Question 1 (5 points). What is the most significant difference between processes and threads?

Answer: the key point is whether they share the same memory space or not. (5 points)

If not directly come to this point, but somehow related (e.g., thread is more lightweight, context switch overhead, communication methods), then give 2-4 points

Question 2 (5 points). Briefly discuss the main advantages and disadvantages of using a Microkernel architecture as opposed to a Monolithic Kernel.

An advantage of using a Microkernel architecture is that it can be more secure and stables, as faults in one service cannot easily propagate to others. (2.5 points)

A disadvantage is the potential performance overhead (low efficiency) due to the need for more context switches and inter-process communication (IPC). (2.5 points)

Microkernl: small, secure and stable, low efficiency, some services in kernel mode others in user mode, better modularity and fault isolation

Monolithic kernel: efficient communication, larger, complex, faster, all the operating system services run in a single address space

Question 3 (5 points). Briefly explain how a **system call** differs from an **API**, concerning their interaction with the operating system.

A system call is typically a request directed at the operating system's kernel, while an API call may be a higher-level abstraction that encapsulates system calls, but also provides services by invoking code running in user space. (5 points)

Question 4 (6 points) Suppose we develop a program with a thread library using a one-to-one thread model (i.e., Each user-level thread mapped to a kernel thread). Is it possible to support preemptive scheduling for the user-level threads using this thread library? Why?

Yes, (2 points)

This is because each user-level thread is paired with a distinct kernel thread. The operating system's kernel can independently manage these kernel threads, and therefore, it can preempt them based on its scheduling policies. (4 points)

Question 5. (9 points)

(a) **(4 points)** Suppose a CPU scheduler favors processes that have utilized the least amount of CPU time in the recent past. Does this scheduler favor I/O-bound programs or CPU-bound programs? Why?

It will favor I/O-bound programs (2 points)

Because I/O-bound programs have shorter CPU time (2 points)

Question 6 (10 points). Given a program as shown below. Suppose **P1** is a process executing this program on a modern Linux machine.

Program
void main(){
pid_t pid;
fork();
fork();
pid = fork();
if (pid == 0):{
printf("Hello!\n"); //print on the screen
}
else:{
printf("Goodbye!\n"); //print on the screen
}
printf("Finish!"); //print on the screen
}

(a) (5 points) Suppose all fork() are successful. How many processes are created in total, including **P1** and all the processes recursively created by **P1** (including, e.g., **P1**'s child processes, **P1**'s grandchild processes ...)? Why?

8. (2 points)

The first call to fork() results in two processes; each of those two processes then calls the second fork() which results in a total of four processes; the third call to fork() results in eight processes; (3 points)

(b) (5 points) How many times will the message "Hello!\n" be displayed? Why?

4. (2 points)

After second call to fork() but before third call to fork(), there are four processes. The third call to fork() results in eight processes, but only the child process (of which there are four) prints "Hello". "Hello!\n" is printed by the child processes created by the third fork() call.

(3 points)

(b) **(5 points)** With the above-mentioned scheduler, will CPU-bound programs suffer from starvation (i.e., have little chance for execution)? Why?

CPU-bound programs will not starve (2 points),

because I/O-bound programs often suspend themselves for I/O, so CPU-bound programs can execute (3 points)

Question 7 (20 points) Consider the following two threads running concurrently on a single-processor machine. The value at the shared memory address 2000 is initialized to 0. Assume that Thread 1 will be scheduled to execute first.

Thread_1

line	code	comments
1	main	# the entry point of the thread
2	mov $3, %ax	# %ax is initialized to 3
3	top	# a label
4	mov 2000, %dx	# get the value from the address to register %dx
5	add $1, %dx	# increment the value in register %dx by 1
6	mov %dx, 2000	# store the value back to the address
7	sub $1, %ax	# decrement the value in register %ax by 1
8	test $0, %ax	# compare the value in register %ax with 0
9	jgt top	# jump to "top" if the result of the comparison is "greater"
10	halt	# stop running this thread

Thread_2

line	code	comments
1	main	# the entry point of the thread
2	mov $5, %ax	# %ax is initialized to 5
3	top	# a label
4	mov 2000, %dx	# get the value from the address to register %dx
5	add $2, %dx	# increment the value in register %dx by 2
6	mov %dx, 2000	# store the value back to the address
7	sub $1, %ax	# decrement the value in register %ax by 1
8	test $4, %dx	# compare the value in register %dx with 4
9	jgt out	# jump to "out" if %dx is greater than 4
10	test $0, %ax	# compare the value in register %ax with 0
11	jgt top	# jump to "top" if %ax is greater than 0
12	out	# a label
13	halt	# stop running this thread

(d) **(4 points)** Suppose the interrupt is enabled and the interrupt could occur at any time (we assume that a thread switches to the other thread when an interrupt occurs). After both threads finish, is it possible for the final value of memory address 2000 to be **3**? Please explain why.

Yes.

Thread 1 switches to thread 2 before writing back 3 to memory 2000, then thread 2 executes and finishes, then finally thread 1 writes back 3.

(e) **(6 points)** Suppose the interrupt is enabled and the interrupt could occur at any time (we assume that a thread switches to the other thread when an interrupt occurs). After both threads finish, is it possible for the value of memory address 2000 to be **6**? Please explain why.

Yes.

thread 1 line 6, write 1 to 2000, (first iteration)

thread 2 load 1, execute for one iteration, write 3 to 2000

thread 1 load 3

thread 2 load 3

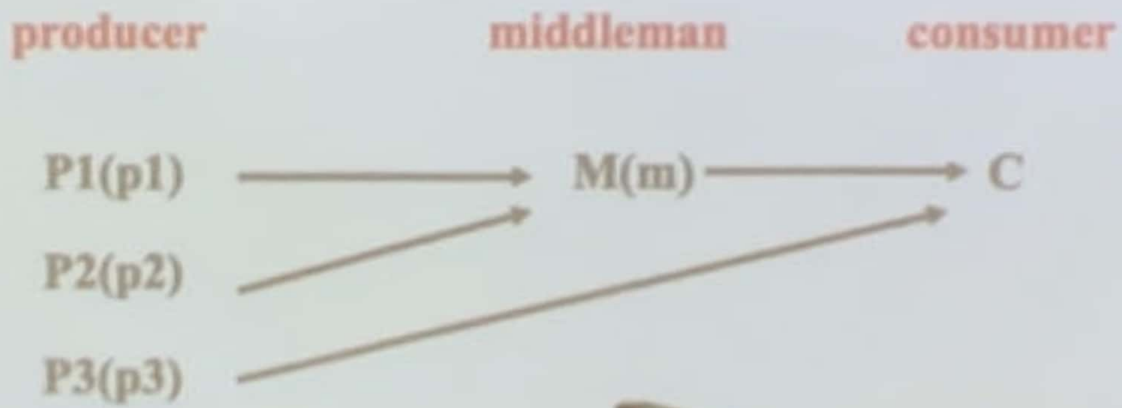
thread 1 write 4 to 2000

thread 2 write 5 to 2000 and halt

thread 1 load 5, add 1, write 6 to 2000

Question 11 (10 points).

- There are three producers **P1**, **P2**, and **P3** that can produce three types of products: **p1**, **p2**, and **p3**.
- There is a middleman **M** who consumes one **p1** and one **p2** to produce another type of product **m**.
- **P1**, **P2**, **P3**, and **M** all need to use the same production facility, so only one of them is allowed to produce products at a time
- There is a consumer **C**, which either consumes **m** or **p3**. When both **m** and **p3** are available, the **C** consumes **m**. When m is unavailable, **C** consumes **p3** instead.
- Each product is associated with a separate buffer of size **N**. When the buffer is full, the corresponding producer (or middleman) cannot put more products into the buffer. When the buffer is empty, the corresponding consumer (or middleman) cannot get that type of product from the buffer.



Question 12 (6 points). The bakery algorithm is another approach to achieve mutual exclusion in a concurrent environment. It assigns a unique number to each process and uses these numbers to determine the order of access to the critical section, similar to customers taking a number in a bakery. Please complete the pseudocode below.

```
Choosing[3] = {0,0,0}
Number[3] = {0,0,0}
Process(i) {
while (true) {
//grab the number
      Choosing[i] = 1;
      Number[i] = 1 + max(Number[0], Number[1], Number[2]);
      ① ______________________________;
      for (j=0; j <=2; j++) {
         while (Choosing[j] != 0);
         while (② ______________________);
      }
      // critical section...
      Number[i] = 0;
      // non-critical section...
   }
}
```

swer:

```
C:{
while True{
# Buy m goods
        if cnt_m > 0:
                wait(full_m)
                wait(mutex)
                # Consume m goods
                cnt_m -= 1
                signal(empty_m)
                signal(mutex)
```

```
else if cnt_3>0
        wait(full_3)
        wait(mutex)
        # Consume p3 goods
        cnt_3 -= 1
        signal(empty_3)
        signal(mutex)
}
}
```

The following gives the pseudocode samples of **P1** and **P2**. Please complete the pseudocode of **P3**, **M** and **C** according to the above description.

```
P1: {
    while (True){
        #produce p1
            wait(empty_1)
            wait(mutex)
            #add p1 to buffer
            cnt_1 += 1
            signal(mutex)
            signal(full_1)
    }
}
```

```
P2: {
    while (True){
            #produce p2
            wait(empty_2)
            wait(mutex)
            #add p2 to buffer
            cnt_2 += 1
            signal(mutex)
            signal(full_2)
    }
}
```

- **How does the Bakery Algorithm work exactly?**

At first, all process didn't choose the number and didn't get the priority.

```
Choosing[3] = {0,0,0}
Number[3] = {0,0,0}
Process(i) {
```

Enter the while loop of each process

```
while (true) {
//grab the number
```

Start to choose the number, Choosing variable help mark if a process has already hold a number

```
    Choosing[i] = 1;
```

The number of each process who want to enter the CS must greater than 0

```
    Number[i] = 1 + max(Number[0], Number[1], Number[2]);
```

When Choosing variable turns 0, it means this process has a number now and can compare with others

```
    ① _____Choosing[i] = 0 _____________;
```

Check all processes in the system

```
    for (j=0; j <=2; j++) {
```

Whether other processes has already hold a number

```
        while (Choosing[j] != 0);
```

Whether other processes want to enter the CS, whether other processes has the priority to enter

```
        while (②_(Number[j]!=0) && ((Number[j],j) < (Number[i],i)) );
    }
    // critical section...
    Number[i] = 0;
    // non-critical section...
}
```