CS130 Operating Systems: Homework 1

Please submit a **PDF** file to Gradescope.

(Due on November 7, 2023)

1 Process & Thread (28pts)

In class, we learned about the process and thread, two of the OS's basic concepts, and the principles of the `fork()` system call as well.

a.(6pts)
• What resources are shared by threads belonging to the same process?
• In context switch, when the target thread meets what relationship with the original thread, do we need to switch the page table?

b.(6pts) When a process containing multiple threads uses `fork()` to create a child process, and it happens that one of the threads is waiting for input from an external device, if the child process duplicates all the threads of the parent process and does nothing else, there will be two threads waiting for input. If the original process contained only one thread, would the above problem still occur? Give your reasoning.

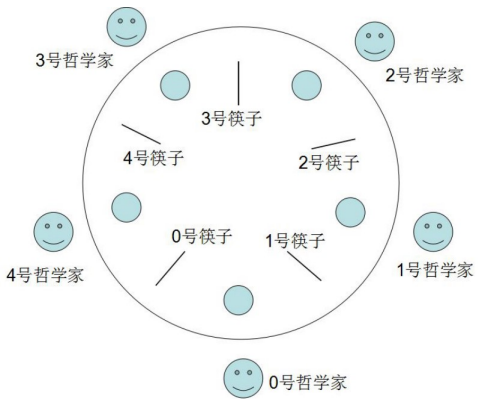
c.(8pts) If there was only one process, how many processes would there be after executing `if (fork() || fork()) fork();`? Give your computational procedure. **Assumption:** For `||` operation, if its left operand evaluates to true, then skip evaluating its right operand.

d.(8pts) User programs run at a lower privilege level, system calls are needed when some privileged operations are required (such as reading and writing files, sending signals to other programs, etc.). When a program enters the kernel state by calling a system call through interrupts or specialized processor instructions (such as syscall and sysenter), the stack pointer register is also modified pointing to another special stack for executing this kernel code. Assuming that this kernel code is switched to another code before it finishes running.

- Do you think that the program running at this point must still be the same user program that is calling the system calls?
- What are the possible sources of the code running at this point?
- In light of the above, when it comes to the creation and allocation of these special stacks, do you think it is better to have one per process or one per thread? Give your thought process.

2 Dining Philosopher (24pts)

Dining Philosopher is a classic problem. In the typical model, there are n philosophers and chopsticks and each Philosopher needs 2 chopsticks to eat, as shown in the following figure.



a.(5 pts) Let's use threads and locks to represent Philosophers and locks, respectively. The following code can cause deadlock at some cases. Suppose $n = 5$, please give an execution order of these 5 threads that will cause deadlock and briefly explain it.

```
lock chopsticks[5];

void Philosopher(){
    // get thread id, which is from 0 to 4
    int i = thread_id();
    while(true) {
        // get the left chopstick
        acquire(&chopsticks[i]);
        // get the right chopstick
        acquire(&chopsticks[(i+1)%5]);
        eat();
        release(&chopsticks[(i+1)%5]);
        release(&chopsticks[i]);
    }
}
```

b.(9 pts) To avoid deadlock, we can introduce tickets. If a philosopher want to eat, he/she needs to first get a ticket from the middle of the table. Filling in blanks in the following codes and briefly explain why there will not be deadlok (you don't need to use all blanks).

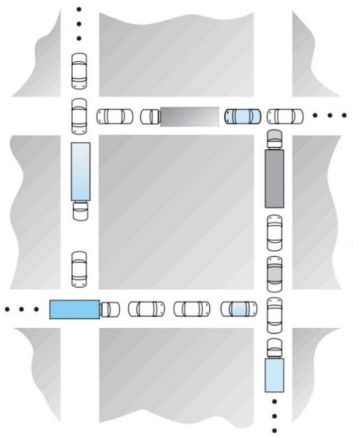
```
struct lock chopsticks[5];
struct semaphore tickets;

int main() {
    // init semaphore with value (same with Pintos)
    sema_init(&tickets, _________________);
    // start threads
    start();
}

void Philosopher() {
    int i = thread_id();

    while(true) {
        // use sema_up/sema_down to
        // control semaphore (same with Pintos)
        ----------------------
        ----------------------
        acquire(&chopsticks[i]);
        acquire(&chopsticks[(i+1)%5]);
        eat();
        release(&chopsticks[(i+1)%5]);
        release(&chopsticks[i]);
        ----------------------
        ----------------------
    }
}
```

c.(10 pts) Provide another solution for the Dining Philosopher problem. You can either write codes or explain your idea.

3 Lock & Deadlock (24pts)

a.(8pts) Briefly describe two approaches to avoid deadlock.

b.(8pts) Consider the traffic deadlock depicted in the picture:



- Show that the four necessary conditions for deadlock hold in this example.
- State a simple rule that will avoid deadlocks in this system.

c.(8pts) Compare the use of locks and semaphores in managing concurrent access to shared resources. Discuss their advantages and disadvantages, and provide specific cases where one is more suitable than the other.

4 Producer & Consumer (24pts)

Consider a multi-threaded program that simulates a simple producer-consumer scenario. The program consists of two types of threads: producers and consumers. Producers produce items and place them into a shared buffer, while consumers retrieve items from the buffer.

The shared buffer has a maximum capacity of N slots. To ensure proper synchronization and avoid race conditions, we use semaphores with 'wait()' and 'signal()' operations: `empty` and `full`, and a mutex, `buffer_mutex`, to protect access to the shared buffer.

Here are the rules for the producer and consumer threads:

Producers: - Producers 'wait()' when the buffer is full (i.e., `full` equals N), and they resume when the buffer has empty slots. - Upon resuming, they 'produce()' a new item to the buffer.

Consumers: - Consumers 'wait()' when the buffer is empty (i.e., `empty` equals 0), and they resume when there are items in the buffer. - Upon resuming, they 'consume()' an item from the buffer.

Fill the blanks for the producer and consumer threads, including the necessary initialization of semaphores and mutex, and the logic to 'produce()' and 'consume()' items from the buffer. Ensure that the code respects the synchronization requirements and avoids issues like race conditions or deadlock.

1. **Initialization (8pts):** Provide the initialization code for `empty` and `full` semaphores, and `buffer_mutex`, including initial values and any other necessary setup.

2. **Producer Code (8pts):** Write the code for the producer threads, including how they use `empty`, `full`, and `buffer_mutex` to 'produce()' items to the buffer. Explain the logic behind your code.

3. **Consumer Code (8pts):** Write the code for the consumer threads, including how they use `empty`, `full`, and `buffer_mutex` to 'consume()' items from the buffer. Explain the logic behind your code.

```
struct semaphore empty, full;
struct lock buffer_mutex;
int buffer[N];

void initialize() {
    ----------------------------------
    ----------------------------------
    ----------------------------------
}

void * producer(void * arg) {
    while (true) {
        ----------------------------------
        ----------------------------------
        produce();
        ----------------------------------
        ----------------------------------
    }
}

void * consumer(void * arg) {
    while (true) {
        ----------------------------------
        ----------------------------------
        consume();
        ----------------------------------
        ----------------------------------
    }
}
```