Introduction to Operating Systems, 2024 Fall, NYCU CS
Question Sheet for Midterm Exam

Part A: Basic Problems

1. [5pts] In I/O scheduling, typically synchronous I/O requests are assigned a higher priority than asynchronous I/O requests. Why?
2. [5pts] Explain the entire procedure of interrupt handling, starting from the firing of a hardware event until the execution of the first line of code of the corresponding ISR. Your answer must (at least) involve the interrupt vector table, the CPU program counter and flag register, and the process stack.
3. [2pts] Consider the system virtual machine architecture, involving the host OS and a guest OS. Which mode (the *real* kernel mode or the *real* user mode) the following component is running under? Why?
    a. The kernel of the host OS
    b. The kernel of the guest OS
4. [4pts] Which one(s) of the following are implemented as a system call? Why?
    ✓a. read(), reading data from a file
    ✓b. fork(), creating a new process
    c. mmap(), mapping a file to the virtual address space of a process
    d. strcpy(), string copying from a source to a destination address
5. Answer the following questions regard the structure of OSes:
    a. [2pts] Why is microkernel considered more secure and more extensible?
    b. [2pts] Why is the monolithic structure highly popular for the operating systems of desktop computers and servers?
    c. [2pts] Why do many embedded operating systems not provide protection based on user-kernel separation?
6. [10pts] Let a process be executing the code in **Fig. 1**. Discuss the process state transition of the process from *statement A* to *statement B* (running→ various states … → running). Also explain why each transition is triggered.

```
main() {
        semaphore S=0;

        // statement A
        ...
        wait(S);
        ...
        // statement B
}
```

Fig. 1

7. [10pts] See the dummy code in **Fig. 2**. Which memory sections do the variables, including *a*, *b*, **p*, the malloc()'ed space, and *x*, belong to?

```
char x;

main() {
        foo(2);
}
foo(int a){
        char *p;
        int b;

        p=(char *)malloc(100);
}
```

Fig. 2

8. [7pts] Suppose that you receive an unknown POSIX-compliant thread library. How do you write a simple program to determine whether the library employs the M-1 model or the 1-1 model?
9. [7pts] In a multithreaded process, all its concurrent threads must have their own stack. Why is this design necessary?
10. Shortest-Job-First (SJF) is a basic scheduling algorithm.
    a. [2pts] Regarding the waiting time of jobs, what is the main advantage and disadvantage of SJF?
    b. [4pts] SJF is not suitable for CPU scheduling implementation but is more commonly seen in I/O scheduling. Why?
11. Answer the following questions regarding multiprocessor scheduling
    a. [$\overset{5}{3}$pts] What is load balancing and what is processor affinity? Why are they necessary?
    b. [$\overset{3}{5}$pts] Why does the Linux load balancing algorithm use a hierarchical approach, balancing workloads from NUMA nodes, physical cores, and then down to logical cores, instead of using a flat model that focuses solely on logical cores?
12. [10pts] A group of villagers shares communal meals from a large pot that holds up to *M* servings of stewed food. When a villager is hungry, he serves himself from the pot, unless it is empty. If the pot is empty, the villager wakes up the cook and waits until the cook refills the pot entirely. The unsynchronized version of villagers and the cook are shown in **Fig 3**. Use semaphores to solve this problem.

```
---villager code---
while(1){
 getServingFromPot();
 // one serving at a time
 eat();
}

---cook code---
while(1){
 putServingsInPot (M)
}
```

Fig. 3

13. Consider that a high-priority process P1 and a low-priority process P2 are scheduled preemptively on a single CPU. The two processes contend for two resources R1 and R2 through mutex locking. Answer the following regarding deadlock avoidance:
    a. [2pts] Use a Gantt chart (scheduling timeline) to demonstrate a deadlock that involves the two processes through resource contention.
    b. [4pts] Now introduce the highest-locker's protocol and show that the deadlock has been avoided.
    c. [4pts] A constraint of using mutex locks is that only the (current) owner of a mutex lock can unlock it. Discuss why this constraint is necessary based on the operation of the highest-locker's protocol.

Part B: Advanced Problems

14. Consider the Linux CFS scheduler based on *vruntime* (i.e., the virtual clock algorithm)
    a. [3pts] Let there be two processes P1 and P2. Suppose that we want P1 to receive twice as much CPU time as P2 does. Show your *vruntime* rates of P1 and P2 and justify your answer by showing the scheduling result. Note: you can define your own rule of tie-breaking.
    b. [3pts] Consider the use of CFS scheduler in real scenarios. Since I/O bound processes increase their *vruntime* slower than other processes, they always have smaller a *vruntime* and may overwhelm the CPU. Propose a solution to this problem.
15. [6pts] *iowait* of a process is the proportion of time that a process waits on I/O operations. A computer has 4 GB of RAM of which the operating system occupies 512 MB. The processes are all 256 MB (for simplicity) and have the same characteristics. Let the memory be fully occupied by processes. If the goal is 99% CPU utilization and higher, what is the process *iowait* that can be tolerated?
16. [3pts] What is the grace period of RCU (Read-Copy-Update)?

Total: 105 pts

For each question, you must sufficiently explain your answer to get full marks.

You are solely responsible for your answer. TAs only provide explanations with good intention.

1. Synchronous I/O will block the orginal process. Giving a high priority will make them finish fast and the original process can go on. It will improve the CPU utilization and reduce the turnaround of the process.
5

2. At first a hardware triggers an interrupt to PIC, then PIC sends it to CPU. CPU will look up the interrupt vector table to find the corresponding ISR. While transfering to ISR, the old process will push the current program counter and flag register to its own stack, then push other registers, and store stack pointer to its PCB. As for the new process (ISR) it will do the same thing in the reversed order: Read stack point into CPU, pop general register, and then pop PC and FLAG.
-2

+0 host kernel → real kernel
3. Both in the real user mode. In VM, only VMM runs in the real kernel mode to protect the system.

+2
4. a, b, d. They all need some privileged instructions such as access specific memory or create a new process. As for mmap(), it only access its own virtual memory.

5. a. Each function is a process. It's easy to add a new function. Also most of the functions are in user mode, giving more security to the system.
+1

b. The structure gives less cost for communication, which is essential for desktop computers since there are many process running on them.
+2

c. Embedded OS is usually small and simple. The programs on it are also tested, which already provides some kind of security. Moreover, the programs in embedded OS won't change often. According to the property of this kind of OS, it doesn't need to seperate user and kernel mode.
+2

6. running → waiting: wait(S), it will wait other processes signal S.
10
waiting → ready: While receiving signal, this process will be in ready status and wait for entering CPU.

ready → running: When it's time for this process, it will become running and run statement B.

7. ✓ a: stack ; b: stack ; *p : stack ; malloc()'ed space : heap ; x : data

8. Create multiple threads for a process and block one thread to see whether the remaining threads will also be blocked or keep going. If all threads are blocked, then it's M-1 model, otherwise it is 1-1 model.
7

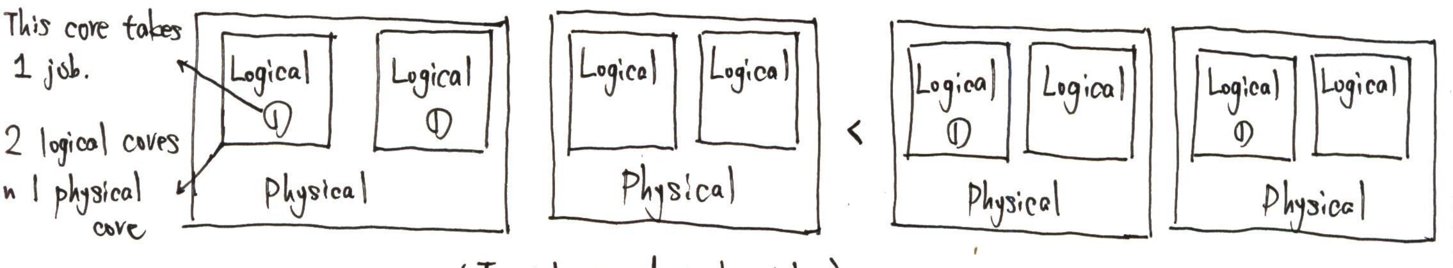
9. Because we hope the processes can do its own thing at the same time to get the advantage of parallization, each processes needs its own local variables to achieve this goal (otherwise the data of this process will be interrupted by other processes). That's why all the concurrent threads should have their own stack.
0
we are talking about "thread" but not process. ...

10. a. Advantage: Optimizing average waiting time.
+2
Disadvantange: Not fair, starvation for long execution time process.

b. In I/O scheduling, those processes which have short I/O time are usually a CPU-bound process. Doing those jobs first could send those processes back to CPU as fast as they can, which can utilize CPU in a more efficient way. Also, for those which spend much time in I/O, they are not that enmergency basically, so they can be processed later.
+3
e.g. tranfering file, backup.

11. a. Load balancing is to balance the loading of each processors, dispatch the job to each processors evenly. It can improve the utilization of the processors.
+5
Processor affinity means that the same job will be taken by the same processor. It will reduce the cost of transfering data in the processors' cache, which may take much time.

b. Multiple logical cores may correspond to a single physical core. If we only focus on logical cores, the below case would happen and waste the second physical core.
+3



(Two jobs need to be taken)

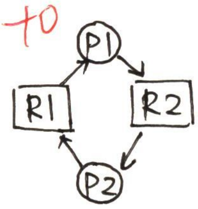
12.

5

```
int num_food=0
semaphore mutex_num_food=1
semaphore empty=0
/* cook code */
while(1){
    wait(empty)
    wait(mutex_num_food)
    putServingsInPot(M)
    num_food += M
    signal(mutex_num_food)    -3
}

/* villager code */
while(1){
    wait(mutex_num_food)
    if(num_food==0){
        signal(mutex_num_food)
        signal(empty)    -2
    }
    else{
        getServingFromPot()
        num_food--
        signal(mutex_num_food)
        eat()
    }
}
```

13. a. +0



1. P2 runs first and locks R2
2. P1 comes with higher priority, preempts CPU and locks R1, causing a deadlock.

b. +4
1. P2 runs first and locks R2
2. P2 increases its priority ✓
3. P1 comes but it cannot preempt CPU since P2 has the same priority as P1 ✓
4. P2 locks R1, finishes job, releases R1 and R2 ^ priority → original
5. P1 takes R1, R2 and does its job.

⇒ No deadlock.

c. The priority of processes in this protocol may change.

+3 If others can unlock a mutex, it will mess up the order of locking and lose the function of the mutex lock.

14. a. Let the clock of P1 run twice slower than P2's

+3 Suppose the clock of P1 runs 1 unit per ms and P2's runs 2 units per ms.

```
              P1 run       P1 run×2
P1 clock time: 0→1 →1 →2 →3 →3 →4 →5 →5 →6 →7 ...
P2 clock time: 0→0 →2 →2 →2 →4 →4 →4 →6 ...
                  P2 run
```

On average, P1 will receive twice time.

b.

+0

15. (4GB - 512MB) ÷ 256 MB = 3584 MB ÷ 256 MB = 14 processes running at the same time. +2

5

$1 - iowait^{14} > 0.99 \Rightarrow iowait^{14} < 0.1 \Rightarrow iowait < \sqrt[14]{0.1}$

≥ 1