上 海 交 通 大 学 试 卷（A 卷）

（2018 至 2019 学年　第 2 学期）

班级号_____ 学号_____ 姓名_____

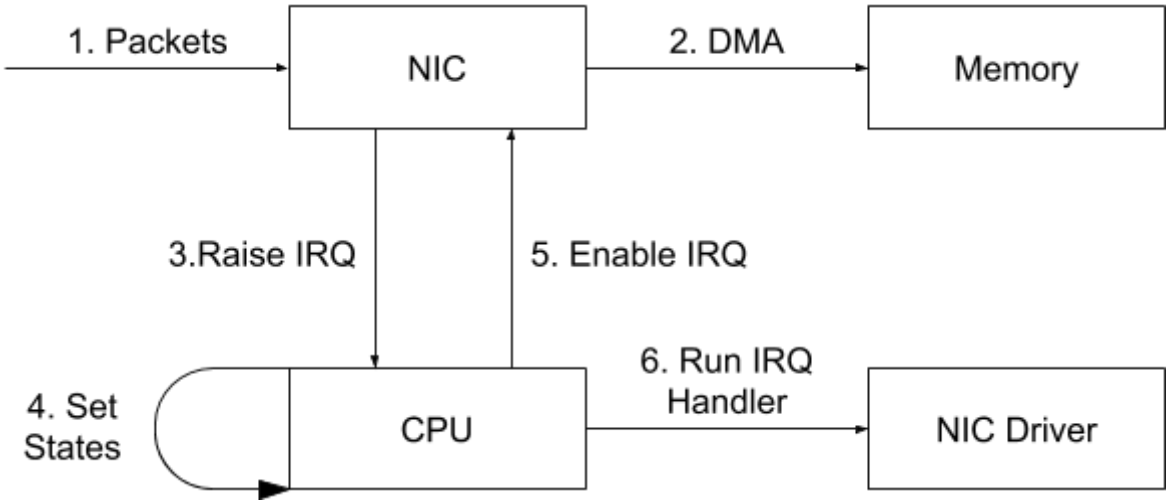
课程名称_____操作系统（期末）_____ 成绩_____

Problem-1: Interrupt (20')

1. Please briefly describe the similarities and differences between interrupts and exceptions, and give a few examples. (4')

Answer:
Similarity: Both alter the sequence of instructions executed by a processor
Difference: Interrupt is asynchronous, generated by hardware, can be delayed and cannot cause another interrupt; exception is synchronous, generated by software, cannot be delayed and can cause another exception.

2. Consider the following example about network packets processing. In this example, once a network packet arrives, the NIC (Network Interface Card) receives and writes the packet to a memory buffer through DMA. Next, the NIC sends an IRQ (Interrupt Request) to inform the CPU of the packet. The CPU then sets and stores some essential states, for example, disables the IRQ and sets the interrupt stack. Afterwards, the CPU enables the IRQ immediately and executes the NIC handler to process the packet. Assume that the NIC sends one IRQ per packet, and the CPU processes one packet per IRQ.



Suppose that the NIC bandwidth is 100 Gbps (1 Gbps = 1,000,000,000 bit/s) and each packet has the same size 128B (1 Byte = 8 bit). The latencies of step 3-6 are as follows. The CPU has **four** cores.

Step	3. Raise IRQ	4. Set States	5. Enable IRQ	6. Run IRQ Handler
Latency (ns)	10	5	10	75*

* 75 ns is the time that CPU needs to process a 128 B packet in the IRQ Handler.

(1) Why does the CPU enable the IRQ (step 5) right after setting the essential states (step 4) instead of enabling it after step 6? Please state the benefit. (4')

Answer:
Enabling IRQ after step 4 allows CPU to respond to NIC more timely, so the maximum throughput can be improved.

(2) Softirq is a basic design of the bottom half in Linux. Since instances of the same softirq can run on multiple cores simultaneously, the *re-entrant problem* must be considered. To protect the shared resources in softirq, which way in the following is correct? Please give your reasons. (4')

A

```
static bool _lock = false;

void softirq() {
    ...
    while (CAS(_lock, false, true)) {}
    operate_on_shared_resources();
    CAS(_lock, true, false);
    ...
}
```

B

```
static int _curTicket = 0;
static int _nowTicket = 0;

void softirq() {
    ...
    int myTicket = FAA(_nowTicket);
    while (myTicket != _curTicket) { sleep(1); }
    operate_on_shared_resources();
    _curTicket++;
    ...
}
```

(Tips: the function `CAS(var, old, new)` compares the current value of `var` with `old`, if they are equal then assigns `new` to `var`; the function `FAA(var)` adds 1 to `var` (fetch-and-add). Both functions return the original value of `var` and are atomic.)

(3) Given the parameters above, what is the maximum possible packet processing throughput for CPU with softirq? (4')

Answer:
Throughput = 128B / 100ns * 4 = 4.77 GB/s

(4) If the incoming packet rate is faster than the CPU throughput that you computed in (3), the CPU may be busy dealing with interrupts and the IRQ handler may be interrupted **frequently**, which degrades the overall performance. How could the interrupt mechanism in the example be modified to mitigate such a scenario? Please state your modification and compute the new throughput. (4')

Answer:
Do polling during interupts.
Throughput = 128B / 75ns * 4 = 6.36 GB/s

Problem-2: File System (20')

1. Please briefly describe the process of file creation in an ext3 file system and a FAT32 file system. (4')

Answer:
Ext3: Look up the inode and block map for available inode and block, then write data to the blocks and update the file attributes in the inode, next update the direct entry.
FAT32: Look up the FAT for available clusters, and write data to the clusters, next update the direct entry.

2. Crash recovery is a major concern in the file system. Please state three recovery approaches to file system failures. (4')

Answer:
1) Synchronous meta-data update + fsck
2) Logging
3) Soft update

3. Consider a new recovery approach: record an operation's name and parameter in the log file. For instance, for an operation that creates file "/d/f", the file system appends the transaction record [create "/d/f"] to the log. The file system ensures that the corresponding transaction record is written to the log before the modified disk blocks are flushed to disk. Upon crash and recovery, the file system re-executes the logged operations and deletes the log after that. Is this design correct? Can it recover a file system correctly from crashes? Please give your reasons. (4')

Answer:
Incorrect. Because the operation is not atomical, and the failure can happen in the middle of an operation when some instructions have been executed, after the

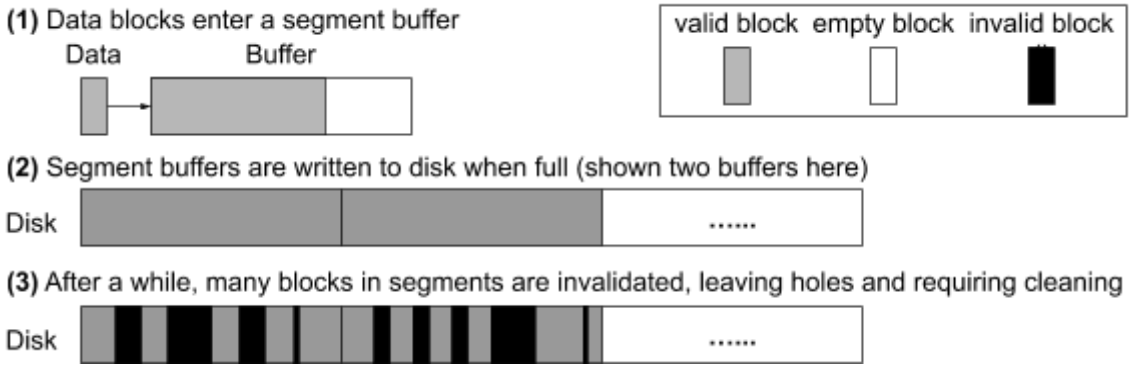
recovery, those instructions of an operation can be done twice, e.g., double decreasing the reference count in unlink.

4. LsFS (Log-structured File System) has been widely wielded in SSD (Solid-state Disk) firmware because LsFS suits SSD pretty well. Please explain how SSD can benefit from the design of LsFS. (4')

Answer:

1) Wear leveling
2) Garbage collection to avoid write-amplification

5. While LsFS provides good write performance for small files, its biggest problem is the high overhead of *log cleaning*. The following figure shows a typical writing process in an LsFS: Data are first written into a segment buffer to form a large log. When the segment buffer is full, the entire buffer is written to a disk segment in a single large disk write. When some of the files are modified, some previous blocks on the disk are invalidated correspondingly. These invalidated blocks become holes in disk segments and have to be cleaned.



Now, there are two observations: (1) the cleaning process will be more time-consuming if there are more invalid blocks, and (2) data blocks can be divided into *active* blocks, which are modified frequently, and *inactive* blocks, which are rarely modified.

Suppose that before a data block is written into the segment buffer, its type (active or inactive) can be accurately predicted. Please give an optimization of the above writing process to improve the log cleaning performance, and explain why your optimization will work. (4')

Answer:

Design two types of buffer: an active one and an inactive one, and write data to the corresponding buffer, so invalid block will be grouped and there is less “holes”.

Problem-3: Virtualization (20')

1. Why a fork bomb will crash a system? Why it cannot crash a (host) system when running in a virtual machine? (4')
   Answer:
   1) Too many processes will exhaust the computation resources (e.g., memory) within a system.
   2) A virtual machine is isolated from the host system. It is essentially a process of the host, whose resource has been confined at the very beginning.

2. Suppose there are 10 VMs, each VM is assigned with 10 VCPUs. How many threads in the host system (assume KVM+Qemu system, ignore the iothread of Qemu)? (4')
   Answer: 10 x 10 == 100

3. IOMMU is widely used to connect I/O devices and main memory. Could you explain why IOMMU is essential to device passthrough for VM? (4')
   Answer: Otherwise, when a VM tends to use DMA to access memory, target device does not know about the mapping from GPA to HPA, which will corrupt the memory.

4. If a device is assigned to a VM, which addresses does the device's IOMMU translate? (from what address to what address) (4')
   Answer: From GPA (device address, I/O address) to HPA.

5. Suppose the guest page table and extend page table translate addresses as below and all the pages are 4K-aligned. If a device needs direct memory access to physical address at 0x0 to 0x1fff, how to configure the IOMMU page table? Please fill in the blanks. (4')

Extended page table	
GPA	HPA
0x0	0x0
0x1000	0x1000
0x2000	0x2000
0x3000	0x3000

Guest page table	
GVA	GPA
0x0	0x2000
0x1000	0x3000

IOMMU page table	
Address before	Address after

Answer: 0x2000 -> 0x2000; 0x3000->0x3000

Problem-4: Microkernel (20')

1. Please list at least three types of kernel structures, give an example for each, and compare their pros and cons. (4')

Answer:

1) Monolithic kernel: Linux
2) Microkernel: L4
3) Exokernel: ExOS

2. Please compare the differences between the page fault handling process of a microkernel and monolithic kernel. (4')

Answer:

1) microkernel: userspace process triggers a page fault -> kernel -> userspace pager handles the page fault -> kernel -> return to process
2) monolithic kernel: process triggers a page fault -> kernel -> return to process

3. A web server usually interacts with I/O devices such as NIC and disk. Please compare the process of writing data to disk on a microkernel and monolithic kernel. (4')

Answer:

1) microkernel: userspace process write data -> kernel -> userspace driver -> device
2) monolithic kernel: userspace process write data -> kernel driver -> device

4. Inter-process Communication (IPC) by message passing is one of the central paradigms of most microkernel-based and other Client/Server architecture. seL4 (an L4-derived microkernel) only supports synchronous IPC, which avoids buffering in the kernel as well as copying cost associated with it.

Answer:

1) When the size of message fits into registers and a server thread is available, seL4 may choose an IPC *fastpath*, by a context switch that leaves the message registers untouched (i.e., client's thread executes in server's domain). Please explain the reason why fastpath has a good performance. (Hint: recap LRPC talked in class) (4')
No schedule overhead, no need to wait.

2) Synchronous IPC negatively impacts the performance of some intensive workloads. The overhead includes direct cost of mode switching and indirect pollution of important processor structures (e.g., cache, TLB). Suppose you are going to address the problem mentioned above in seL4, please propose a design. (Hint: multicore) (4')
Like flexSC, allocate a pool to temporarily store IPC requests. A process on core #0 sends a request and continue running (unless it needs to wait the result of the IPC). Some kernel threads on other cores (not #0) use polling to serve IPC and forward to the target process and write back the result.

Problem-5: Scalable Lock (20')

Caidodo designed a single-threaded key-value store server and wants to extend it into a multi-threaded version. He found that multiple requests could run in parallel and try to modify the same entry, so he decides to protect each key-value entry with a simple ticket lock. After several hardware upgradations, he found that the throughput did not scale linearly. Please help him to analyze the problem.

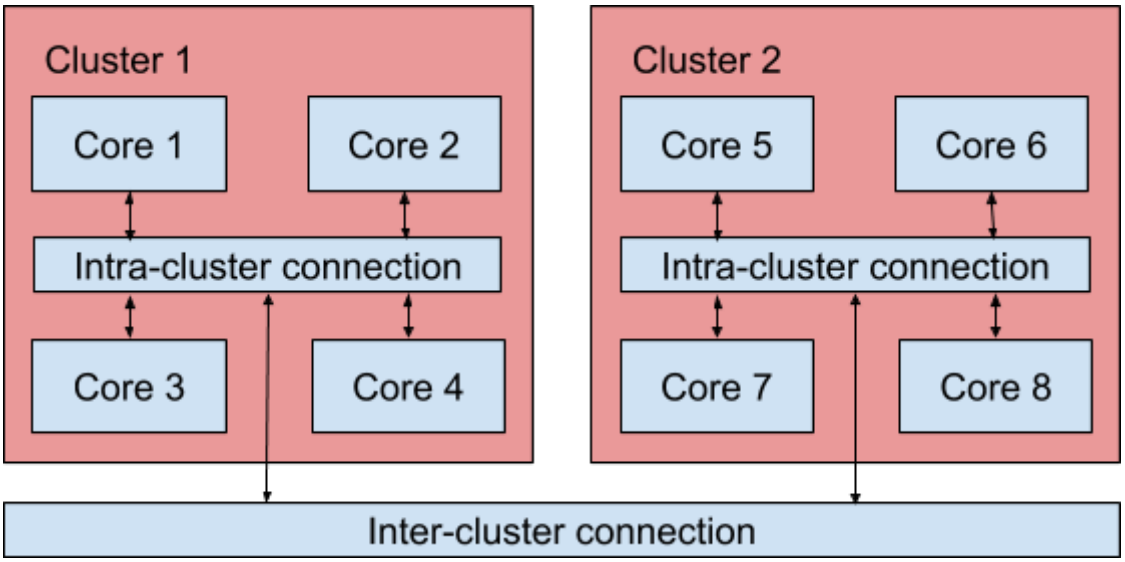
1. Why could not a ticket lock scale to many-core system? (4')

In ticket lock, all waiters are spinning on the same memory address. When the releaser wants to release the lock, it will modify the lock variable. This leads to a broadcast in cache coherence protocol to invalid all copies in other cores' private cache. After the releaser modified the variable, it will be synced to all other cores' private cache again. The cache coherence protocol will introduce overhead which grows linearly to the number of cores.

2. How does MCS lock solve the scalability problem? (4')

In MCS lock, the waiter spins on its own variable. The lock variables are not shared so no frequent cache coherence operation will occur. When a releaser wants to release the lock, it only modifies the variable in the next waiter, so cache coherence protocol is only performed between two cores, whose overhead is not related to the number of cores.

3. However, MCS lock performs badly on **multi-cluster system**. A **multi-cluster system** has a configuration like this:



In such a configuration, transferring a cache line between two cores (e.g. core 1 to core 5) in different cluster could incur more overhead compared to a transfer in the same cluster (e.g. core 1 to core 2). Please compare the performance of MCS lock under the following two schedulings (the number means on which core the thread gains lock), and think in which scheduling the MCS lock performs better? Why? (4')

1. Schedule A: 1 2 3 4 8 7 6 5
2. Schedule B: 1 5 2 6 3 7 4 8

Schedule A is better, because in schedule A only 1 inter-cluster transferring happens, while in schedule B 7 inter-cluster transferrings happen.

4. Caidodo found that most of the operations in his system is `read`, so he decides to further optimize his key-value store by using BR-lock. Why BR-lock performs better in a read-mostly system? (4') ((Hint: In BR-lock, the reader only needs to gain its local lock, so multiple readers could perform operations concurrently. The writer needs to gain locks of all threads/cores, so it could block all other readers and writers.))

In BR-lock, reader only needs to spin on its local lock, which means the cache line won't be transferred between two readers. So that in a read-mostly system there is few number of cache line transferring, and the system could scale to many-core setup. However, in write-mostly system, every writer must collect core-number times lock variable from other cores' cache, which means the overhead is linear to the number of cores.

5. Caidodo firstly used Algorithm A, and found that a writer may hold locks and wait for a lock owned by a long-running reader. This will hurt the performance of readers with lower thread id. He designs a new algorithm B, which is also a BR-lock, but in Algorithm B a writer will release all previous locks if it fails to gain one lock within limited time.

Algorithm A:	Algorithm B:

```
read_lock():
   spin_lock(&locks[get_thread_id()])

read_unlock():
   spin_unlock(&locks[get_thread_id()])

write_lock():
   for i := 0 to NTHREADS - 1:
      spin_lock(&locks[i])

write_unlock():
   for i := 0 to NTHREADS - 1:
      spin_unlock(&locks[i])
```

read_lock/read_unlock/write_unlock is the same as algorithm A

```
write_lock():
   for i := 0 to NTHREADS - 1:
      retry = 3 // try 3 times
      while (retry--):
         if (spin_trylock(&locks[i])):
            break // successfully locked
         sleep(1) // wait for 1 millisecond
      if (retry == -1):
         // fail to get lock i in 3ms
         // thread i is a long-running reader
         write_lock_slowpath(i)
         return

write_lock_slowpath(start):
   for j := 0 to start - 1:
      spin_unlock(&locks[j]) // release all
                             // previous locks
   // try again to gain all locks
   // but this time we wait for the long-running
   // thread first so it won't stuck other threads
   i = start
   do:
      spin_lock(&locks[i])
      i = (i + 1) % NTHREADS
   while i != start
```

Here is an example of difference between Algorithm A and B:

Assume we have 8 threads. Thread 3 wants to get a write lock, but thread 5 holds its read lock and is performing a long read-only critical section. In Algorithm A, thread 1-4 will all wait for thread 5 to finish, while in Algorithm B thread 3 will temporarily release lock 1-4 so thread 1,2,4 could do read operation in parallel with thread 5.

Compared to Algorithm A, Algorithm B: (multiple choice) (2')

A. Benefits reader, hurts writer
B. Benefits writer, hurts reader
C. Benefits both reader and writer

However, Algorithm B could not always work. It may lead to: (multiple choice) (1')

A. Dead lock; B. Live lock; C. Starvation

Why? You could give a scheduling example. (1')

5 read lock
3 write lock, failed and release 1-4, waiting on 5
2 read lock

1 write lock, failed and release 1, waiting on 2

2 read unlock

1 do the write lock slow path, lock 2, 3, 4, waiting on 5

5 read unlock

3 do the write lock slow path, lock 5, 6, 7, 8, 1 and waitng on 2

Now thread 1 held lock 2 and wait for lock 5, thread 3 held lock 5 and wait for lock 2, dead lock

我承诺，我将严格遵守考试纪律。

承诺人：_____

题号									
得分									
批阅人(流水阅卷教师签名处)									