上 海 交 通 大 学 试 卷（ A 卷）

（ 2017 至 2018 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

1. [1] [2]

   [3] [4]

   [5] [6]

2. 1)

   2)

Problem 2: Replacement Policy

1. [1]

   [2]

   [3]

   [4]

   [5]

2. [6]

   [7]

   [8]

   [9]

   [10]

3.

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 3: Address Translation

1 [1] [2]

[3] [4]

[5] [6]

2 [1] [2] [3] [4] [5]

[6] [7] [8] [9] [10]

Problem 4: Concurrency

1

2

Problem 5: Memory Mapping

1.

2.

3.

4

Problem 6: Lock

1

2

3

4

Problem 1: Scheduling (20 points)

1. We have following jobs in the workload. No I/O issues are involved.

Job	Arrival Time	Run time
A	0ms	4ms
B	1ms	1ms
C	4ms	5ms
D	6ms	2ms

   - ✧ When a job arrives, it is added to the tail of the work queue.
   - ✧ CPU picks job to run after all queue operations.
   - ✧ The **MLFQ** policy has 2 priority queues, higher one with time-slice of 1ms and lower one with time-slice of 2ms. We use **RR** in each queue. Priority boost isn't supported.
   - ✧ No preemption in **MLFQ**.
   - ✧ We do RR by moving the **recently executed task** to the end of the queue.
   - ✧ The priority of operations is RR movement > accepting new job.

   Please calculate the **average** response time and **average** turnaround time for different scheduling policies. (2' * 6)

Scheduling Policy	Turnaround Time	Response time
FIFO	[1]	[2]
STCF	[3]	[4]
MLFQ	[5]	[6]

2. We decide to use **MQMS** on a machine with CPU 0 and 1. Each CPU has a scheduling queue.
   - ✧ There are no I/O issues involved.
   - ✧ Each queue uses RR policy with time-slice of 1ms.
   - ✧ We do RR by moving the **recently executed task** to the end of the queue.
   - ✧ New job will be added to the tail of the queue with less jobs. (If equal, to queue 0)
   - ✧ During work stealing, each queue peek at another. If that queue has more than 2 jobs, it will steal the last job and put it at the end of itself.
   - ✧ The priority of operations is RR movement > accepting new job.
   - ✧ CPU picks job to run after all queue operations.

   Given the execution of CPUs, time 0 means the task running during [0ms,1ms)

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
CPU0	A	A	B	A	B	C	A	B	A	A	A	A	A	F
CPU1	D	E	D	E	D	E	D	E	C	F	C	F	G	C

   1) Please tell the arrival time of job A, B, C. (2' * 3)
   2) What's the frequency of work stealing? (2')

Problem 2: Replacement Policy (18 points)

Assume we have a primary device with 3 physical blocks, please complete the following questions.

1. Suppose we are using LRU replacement policy, please complete the following table. (**NOTE:** you do not need to consider the order of primary device contents) (1.5' * 5)

Reference String	1	2	3	4	2	1	3	1	4
Primary Device Contents	1	1 2	1 2 3	2 3 4	[1]	[2]	[3]	[4]	[5]

2. Suppose we are using clock algorithm with rules:
   - ✧ The clock pointer points to position 0 initially.
   - ✧ We do not reset the clock pointer after we have found an evict page, so next time we start from the position after previous victim.
   - ✧ Each page brought in were set as accessed initially.

   Please complete the following table. (**NOTE:** use "*" to represent current clock pointer) (1.5' * 5).

Reference String	1	2	3	4	2	1	3	1	4
Primary Device Contents	1 *	1 2 *	1* 2 3	4 2* 3	[6]	[7]	[8]	[9]	[10]

3. Why do we use clock algorithm instead of directly implement LRU policy in most realistic systems? Please give your reason. (3')

Problem 3: Address Translation (21 points)

Assume we have a machine with the following specifications:

- ✧ The memory is byte-addressable
- ✧ 64KB physical memory space
- ✧ 1MB virtual memory space
- ✧ Each page is 256B
- ✧ The size of one page table equals to the size of page
- ✧ length of each PTE is 16B
- ✧ 8 entries, 2-way associative TLB
- ✧ LRU replacement policy in TLB
- ✧ Each L1 cache line is 4B
- ✧ 8 entries, 2-way associative L1 cache

1. Please fill the following table. (1' * 6)

The VPO bits	[1]
The VPN-1 bits	[2]
The TLB tag bits	[3]
The number of PTE in one page table	[4]
The number of page table level	[5]
The maximum size of the whole page table	[6]

2. Given the following page table contents and cache/TLB state, finish the following address translation. (1.5' * 10)
   NOTE: Accesses are independent, which means they won't affect the TLB and cache state in the next access.

VPN	Addr	Valid
0	9600	1
...	...	...
f	1e00	1

Part of L1 page table

VPN	Addr	Valid
d	b900	1
f	1e00	1

PT @0x9600

VPN	Addr	Valid
5	d800	1
6	bc00	1

PT @0xb900

VPN	Addr	Valid
0	ac00	1
1	ad00	1
...	...	...
d	b900	0
...	...	...
f	bb00	1

PT @0x1e00

Set	Valid	Tag	PPN	Valid	Tag	PPN
0	1	00b3	bc	1	03f4	d9
1	0	0035	d8	1	02ea	35
2	1	0398	a2	0	021f	3e
3	0	03d7	d1	1	0171	3d

TLB state

Set	Valid	Tag	bytes	Valid	Tag	bytes
0	0	3bd	...	1	63d	...
1	1	bc3	...	1	d93	...
2	1	274	...	0	bd6	...
3	0	d50	...	1	d80	...

cache state

Parameter	Value
Virtual Address	0xd50c
TLB Hit? (Y/N)	[1]
Number of Memory Accesses to Page Table	[2]
Page Fault? (Y/N)	[3]
Physical Address	[4]
Cache Hit? (Y/N)	[5]

Parameter	Value
Virtual Address	0xfd635
TLB Hit? (Y/N)	[6]
Number of Memory Accesses to Page Table	[7]
Page Fault? (Y/N)	[8]
Physical Address	[9]
Cache Hit? (Y/N)	[10]

Problem 4: Concurrency (12 points)

Deadlock is a problem in concurrent programs. Please consider the below execution flow.

Initially: a=1, b=1, c=1		
Thread	Thread 1	Thread 2
Step1	P(A)	P(B)
Step2	P(C)	P(A)
Step3	P(B)	V(B)
Step4	V(B)	P(C)
Step5	V(A)	V(C)
Step6	V(C)	V(A)

1. Does it cause deadlock? (2’) Please draw progress graph and **explain why** base on the graph. (6’)

2. If we change the order of **two steps with operation P** in Thread 2, the deadlock will be erased. What are the two steps? (4’)

Problem 5: Memory Mapping (16 points)

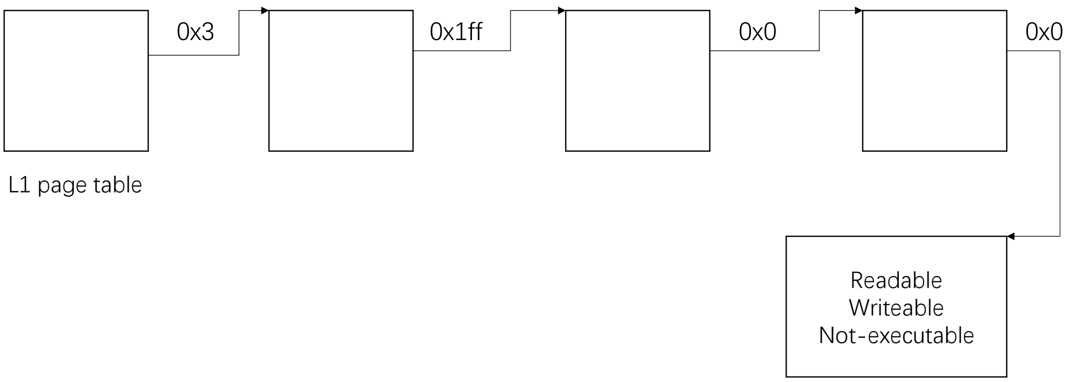
```
... // include headers
int main() {
    int fd1 = open("a.txt", O_RDWR);
    int fd2 = open("b.txt", O_RDWR);
    char *a = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd1, 0);
    a[0] = 1;

    char *b = mmap(NULL, 4096, PROT_READ | PROT_WRITE |
                   PROT_EXEC, MAP_PRIVATE, fd2, 0);

    b[0]++;

    if (fork() == 0) {
        b[0]++;
        a[0] = 2;
    }

    printf("a[0] = %d\n", a[0]);
    printf("b[0] = %d\n", b[0]);
    return 0;
}
```

Assumption:

- ✧ The program runs on an Intel **`x86_64`** Linux system.
- ✧ Before the program executes, a.txt and b.txt are both 4096B long and filled with zero.
- ✧ After **`fork()`**, the child process is always executed first.
- ✧ No preemption, so only after the child process exits, the parent process will run.
- ✧ **`mmap()`** in line 5 returns **`0x1ffc0000000`**, and before line 5 the L3/L4 page table of this address doesn't exist.
- ✧ **`mmap()`** in line 9 returns **`0x1ffc0200000`**.

1. Given the page table of the process after line 8, please draw the page table of **CHILD** process before it is executed. (6')

   NOTE: You should mark the permissions of the physical page as the given figure does.



2. Show **ALL** the code location where the private CoW page is copied. (2')

3. Please give the output of the program. (4')

4. After all the processes exit, what's the content of file a.txt and b.txt. (4')

Problem 6: Lock (13 points)

```
typedef struct __node_t {
  __node_t* next;
  tid_t tid;
} node_t;

typedef struct __lock_t {
      node_t* head;
} lock_t;

void lock_init(lock_t* lock) {
      lock->head = NULL;
}
/* each caller should alloc and hold a node by itself */
void lock(lock_t* lock, node_t* node) {
      if (lock == NULL || node == NULL) {
          /* output error message... */
          exit(-1);
      }
      node->next = NULL;
      node->tid = gettid();
      node_t* old = test_and_set(&lock->head, node);
      if (old == NULL) return;
      old->next = node;
      park();
}

void unlock(lock_t* lock, node_t* node) {
      if (lock == NULL || node == NULL) {
          /* output error message... */
          exit(-1);
      }
      if (node->next == NULL) {
          if (compare_and_swap(_[1]_, _[2]_, _[3]_)) {
              return;
          }
          while(node->next == NULL)
              continue;
      }
      unpark(node->next->tid);
}
```

Above code is an implementation of a kind of lock called MCS. Each **`lock_t`** structure represents a lock. Each thread who wants a lock will maintain a **`node_t`** structure. Nodes are connected through their arriving order. The head of lock will always point to the newest node. **`Test_and_set`** and **`compare_and_swap`** operations are **atomic**. **DO NOT** consider the CPU may reorder the instructions.

1. **`compare_and_swap(a, b, c)`** will set a's value to c and **return 1** if and only if a's old value equals b, otherwise **return 0 and do nothing**. Fill the blanks in the code. **Hint:** consider the situation that another thread acquires the same lock between line 32 and 33. Your code is used to keep the lock running correctly under this situation. (3')

2. Why do we need line **36** and **37**? (2')

3. There are two problems in this implementation. Figure them out and try to fix them **if possible**. **Hint:** Consider about **correctness** and **security**. (4')

4. Analyze the lock about its **fairness** and **performance**. (4')