上　海　交　通　大　学　试　卷（ A 卷）

（ 2015　至 2016　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: Address Translation (22 points)

1. [1]　　[2]　　[3]　　[4]

2. [1]　　[2]　　[3]　　[4]

   [5]　　[6]　　[7]

   [8]　　[9]　　[10]　　[11]

   [12]　　[13]　　[14]

Problem 2: Deadlock (12 points)

1.

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 3: Concurrency (14 points)

1.	[1]	[2]	[3]
	[4]	[5]	[6]
	[7]	[8]	[9]

2.

Problem 4: Networking (18 points)

[1]

[2]

Problem 5: Virtual Memory (24 points)

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

2

3

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

5

Problem 6: Concurrent Hash Table (10 points)

1. [1]

Problem 1: Address Translation (22 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 **one-level** page table.
✧ Virtual addresses are **25 bits** wide.
✧ PPN is **9 bits** wide.
✧ The amount of PTE is $\mathbf{2^{14}}$.
✧ The TLB is **4-way** set associative with **16** total entries.

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

The VPO bits	__[1]__
The physical address bits	__[2]__
The page size	__[3]__
The number of bits for TLBT	__[4]__

The contents of the TLB and first 16 entries of the page table are given below. All numbers are in hexadecimal. According to the illustration and answer the questions. Please fill in the blanks.

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	0CD	09	1	AA1	00	1
	3E0	62	0	C4C	48	1
1	312	45	0	010	75	1
	987	3A	1	D39	3F	0
2	038	E3	0	0A7	13	1
	18B	52	1	49B	11	0
3	6C0	42	0	075	50	0
	013	39	1	0F2	0D	0

TLB: 4 sets, 16 entries, 4-way set associative

VPN	PPN	Valid	VPN	PPN	Valid
00	39	1	08	OD	0
01	52	1	09	45	0
02	E3	0	0A	13	1
03	00	1	0B	3A	1
04	11	0	0C	09	1
05	50	0	0D	42	0
06	62	0	0E	3F	0
07	75	1	0F	48	1

Page Table: Only the first 16 PTEs are shown

2. Please translate virtual address to physical address and get the data from cache if hit (otherwise entering '--') (1′ * 14 = 14′)

Parameter	Value
Virtual Address	0x014F2D3
VPN	0x__[1]__
TLB Index:	0x__[2]__
TLB Tag:	0x__[3]__
TLB Hit? (Y/N)	__[4]__
Page Fault? (Y/N)	__[5]__
PPN	0x__[6]__
Physical Address	0x__[7]__

Parameter	Value
Virtual Address	0x07C01A5
VPN	0x__[8]__
TLB Index:	0x__[9]__
TLB Tag:	0x__[10]__
TLB Hit? (Y/N)	__[11]__
Page Fault? (Y/N)	__[12]__
PPN	0x__[13]__
Physical Address	0x__[14]__

Problem 2: Deadlock (12 points)

1. Please consider the following executing flow that will cause **deadlock.** You need provide at least **TWO** possible cases. (You just need to answer the states of involved threads. e.g. After T1 finishes step1 and T2 finishes step2) (6′)

Initially: a=1, b=1, c=1			
Thread	Thread 1	Thread 2	Thread 3
step1	P(a)	P(b)	P(b)
step2	P(b)	P(c)	V(b)
step3	V(a)	V(c)	P(c)
step4	P(c)	P(a)	P(a)
step5	V(b)	V(b)	V(c)
step6	V(c)	V(a)	V(a)

2. If we remove **Thread 3**, then will it cause **deadlock**? Please draw **progress graph** and explain your reason. (6′)

Problem 3: Concurrency (14 points)

```
void *do0(void *arg) {
  int i;
  for(i=0; i<M; i++) {
    __[1]__
    printf("+3");
    __[2]__
  }
}

void *do1(void *arg) {
  int i;
  for(i=0; i<2*M; i++) {
    __[3]__
    printf("+2");
    __[4]__
  }
}

void *do2(void *arg) {
  int i;
  for(i=0; i<M; i++) {
    __[5]__
    printf("=7\n");
    __[6]__
  }
}
```

```
#include "csapp.h"
#define N   3
#define M   100   // M >> N

sem_t a, b, c;
int main(void) {
  void *(*threads[3])(void *)= {do0, do1, do2}
  pthread_t tid[N];
  int i

  sem_init(&a, 0, __[7]__);
  sem_init(&b, 0, __[8]__);
  sem_init(&c, 0, __[9]__);
  for(i=0; i<N; i++) {
    pthread_create(&tid[i], NULL,
                   threads[i%3], NULL);
  }

  for(i=0; i<N; i++) {
    pthread_join(tid[i], NULL);
  }

  /* destroy sem_t */
  ...
  return 0;
}
```

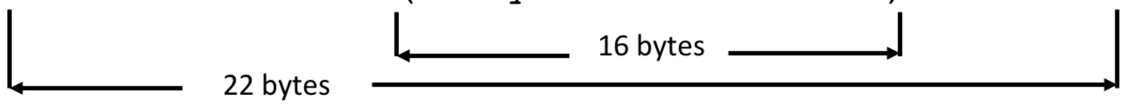
1. Suppose **N** is 3, please fill in the blanks to make above program print right equations (e.g. +2+3+2=7\n+2+2+3=7\n...). (**NOTE**: You can **ONLY** fill in several P(x) and V(x) operations in [1]-[6]) (9')
2. When **N** is an arbitrary multiple of 3 (3<N<MAXINT, e.g. N=6, or N=9), the program can't run properly. Please explain the reason and propose a solution so that the program will print right equations again. (**NOTE**: You can add **more** semaphores) (5')

Problem 4: Networking (18 points)

We have designed a simple file downloading protocol. The client sends out a request to the server by starting a connection and writing the file name. It ends the request by writing two characters '\r' '\n'.

Instead of raw file content, the server sends back a serial of **chunks**. A chunk contains a small part of file content. Here is the layout of a chunk with **16-byte** file content, while the chunk itself is encoded in a serial of 22 bytes:

```
‘1’ ‘6’ ‘\r’ ‘\n’ (16 bytes file content) ‘\r’ ‘\n’
```



A chunk begins with several ASCII characters ranging from '0'(0x30) to '9'(0x39), which imply the size of file content embedded in the chunk. Between the size string and actual file content, there are two characters '\r' '\n'. Another group of '\r' '\n' can be found at the end of this chunk.

A second chunk will follow the last byte of a first chunk. The server will end the chunk serial by a chunk with zero-byte file content, which is exactly a string "0\r\n\r\n". Assume there is no zero-sized file on server. If the first chunk in the serial has zero-byte file data, it indicates the requested file doesn't exist on server.

The client should restore the file by merging **only chunk file content in received order** from all chunks.

Examples:

```
1] Command: ./downloader www.file.com 7890 foo.txt
   Client => Server, sends 9 bytes: "foo.txt\r\n"
   Server => Client, sends 91 bytes in 3 chunks:  '6' '4' '\r' '\n' (64 bytes)
   '\r' '\n' '1' '0' '\r' '\n' (10 bytes) '\r' '\n' '0' '\r' '\n' '\r' '\n'
   Client saves 74 bytes in foo.txt

2] Command: ./downloader www.file.com 7890 bar.txt
   Client => Server, sends 9 bytes: "bar.txt\r\n"
   Server => Client, sends 5 bytes in 1 chunk: '0' '\r' '\n' '\r' '\n'
   Client prints "File bar.txt not found on server!"
```

Please complete the following client-side code and focus on functionality. You don't need to check any error return value. If the requested file exists on server, the client saves the merged chunk contents into a local file, which has been open as **`filefd`**. Otherwise the client should call **`report_file_not_found()`**. You don't have to concern the empty file created locally on this failed request.

Hint: you can use **`atoi()`** to transform a string like **`"64"` `or even` `"64\r\n"`** to an **`int`** value **`64`**. **`atoi()`** ignores non-digit characters at the tail of the string.

```
#include "csapp.h"

/* call this if the client gets only one zero-sized chunk. */
void report_file_not_found(const char *fname) {
   fprintf(stderr, "File %s not found on server!\n", fname);
}

/* the program is executed by ./downloader <hostname> <port> <filename> */
int main(int argc, char **argv) {
   int port = atoi(argv[2]);

   /* connect to <hostname>:<port> and send out the request */
   [1] // Write your code here (6')

   int filefd = Open(argv[3], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);

   /* read the chunks from server */
   /* write all chunk file content to file <filename> */
   [2] // Write your code here (12')

   Close(filefd);
   return 0;
}
```

Problem 5: Virtual Memory (24 points)

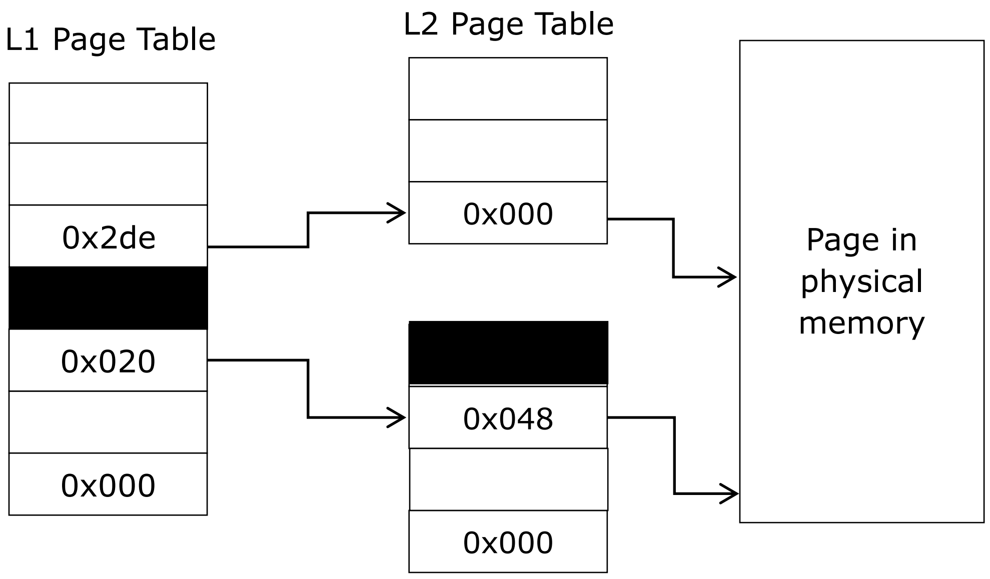
Suppose the program runs on the Pentium/Linux Memory System discussed in section 9.7 of the CSAPP. Please answer the following questions.

```
#include <unistd.h>
#include <sys/mman.h>
#include <stdio.h>

#define ARRAY_LEN (125*1024)

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] = sbuf[idx] + 1;
        pbuf[idx] = pbuf[idx] + 1;
    }
C:
    fork();
D:
    for (idx = 0; idx < ARRAY_LEN; idx++) {
        pbuf[idx] = pbuf[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 mapping 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 mapped area memory 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**.

The address of **`sbuf`** is `0xb7be2000` and that of **`pbuf`** is `0xb7cd4000` before label **B**. 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. (4′)
   The size of array **`pbuf`** is __**[1]**__ bytes
   The address range of array pointed by **`sbuf`** is between `0xb7be2000` and __**[2]**__
   The address range of array pointed by **`pbuf`** is between `0xb7cd4000` and __**[3]**__
   The index of L1 PTE for **`pbuf`** is __**[4]**__
2. How many page fault exception are raised by code between label **B** and **C**? (1′) How many of them will handle **COW**? (1′) Please also write down your explanation.(2′)
3. How many page fault exception are raised by code between label **D** and **E** of the parent process? (1′) How many of them will handle **COW**? (1′) Please also write down your explanation.(2′)
4. Please write down the value of below variables of process when the program reach label **C**.(4′)

   `sbuf[0]`=____[1]_____　　**`sbuf[1]`**=____[2]____

   `pbuf[0]`=____[3]_____　　**`pbuf[1]`**=____[4]____
5. Please draw a graph like above to show the page table of the process when the program reach label **E**. **Note** that you can use a **black** block to represent the consecutive filled PDE/PTEs. And use an **empty** block to represent the consecutive unfilled PDE/PTES. (For example, you can draw (0x1) (black block) (0xA) to represent the consecutive filled entries from 0x1 to 0xA) (8′)

Problem 6: Concurrent Hash Table (17 points)

Rehashing is to rearrange all the elements in an existing hash table into a new one with different bucket number. Consider the code below. A group of threads are working on rehashing the elements of **hashtb_old** into **hashtb_new**. The concurrent hash table is implemented with a configurable bucket number, which is different than the textbook. **list_t**, **List_Init()** and **List_Insert()** are exactly the same implementation as the concurrent linked list in class. (Note that **List_Insert()** is thread-safe)

```
typedef struct __hash_t {
   list_t *lists;
   int bucket_num;  // a new field than the code introduced in class
} hash_t;

void Hash_Init(hash_t *H, int bucket) {
  H->lists = (list_t *)malloc(bucket * sizeof(list_t));
  for (int i = 0; i < bucket; i++)
    List_Init(&H->lists[i]);
  H->bucket_num = bucket;
}

void Hash_Insert(hash_t *H, int key) {
  /* use modulo operation as the hash function */
  List_Insert(&H->lists[key % H->bucket_num], key);
}

/* add 1 to *ptr and return old *ptr atomically */
extern int FetchAndAdd(int *ptr);

hash_t hashtb_old, hashtb_new;

void *rehash(void *args) {
     /* the next bucket in hashtb_old to be rehashed */
     static int next_bucket = 0;

     /* update next_bucket atomically */
     /* rehash elements of the bucket into hashtb_new */
     /* go on to a second bucket until all buckets are rehashed */
     [1] // Write your code here (10')

  return NULL;
}
```

```
#define BUCKET_NUM_OLD 101
#define BUCKET_NUM_NEW 211
#define THREAD_NUM 8
#define DATA_NUM 100000000

int main(void) {
   /* initialize hash tables */
   Hash_Init(&hashtb_old, BUCKET_NUM_OLD);
   Hash_Init(&hashtb_new, BUCKET_NUM_NEW);

   /* insert some data */
   for (int i = 0; i < DATA_NUM; i++)
      Hash_Insert(&hashtb_old, i * 7 + 1);

   /* rehash concurrently */
   pthread_t tid[THREAD_NUM];
   for (int i = 0; i < THREAD_NUM; i++)
      pthread_create(&tid[i], NULL, rehash, NULL);
   for (int i = 0; i < THREAD_NUM; i++)
      pthread_join(tid[i], NULL);

   return 0;
}
```

1. Each thread should find a unique bucket of **`hashtb_old`** and rehash its elements into **`hashtb_new`**. Then it should move on to another unique bucket until all elements of **`hashtb_old`** are rehashed. Please complete **`rehash()`** in blank [1]. (10′)

   NOTE:

   1) You cannot declare any static local variable in your code.

   2) You don't have to free the memory of **`hashtb_old`**.

   3) You can call **`FetchAndAdd()`**, which has been introduced in class.