

Name:_____ **NetID:**_____

1. Kernel Interface (5 points)

We learned in the class that `strace` is the tool for tracing system calls from a user application. Cathy wants to trace system calls for `ls`, so she runs:

```
strace ls t > /dev/null
```

And below are partial traces given by `strace`:

```
execve("/usr/bin/ls", ["ls", "t"], 0x7ffe9be0f1f8 /* 82 vars */) = 0
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=211975, ...}) = 0
mmap(NULL, 211975, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f825c07a000
close(3)                                = 0
write(2, "cannot access 't'", 17cannot access 't')       = 17
write(2, ": No such file or directory", 27: No such file or directory) = 27
write(2, "\n", 1)                       = 1
exit_group(2)                           = ?
```

(1) Please explain what `mmap` does in line 6. (1 point)

(2) There is a `write` system call in line 7. Please explain what the first argument (the number 2) stands for, and explain how **this** write system call works. (2 points)

(3) In x86-64, the `syscall` instruction is a serializing instruction, which means that it cannot execute until all the older instructions are completed, and that it prevents the execution of all the younger instructions until it completes. Please explain why this is necessary. (2 points)

2. Memory Management (7 points)

We learned the mechanism of virtual memory management in the class based on the x86-64 semantics. As we know, memory capacity has been growing and terabyte-scale memory is already available on commodity servers. So, four-level page tables are no longer sufficient. As a result, Linux recently supported five-level page tables.

Let's think of a five-level page table, where each page-table page is 4KB, each page table entry is 8 bytes, and huge pages are 2MB and 1GB. We use L1 to refer to the last level of the page table, and L5 to refer to the highest level.

(1) How large virtual memory can this five-level radix-tree page table support? (1 point)

(2) We learned in the class that walking over a multi-level page table is slow. Apparently, a five-level page table is slower than a four-level page table. A proposal from the class is to merge the intermediate levels to reduce the height of the page table tree. Let's say we merge L2 and L3 and turn the page table tree back into four levels: L5, L4, merged(L3, L2), and L1. In this case, what is the minimal amount of memory a process will need to use for its page table? (2 points)

(3) Besides improving the hardware (e.g., TLB and PWC), can you give an idea to reduce off-TLB memory translation overhead, which only needs to change software? Please explain your idea, why does it help? (2 points)

(4) In this five-level page table, can you describe what is the most expensive page fault and what is the overhead to address the page fault? Please give a detailed explanation (2 points)

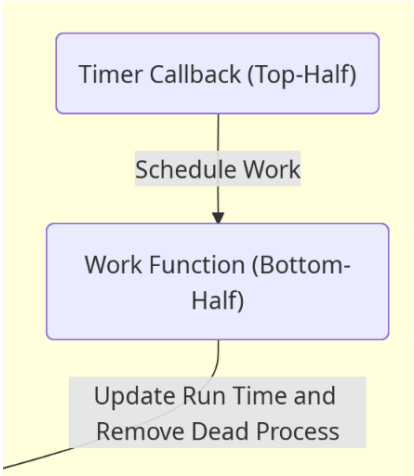
3. MP-1 Continued (5 points)

In MP-1, we have implemented a kernel module that measures the user space CPU time of a list of registered processes.

Cathy uses a single global lock to protect her MP-1 process list. When updating the CPU time for each process, Cathy first holds the global lock, and then iterates through the process list. While iterating, she updates the CPU time for each process. Finally, she releases the global lock.

(1) Peizhe thinks the critical section is too long. He argues that Cathy can unlock while updating the CPU time for each process, and then lock again to continue the iteration. Do you agree with his argument? If yes, explain how this can improve the performance. If not, explain why this does not work. (1 point)

Remember that we are required to use the 2-halves approach in MP-1, which is shown on the right. Timer callback, which is the first half, will schedule the work function, which is the second half.



(2) Why do we use such a 2-halves approach, instead of implementing a monolithic timer callback function? Please explain. (1 point)

Additionally, timer callback is based on `softirq`, which makes it an interrupt context. As we emphasized in the MP-1 documentation, interrupt context has many limitations.

(3) We can use spinlock in timer callback, but we cannot use mutex. Please explain the reason. (1 point)

When grading MP-1 submissions, Cathy had an idea of exploring a new way to test the result by calling `tail /proc/mp1/status`. But to her surprise, this crashed many submissions' kernels even when `cat /proc/mp1/status` worked as expected. She explored a bit by checking the system call traces:

```
strace cat /proc/mp1/status > /dev/null

execve("/bin/cat", ["cat", "/proc/mp1/status"], ...) = 0
# .. traces omitted
fstat(1, {...}) = 0
openat(AT_FDCWD, "/proc/mp1/status", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0666, st_size=0, ...}) = 0
read(3, "", 262144)                     = 0
```

```
strace tail /proc/mp1/status > /dev/null

execve("/bin/tail", ["tail", "/proc/mp1/status"], ...) = 0
# .. traces omitted
openat(AT_FDCWD, "/proc/mp1/status", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0666, st_size=0, ...}) = 0
lseek(3, 0, SEEK_CUR[   16.270596] BUG: kernel NULL pointer dereference,
address: 0000000000000000
# .. panic message omitted
[   16.273918] Kernel panic - not syncing: Fatal exception
[   16.273918] Kernel Offset: disabled
[   16.273918] ---[ end Kernel panic - not syncing: Fatal exception ]---
```

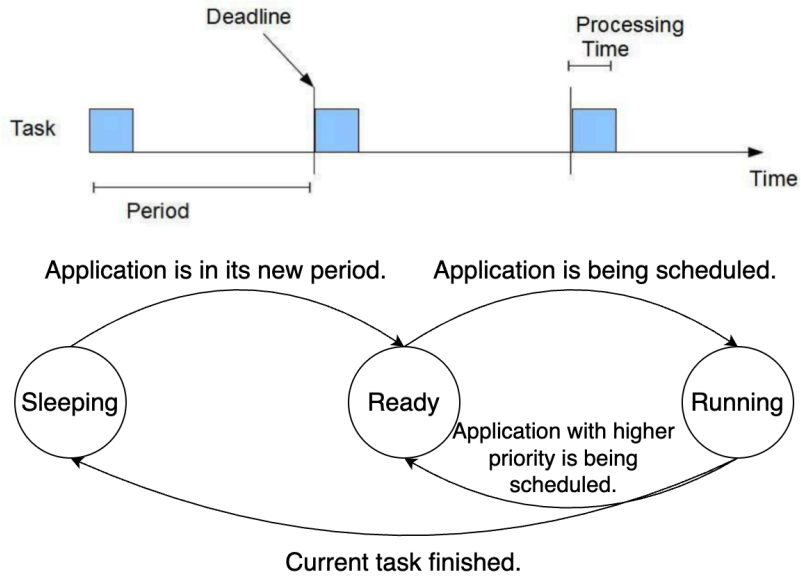
Oh, no! The kernel crashed during the `strace` run. Cathy was very curious about the root cause, so she brought up the Linux kernel source, and looked for the `proc_ops` interface as a reference:

```
struct proc_ops {
   int (*proc_open)(struct inode *, struct file *);
   ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
   ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
   ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t *);
   loff_t  (*proc_lseek)(struct file *, loff_t, int);
   int (*proc_release)(struct inode *, struct file *);
   // .. omitted
};
```

(4) Please help Cathy identify the possible cause for the kernel panic. You can directly mark on the logs/code above and briefly explain the cause. (2 points)

4. MP-2 Continued (5 points)

In MP-2, we have implemented a Rate Monotonic Scheduler (RMS), where each task is modeled by its period and computation time, and have three states: Sleeping, Ready, and Running. The task model, state machine, and state transition diagram is shown as follows:



Remember that each task must call `yield` after it finishes in the current period. In this section, you will work on the yield handler with your hardworking TA Peizhe, and explore the early/late yield problem. Peizhe's current implementation is provided below.

```
static int mp2_process_yield(int pid)
{
    struct mp2_pcb *process;
    mutex_lock(&mp2_pcb_list_lock);
    if (!(process = mp2_process_lookup_locked(pid))) {  // Look up the PCB
        mutex_unlock(&mp2_pcb_list_lock);
        return -ESRCH;
    }
    mp2_process_sleep_locked(process); // Put the task to sleep and set state
    mod_timer(&process->timer, jiffies + msecs_to_jiffies(process->period));
    mutex_unlock(&mp2_pcb_list_lock);
    wake_up_process(mp2_dispatcher_kthread);
    return 0;
}
```

(1) Peizhe noticed that his RMS was not behaving properly. **Initially all tasks run fine, but they will eventually miss deadlines, breaking the task model.** How can it break real-world RT applications? Please provide an example**.** (1 point)

(2) Considering the problem symptom and his implementation, please help him identify the problem. You can directly mark on the code at the previous page. (1 point)

As we already emphasized in the MP-2 documentation and walkthrough, your user application must register its computation time accurately. We now define the **early/late yield problem**: the user application calls yield earlier or later than its registered computation time.

(3) What are the consequences of the early/late yield problem in RMS? Please provide one example for each case (early/late). (2 points)

(4) Thank you, now we are able to detect the problem. Peizhe wants to resolve this problem by updating the inaccurate computation time in PCB. Do you agree with his solution? If yes, explain how this can avoid consequences above. If not, explain why this does not work. (1 points)