上 海 交 通 大 学 试 卷（A 卷）

（ 2020 至 2021 学年 第 2 学期 ）

班级号_____ 学号_____ 姓名 _____

课程名称 计算机系统基础（组成） 成绩 _____

Problem 1: Cache

1.

2. [1] [2] [3]

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

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

[9] [10]

4.

5.

6.

Problem 2: Linking

1.

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

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

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

我承诺，我将严格遵守考试纪律。

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

3. [1] [2]

[3] [4] [5]

[6] [7] [8]

[9] [10]

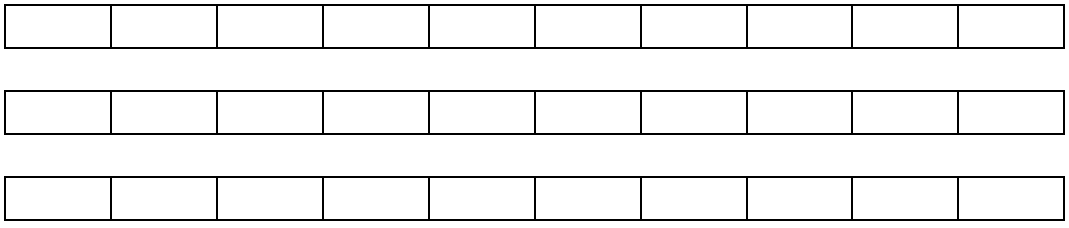
4. [1]

[2] [3]

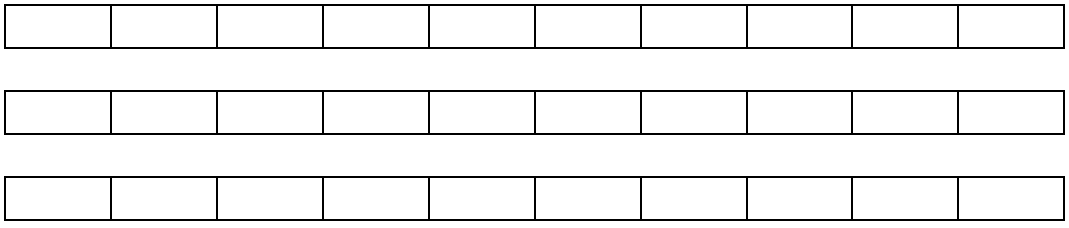
[4] [5]

Problem 3: Memory Allocation

1.



2.



3.

Problem 4: Address Translation

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

2.

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

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

4.

Problem 5: Memory Mapping

1.

2.

3. [1] [2]

[3]

4.

5.

Problem1: Cache (27 points)

Consider a **12-bit machine** with a **2-way set** associative cache. The size of each block is **4 bytes**, and the total number of sets is 4. **LRU** policy is used for eviction and **write back** policy is used. The following table shows the content of the data cache at time T.

Set	Tag	Valid	Byte0	Byte1	Byte2	Byte3	Tag	Valid	Byte0	Byte1	Byte2	Byte3
0	0x08	1	0xAC	0xAB	0x08	0x60	0x6A	1	0x11	0x1F	0x64	0x04
1	0xFE	1	0x56	0x01	0x00	0xCC	0x92	0	0xCA	0xBA	0x9F	0xF0
2	0x08	0	0x4C	0xAB	0xFF	0xFF	--	--	--	--	--	--
3	--	--	--	--	--	--	0x92	1	0x93	0x10	0xFD	0xBB

1. What is the total size of the cache? _____ (3')
2. How would a 12-bit physical memory address be split into tag/set-index/block-offset fields in this machine? (6')

   tag _[1]_ bits, set-index _[2]_ bits, and block-offset _[3]_ bits
3. A short program will read memory in the following sequences starting from time T. Each access will read **one byte**. Please fill the following blanks. If there is a cache miss, fill '-' for 'Byte Returned'. (10')

Order	Address	Hit/Miss	Byte Returned
1	0x081	[1]	[2]
2	0x92C	[3]	[4]
3	0x680	[5]	[6]
4	0xFE7	[7]	[8]
5	0x6A0	[9]	[10]

One of the TA write a program and test it on this cache. The size of short type is **2 bytes**. The cache is **empty** before the execution of the program. Please only consider **data cache** access and ignore instruction cache access. Other processes will not interfere status of memory and cache during this execution.

```
1.  short data1[4][2];
2.  short data2[2][4];
3.
4.  void total() {
5.     int i, j;
6.
7.     for (i = 0; i < 2; i++) {
8.        for (j = 0; j < 4; j++) {
9.           data1[j][i] = data2[i][j] * i;
10.       }
11.    }
12. }
```

Assume the address of data1[0] is 0x00. The address of data2[0] is 0x40. Answer the following questions:

4. Please calculate the cache miss rate. (2’)
5. Is there a way to optimize the cache miss rate if we **only** reorder some part of the code? (2’)
6. We can modify the cache through one of the following ways:
   a) Double the number of sets while keeping the cache size unchanged.
   b) Double the block size while keeping the cache size unchanged.
   c) Increase the cache size via doubling the number of cache lines in each set.

   Please choose the option(s) above that can optimize the miss rate of the **original program** (2’)**, and explain why** (2’).

Problem2: Linking (19 points)

The following program consists of two source files: **main.c** and **foo.c**. The relocatable object files are also listed. (Suppose do linking on an x86-64 little-endian machine).
NOTE: On an x86-64 machine, **sizeof(int)=4** and **sizeof(long)=8.**

/*main.c*/

```
1.  #include <stdio.h>
2.  int a = 6, b, c = 2;
3.  extern int array[];
4.  int foo(void);
5.  int main() {
6.      b = foo();
7.      printf("%d %d\n", a, array[b]);
8.      return 0;
9.  }
```

```
.text:
0000000000000000 <main>:
   0:   48 83 ec 08              sub    $0x8,%rsp
   4:   e8 00 00 00 00           callq  9 <main+0x9>
   9:   89 05 00 00 00 00        mov    %eax,0x0(%rip)
   f:   48 98                    cltq
  11:   48 8d 15 00 00 00 00     lea    0x0(%rip),%rdx
  18:   8b 14 82                 mov    (%rdx,%rax,4),%edx
  1b:   8b 35 00 00 00 00        mov    0x0(%rip),%esi
  21:   48 8d 3d 00 00 00 00     lea    0x0(%rip),%rdi
  28:   b8 00 00 00 00           mov    $0x0,%eax
  2d:   e8 00 00 00 00           callq  32 <main+0x32>
  32:   b8 00 00 00 00           mov    $0x0,%eax
  37:   48 83 c4 08              add    $0x8,%rsp
  3b:   c3                       retq
```

/*foo.c*/

```
1.  static int a = 10;
2.  int c;
3.  int array[6] = {4, 8, 12, 0};
4.  int foo() {
5.      for (int i = 0; i < 4; i++) {
6.          if (array[i] > a)
7.              return i;
8.      }
9.      return 0;
10. }
```

1. What is the output of the program? (3’)

2. For symbols that are defined and referenced in **main.o**, please complete the symbol tables. The format of them are same as ones in section 7.5 of our book. (6')

Module	Name	Value(Hex)	Size	Type	Bind	Ndx
main.o	foo	00000000	0	NOTYPE	GLOBAL	UND
	b	00000004	[1]	[2]	GLOBAL	[3]
	array	[4]	[5]	[6]	GLOBAL	[7]
	c	00000000	4	[8]	GLOBAL	[9]
	a	[10]	[11]	[12]	GLOBAL	.data

3. Fill in the relocation entries of the **.text** section of **main.o**. (5')

Relocation entries of **main.o**:

Module	Offset	Type	Symbol Name	Addend
main.o	0000000b	R_X86_64_PC32	[1]	[2]
	00000014	[3]	[4]	[5]
	0000001d	[6]	[7]	[8]
	00000024	[9]	.LC0	[10]

4. Assume we compile foo.c into shared library with -fpic and -shared options. Fill in the **GOT content** in the table below before execution. (5’)

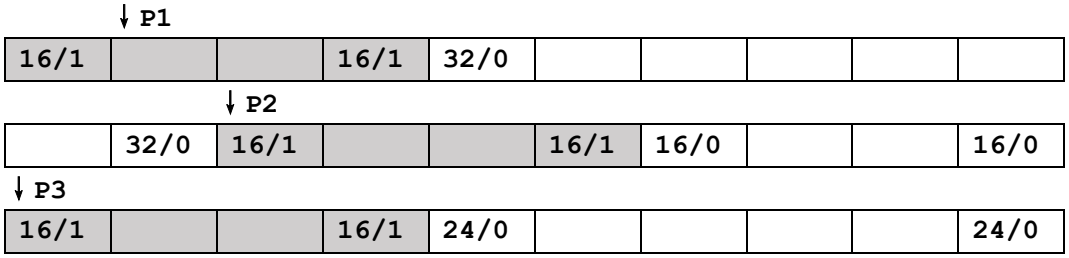
Address	Entry	Content	Description
0x200fc8	GOT[3]	0x0676	PLT entry for printf
0x200fd0	GOT[4]	[1]	PLT entry for foo

Fill in the **GOT content** in the table below after the execution of the given lines of main.c: (**Note**: You can use **addr(x)** to represent the absolute address of function x)

Line	GOT Entry	Content
6	GOT[3]	[2]
	GOT[4]	[3]
7	GOT[3]	[4]
	GOT[4]	[5]

Problem 3: Memory Allocation (14 points)

There are some **unallocated memory** and **heap**, as shown below. Heap is organized as a sequence of **contiguous** allocated and free blocks. **Allocated** blocks are shaded, and **free** blocks are blank (each block represents 1 word = 4 bytes). **Headers** and **footers** are labeled with the number of bytes and allocated bit. The allocator maintains **double-word** alignment. You are given the execution sequence of memory allocation operations (i.e., **`malloc()`** or **`free()`**) from 1 to 6.




```
1. P4 = malloc(4)
2. free(P1)
3. P5 = malloc(9)
4. P6 = malloc(5)
5. free(P3)
6. P7 = malloc(2)
```

Please answer the questions below. Assume that **immediate coalescing** strategy and **splitting free blocks** are employed.

1. Assume **first-fit** algorithm is used to find free blocks. Please draw the **final** status of memory and mark with block size in headers and footers after the operation sequence is executed (5').
2. Assume **best-fit** algorithm is used to find free blocks. Please draw the **final** status of memory and mark with block size in headers and footers after the operation sequence is executed (5').
3. Please calculate the total bytes of **internal fragments** (Note: **DO NOT** consider **`P1`**, **`P2`** and **`P3`** for internal fragments) (4').

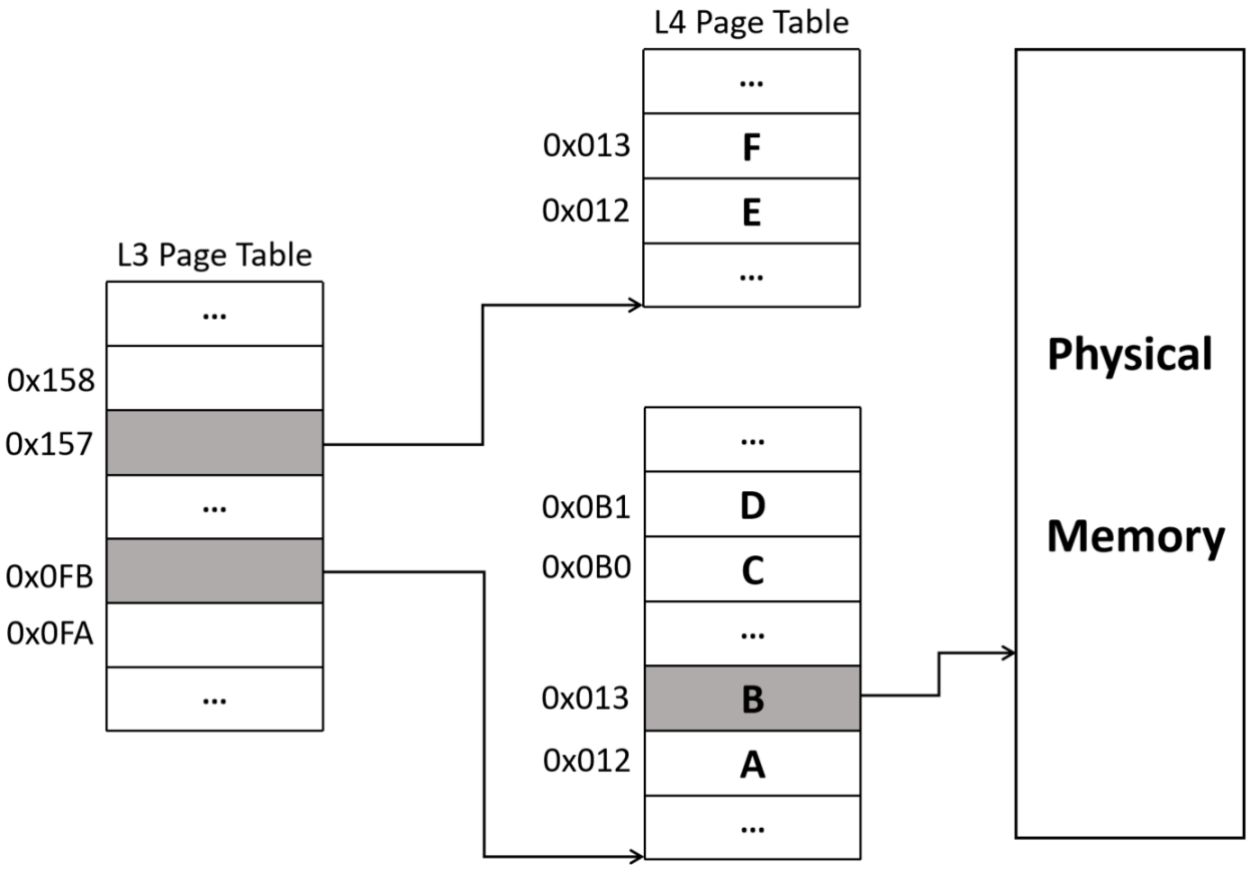
Problem 4: Address Translation (22 points)

Suppose we are using the Intel Core i7 memory system (Refer to **section 9.7** in CSAPP) and have a program showed below:

```
1.  #define ARRAY_LEN 500 * 4 * 1024
2.  int x;
3.  int y;
4.  char p[ARRAY_LEN];
5.  void main (void) {
6.      x = 5;
7.      y = x;
8.      for (int i = 0; i < ARRAY_LEN; i++) {
9.          p[i] = '0';
10.     }
11. }
```

Assumptions:

- All three variables **x**, **y** and **p** share the same L3 Page Table, showed in figure below.
- At the **beginning** of the program, part of the memory page tables is:

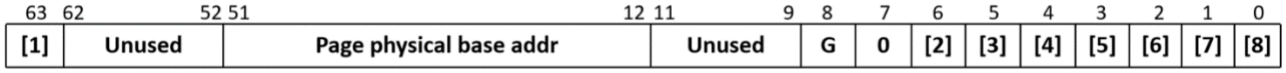


- ✧ **Grey** PTE entry's child page is **present**; **White** PTE entry's child page is **not present**
- ✧ **x**'s virtual address is **0x9F6128F0**, **p**'s virtual address is **0x9F400000**
- ✧ The program runs on an Intel **x86_64** Linux. The page size is **4kB**. We **don't** use **2MB** and **1GB** pages.

1. Please fill each part of **x**'s virtual address. (Refer to **Figure 9.22**) (8')

   VPN1 = 0x_[1]_, VPN2 = 0x_[2]_, VPO = 0x_[3]_, L1 TLB Tag=0x_[4]_

2. When we execute **Line 6**, please **determine whether** the page fault will happen. If it happens, please specify which page table entry (**A-F**) will be filled. (3')
3. Suppose variable **y**'s virtual address is in **PTE entry B**'s range, please fill the blanks in **PTE entry B** at the **end** of the program. (8')
   - Use **X/not-X** for [1].
   - Use **U/S** for [6].
   - Use **R/W** for [7].
   - Use **0/1** for the rest of fields or "**---**" if the value of field is uncertain.



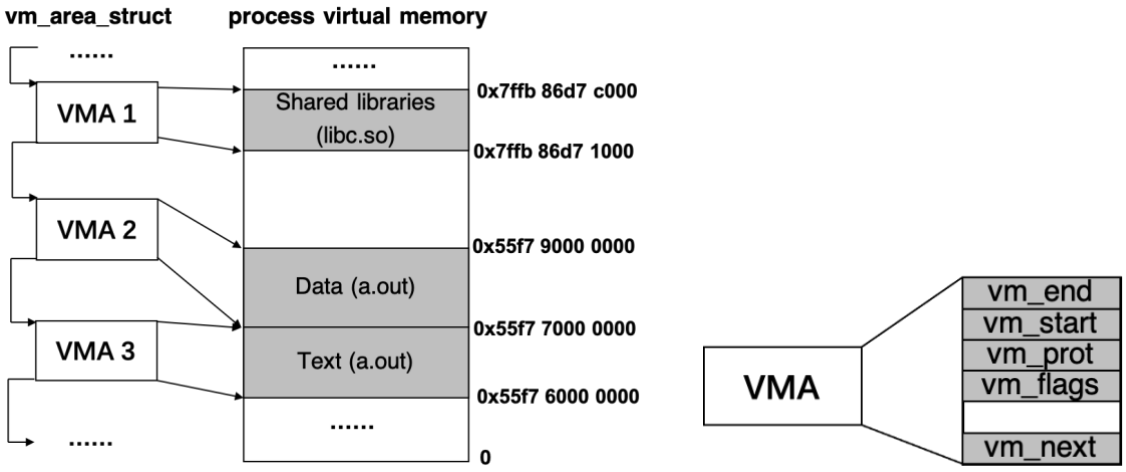
4. Assume there is no swapped-out page table. After the program ends, **how many physical pages** will be allocated due to the access of **p**? (Note: You need to consider both pages for **data** and for **page tables**) (3')

Problem 5: Memory Mapping (18 points)

The followings are parts of codes for program **a.out**.

```
1.  ... // include headers
2.  #define BUF_SIZE (2 * 1024 * 1024)  // 2MB
3.  int main() {
4.      int fd = open("ics.txt", O_RDWR);
5.  A:
6.      char *pbuf = mmap(0, BUF_SIZE,
7.              PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
8.      char *sbuf = mmap(0, BUF_SIZE,
9.              PROT_READ | PROT_WRITE | PROT_EXEC, MAP_SHARED, fd, 0);
10.     char *rbuf = mmap(0, BUF_SIZE,
11.             PROT_READ, MAP_PRIVATE, fd, 0);
12.     char c;
13. B:
14.     for (int i = 0; i < BUF_SIZE; i++) {
15.         sbuf[i] = '1';
16.         c = pbuf[i];
17.         pbuf[i] = c + '1';
18.     }
19. C:
20.     c = rbuf[0];
21. D:
22.     munmap(pbuf, BUF_SIZE);
23.     munmap(sbuf, BUF_SIZE);
24.     munmap(rbuf, BUF_SIZE);
25.     close(fd);
26.     return 0;
27. }
```

The following figure at left shows parts of **vm_area_structs(VMA)** layout when the program arrives at **Label A**. For each VMA, the figure also shows which file(**libc.so**, **a.out**) it belongs to, start address and end address. The figure at right shows the fields in **vm_area_struct,** as described in Textbook **Section 9.7.2**.



Assumptions:

- The program runs on an Intel **x86_64** Linux. The page size is **4kB**. We **DO NOT** use **2MB** and **1GB** pages.
- Sizes of file ics.txt is larger than **BUF_SIZE**.
- After setting **MAP_SHARED** flag, updates to the shared mapping area will be immediately synchronized to the file.
- 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 will cause a copy-on-write (COW) mapping.
- The **pbuf** value at **Line 6** is **0x6fff c000 0000**. The **sbuf** value at **Line 8** is **0x6fff c020 0000**. The **rbuf** value at **Line 10** is **0x6fff c040 0000.**
- The page table entries(**PTE**) in last level page tables for **pbuf, sbuf** and **rbuf** is **zero** when program arrives at **Label B**.
- You should only consider the page faults caused by the **sbuf, pbuf** and **rbuf**, and don't need to consider page faults caused by accessing page table pages.
- For **vm_prot** field, we only consider **PROT_NONE, PROT_READ, PROT_WRITE** and **PROT_EXEC**.

1. What is the advantage of using **mmap()**? (2')
2. The **mmap()** between **Label A** and **Label B** will affect the layout of **vm_area_structs**. Please draw the layout of **vm_area_structs** when program arrives at **Label B**. For each **vm_area_struct**, you should show which file it belongs to, their start address and end address. (4')

3. Please show the **vm_prot** in its **vm_area_struct** when program arrives at **Label B**. (**Examples: vma1**'s vm_prot: PROT_READ | PRTO_EXEC, **vma2**'s vm_prot: PROT_READ | PROT_WRITE) (6')

   pbuf's vm_prot: ___[1]___ **sbuf**'s vm_prot: ___[2]___

   rbuf's vm_prot: ___[3]___

4. Will the **vm_prot** you filled in problem 3 change between **Label B** and **Label D**? Why? (2')

5. How many page fault execptions will be raised between **Label B** and **Label C.** How many of them will handle **COW**? Please also write down your explanation. (4')