上 海 交 通 大 学 试 卷（ A 卷）

（ 2016 至 2017 学年 第 2 学期 ）

班级号______________________ 学号______________ 姓名 ____________

课程名称____计算机系统基础（2）______________________成绩 _____________

Problem 1: CPU Scheduling (21 points)

1. [1] [2]

   [3] [4]

   [5] [6]

   [7] [8]

2. 1) [1] [2] [3]

   [4] [5]

   2)

我承诺，我将严格遵守考试纪律。

承诺人：________

题号	1	2	3	4	5				
得分									
批阅人(流水阅卷教师签名处)									

Problem 2: Address Translation (29 points)

1. [1] [2] [3] [4]

2. [1] [2] [3] [4]

   [5] [6] [7] [8]

   [9] [10] [11] [12]

   [13] [14] [15] [16]

3.

Problem 3: Memory Mapping (28 points)

1 [1] [2]

[3]

[4]

2

3 [1] [2]

[3] [4]

4

Problem 4: Concurrency (22 points)

1.

2.

1. [1] [2] [3]

   [4] [5]

2.

3.

Problem 1: CPU Scheduling (21 points)

1. The following table shows the information of four jobs. No I/O issues are involved.

Job	Arrival Time	Length of run-time
A	0ms	9ms
B	2ms	6ms
C	6ms	5ms
D	10ms	2ms

✧ The **RR** time-slice is **1ms**.

✧ Suppose when a job arrives, it is added to the tail of a work queue. The **RR** policy selects the next job of the current job in the queue.

Please calculate the average turnaround time and average response time for various scheduling policies. (8′)

Scheduling Policy	Average Turnaround Time	Average Response Time
FIFO	[1]	[2]
SJF	[3]	[4]
STJF	[5]	[6]
RR	[7]	[8]

2. Suppose we use **MLFQ** scheduling policy. (8′)

✧ There are 3 priority queues Q0, Q1, and Q2; Q2 has the **highest** priority, and Q0 has the **lowest** priority.

✧ **FIFO** is used in each queue.

✧ The CPU scheduling is carried out only at **completion** of processes or time-slices.

✧ The right table shows the **arrival time** of jobs in the workload.

Job	Arrival Time
A	0ms
B	7ms
C	15ms
D	19ms

Following table shows the execution of CPU. No I/O issues are involved.
NOTE: **X** represents an unknown time quantum

Time	0	X	3X	4X	6X	7X	8X	9X	10X	11X	13X	15X	19X
CPU	A	A	B	B	C	D	C	A	D	A	D	A	D

1) Please determine the following values. (All the answers are integers) (10′)
X: __[1]__ ms
Time-slices: Q2 = __[2]__ ms, Q1 = __[3]__ ms, Q0 = __[4]__ ms
Time between two priority boosting: __[5]__ ms

2) Based on the above execution, can you list an unwise parameter of this **MLFQ** scheduling policy, and explain why? (3′)

Problem 2: Address Translation (29 points)

This problem concerns the way virtual addresses are translated into physical addresses. Below are the specifications of the system on which the translation occurs:

- The main memory is byte addressable.
- The memory accesses are to **1-byte** words (not 4-byte words).
- The system uses a **two-level** page table.
- The page size is **512B**.
- PPN is **5 bits** wide.
- The number of PTEs in L1 page table is **equal** to that in L2 page table.
- The TLB is **2-way** set associative with **8** entries in total.
- TLBT is **4 bits** wide.
- The following figure shows the formats of the virtual addresses:

VPN-1	VPN-2	VPO

Virtual Address

1. Warm-up Questions (2' * 4 = 8')

The VPO bits	__[1]__
The VPN-1 bits	__[2]__
The virtual address bits	__[3]__
The physical address bits	__[4]__

The contents of TLB and the first 4 entries of the two-level page table are given below. And only some of the L2 page tables are shown. Note that the **ADDR** column of L1 page table means the base physical address of L2 page tables. Also, the base physical address (**ADDR**) is shown in the bottom of each L2 page table. All numbers are in hexadecimal.

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	3	14	1	1	01	1
1	8	11	1	0	09	0
2	2	0d	1	e	0f	1
3	f	13	0	8	16	1

TLB: 4 sets, 8 entries, 2-way set associative

VPN-1	ADDR	Valid
00	3800	1
01	3a00	1
02	2e00	0
03	3600	1
...	...	...

Part of L1 Page Table

VPN-2	PPN	Valid
00	0c	1
01	0d	1
02	0e	0
03	12	1
...	...	...

ADDR: 0x3c00

VPN-2	PPN	Valid
00	11	0
01	13	1
02	0f	1
03	03	1
...	...	...

ADDR: 0x3a00

VPN-2	PPN	Valid
00	04	1
01	0f	1
02	06	1
03	07	1
...	...	...

ADDR: 0x3800

VPN-2	PPN	Valid
00	05	0
01	14	1
02	0e	0
03	15	0
...	...	...

ADDR: 0X3600

Part of L2 Page Table: Only some of the L2 page tables are shown

2. Please translate virtual address to physical address and fill in the following blanks (If the value is unknown or meaningless, enter '--' for them) (1' * 16 = 16')

Parameter	Value
Virtual Address	0x47e0
VPN-1	0x__[1]__
VPN-2	0x__[2]__
TLB Index	0x__[3]__
TLB Tag	0x__[4]__
TLB Hit? (Y/N)	__[5]__
Page Fault? (Y/N)	__[6]__
PPN	0x__[7]__
Physical Address	0x__[8]__

Parameter	Value
Virtual Address	0x03cc
VPN-1	0x__[9]__
VPN-2	0x__[10]__
TLB Index	0x__[11]__
TLB Tag	0x__[12]__
TLB Hit? (Y/N)	__[13]__
Page Fault? (Y/N)	__[14]__
PPN	0x__[15]__
Physical Address	0x__[16]__

3. What is the register **CR3** used for? (2') Which type of addresses (**physical** address or **virtual** address) is stored in it? Please also explain why this type of address is used. (2') When will the value in **CR3** be changed? (give an example) (1')

Problem 3: Virtual Memory (28 points)

Suppose the program runs on a byte-addressable system which has **32-bit** wide virtual address and **two-level** page tables. Each L1 and L2 page table entry takes **4 bytes**. Each L1 page table and L2 page table has 1024 entries. The size of each page is **4 KB**. Please answer the following questions.

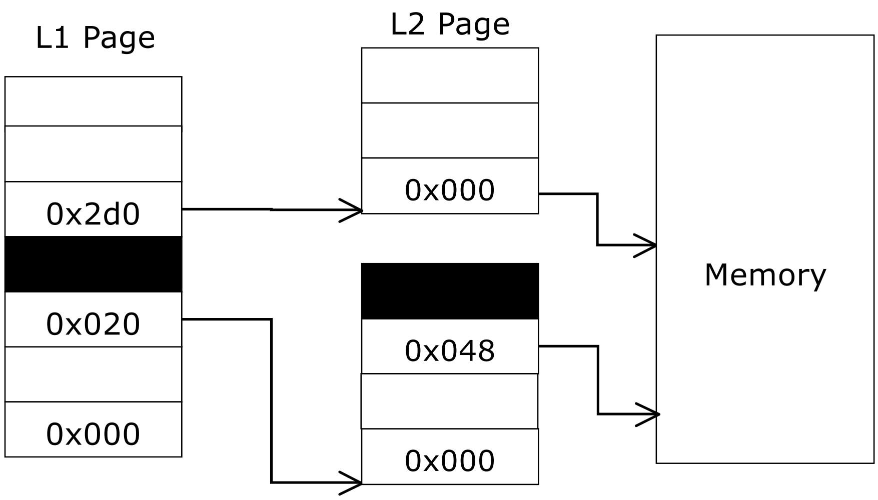
```
#define PAGE_SIZE (1<<12)
#define ARRAY_LEN (1024 * PAGE_SIZE)

int main(void) {
    int idx;
    int fd = open("ics.txt",O_RDWR);
A:
    char* sbuf = mmap(0, ARRAY_LEN,
                      PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    char* pbuf = mmap(0, ARRAY_LEN,
                      PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
B:
    for (idx = 0; idx < ARRAY_LEN; idx++)
        sbuf[idx] = pbuf[idx] + 1;
C:
    fork();
D:
    for (idx = 0; idx < ARRAY_LEN; idx++) {
        sbuf[idx] = pbuf[idx] + 1;
        pbuf[idx] = sbuf[idx] + 1;
    }
E:
    munmap(sbuf, ARRAY_LEN);
    munmap(pbuf, ARRAY_LEN);
    close(fd);
    return 0;
}
```

Assumption:

1) After setting **MAP_SHARED** flag, the updates to the shared area will be immediately synchronized to the file.
2) After setting **MAP_PRIVATE** flag, the changes made to the file after **mmap()** call and before copy-on-write are visible to the mapped area. The modification on the private area will cause a copy-on-write (**COW**) mapping.
3) The `ics.txt` is a file filled with a lot of character '0' and its size is more than `ARRAY_LEN`.

Before label **C**, the **virtual** and **physical** address of `pbuf` are `0xb7c00000` and `0xef400000` respectively, and the **virtual** address of `sbuf` is `0xb7800000`.

The following figure shows part of the page table when the program arrives at label **A**. The number within block is the index of page table. The white block without number means one or more empty page table entries. Please answer the following questions (**NOTE:** please ignore the page fault on the **stack** for the following questions)



1. Please fill in the following blanks. (2'*4 = 8')
   1) Bit range [31:0] is the total bit range of a virtual address.

      Bit range ___[1]___ of a virtual address represents the index in L1 page table.

      Bit range ___[2]___ of a virtual address represents the offset within a page (VPO).
   2) How many L1 and L2 page table entries are needed **totally** in the given virtual memory system to express an array whose virtual address range is **`[0, 0xffffff]`**. **HINT**: more than one L2 page table is needed.

      L1 page table entries: ___[3]___ total L2 page table entries: ___[4]___

2. How many **page fault** exceptions are raised by code between label **B** and **C**? Note: Each page fault will fill in both the L1 page table entry (if necessary) and the L2 page table entry. (2') How many of them will handle **COW**? (2') Please also write down your explanation. (2')

3. Please write down the addresses of below variables of the **child** process when the program reaches label **D**. Write '**`DK`**' if it is unknown. (2'*4=8')

   `pbuf[0]`: virtual address = ___[1]____ physical address = ___[2]____

   `pbuf[2048]`: virtual address = ___[3]____ physical address = ___[4]____

4. Suppose that both **`pbuf`** and **`sbuf`** are configured to use **huge** pages and the size of each huge page is **4MB**. Please draw a graph like above to show the page table of the process when the program reaches label **E**. (6') NOTE: you can use a **black** block to represent the consecutive filled page table entries and use an **empty** block to represent the consecutive unfilled page table entries. For example, you can draw (0x1) (black block) (0xA) to represent the consecutive filled entries from 0x1 to 0xA

Problem 4: Concurrency (22 points)

TAs find using normal **`test-and-set (TAS)`** is not a good approach, because **`TAS`** instruction is quite expensive (as compared to a normal load from memory). Thus, they instead use this code to implement a lock, so-called "**`double-test-and-set`**".

```
typedef struct __lock_t { int flag; } lock_t; // init to 0

void lock(lock_t *lock) {
  do {
    while (lock->flag) // unprotected lock check
      ; // spin
  } while (TAS(&lock->flag, 1)); // actual atomic locking
}

void unlock(lock_t *lock) { lock->flag = 0; }
```

1. Does this lock work correctly? Why or why not? (3')
2. When does it perform better than a simple spinlock built with **`test-and-set`**? Why? Hints: Consider the number of threads acquiring the same lock. (3')

To avoid spin, we use **`park()`** and **`unpark()`** to refine the implementation of lock. (NOTE: the following C code is identical to the C code shown in class)

```c
typedef struct __lock_t {
  int flag;
  int guard;
  queue_t *q;
} lock_t;
```

```c
void lock_init (lock_t *lock) {
  lock->flag  = 0;
  lock->guard = 0;
  queue_init(lock->q);
}
```

```c
void lock (lock_t *lock) {
  while (TAS(&lock->guard, 1) == 1) ;        // L1
  if (lock->flag == 0) {                     // L2
    lock->flag = 1; // lock is acquired      // L3
    lock->guard = 0;                         // L4
  } else {
    queue_add(lock->q, gettid());            // L5
    setpark();                               // L6
    lock->guard = 0;                         // L7
    park();                                  // L8
  }
}
```

```c
void unlock (lock_t *lock) {
  while(TAS(&lock->guard, 1) == 1);          // U1
  if (queue_empty(lock->q)) {                // U2
    // no one wants it
    lock->flag = 0;                          // U3
  } else {
    // hold lock (for next thread!)
    unpark(queue_remove(lock->q));           // U4
  }
  lock->guard = 0;                           // U5
}
```

Suppose that there are **one uniprocessor** and **two threads** (**T** and **S**). The CPU scheduler runs **T** and **S** alternately, and each line of above C code (labeled by number **`L1-L8`** and **`U1-U5`**) runs **atomically**. The execution flow of **T** and **S** scheduled on the uniprocessor is shown by a sequence of "**`t`**" and "**`s`**" (each "**`t`**" or "**`s`**" represent one line of C code was executed by the thread T and S). For example, the sequence "**`tttss`**" means that **T** runs 3 lines of code and then **S** runs 2 lines of code.

The initial state of the shared lock is **NOT** held, and two threads T and S call **`lock()`** at the same time to acquire the same shared lock. Suppose the execution flow of T and S is "**`lock()->critical path->unlock()`**" and the number of code lines in critical path is more than 3.

1. Fill in the following blanks using `L1-L8` or `U1-U5`. (2'*5=10')
   Suppose the current execution sequence is "**`t`**" (NOTE: it means that the CPU scheduler runs 1 line of code in **`lock()`** by **T**). Which line of code in **`lock()`** will be executed by **T** when it is scheduled **again**? ___**`L2`**___
   1) Then, the CPU scheduler further runs "**`tt`**" (NOTE: the current full execution sequence is "**`ttt`**"). Which line of code in **`lock()`** will be executed by **T** when it is scheduled **again**? ___[1]___
   2) Then, the CPU scheduler further runs "**`sss`**" (NOTE: the current full execution sequence is "**`tttsss`**"). Which line of code in **`lock()`** will be executed by **S** when it is scheduled **again**? ___[2]___
   3) Then, the CPU scheduler further runs "**`tttsss`**" (NOTE: the current full execution sequence is "**`tttsssstttsss`**"). Which line of code in **`lock()`** will be executed by **S** when it is scheduled **again**? ___[3]___
   4) Then, suppose **S** is waiting on the lock. Which line of code in **`lock()`** makes **S** wait for the lock? ___[4]___. Meanwhile, **T** has completed its critical section and starts to call **`unlock()`** to release the lock. Then, the CPU scheduler runs "**`tt`**". Which line of code in **`unlock()`** will be executed by **T** when it is scheduled **again**? ___[5]___

2. If the line **`L6`** in **`lock()`** is deleted, the modified code exists a **race-condition** bug. Please explain how does it happen? (3')
3. Please list at least two purposes of the **`lock->guard`** variable? (3')