COMP2310/6310 Final Exam Review

Course Topics

- C to x86_64
- Processes and Signals
- Locality and Cache Memories
- Disk storage
- Linking
- Virtual Memory
- I/O
- Networking
- Concurrent programming

Exam

- **What to study?**
  - Chapters posted on the course website
- **How to Study?**
  - Read each chapter many times, work practice problems in the book and do problems from course website.
  - Practice problems allows you to get a feel for the questions on the the exam

Topics for Today

- C to x86_64
- Virtual Memory
- I/O Redirection
- Threading
- Processes and Signals
- Deadlock
- Hyperthreading
- Sequential consistency

- **Note: other topics will appear on the final exam!**

C to x86_64

- The following C code declares a structure. The declaration embeds one structure within another, just as arrays can be part of structures, and we can have arrays within arrays (e.g., two-dimensional arrays). The procedure on the left operates on the `comp2310` structure. We have intentionally omitted some expressions.

```
struct comp2310 {
    short *p;
    struct {
        short x;
        short y;
    } s;
    struct comp2310 *next;
};
```

```
void init(struct comp2310 *cp) {
    cp->s.y = ;
    cp->p = ;
    cp->next = ;
}
```

- What are the offsets (in bytes) of the following fields?
  - p
  - s.x
  - s.y
  - next
- How many total bytes does the structure require?

C to x86_64

- The compiler generates the following code for `init`

```
# void init(struct comp2310 *cp)
# cp in %rdi
init:
  movl 8(%rdi), %eax
  movl %eax, 10(%rdi)
  leaq 10(%rdi), %rax
  movq %rax, (%rdi)
  movq %rdi, 12(%rdi)
  ret
```

- Fill in the missing expressions in the C code for init based on this information.

Assembly Loops

- Recognize common assembly instructions
- Know the uses of all registers in 64 bit systems
- Understand how different control flow is turned into assembly
  - For, while, do, if-else, switch, etc
- Be <u>very</u> comfortable with pointers and dereferencing
  - The use of parens in mov commands.
    - %rax vs. (%rax)
  - The options for memory addressing modes:
    - R(Rb, Ri, S)
  - lea vs. mov

Array Access

- A suggested method for these problems:
  - Start with the C code
  - Then look at the assembly Work backwards!
  - Understand how in assembly, a logical 2D array is implement as a 1D array, using the width of the array as a multiplier for access

[0][0] = [0]	[0][1] = [1]	[0][2] = [2]	[0][3] = [3]
[1][0] = [4]	[1][1] = [5]	[1][2] = [6]	[1][3] = [7]
[2][0] = [8]	[2][1] = [9]	[2][2] = [10]	[2][3] = [11]

[0][2] = 0 * 4 + 2 = 2

[1][3] = 1 * 4 + 3 = 7

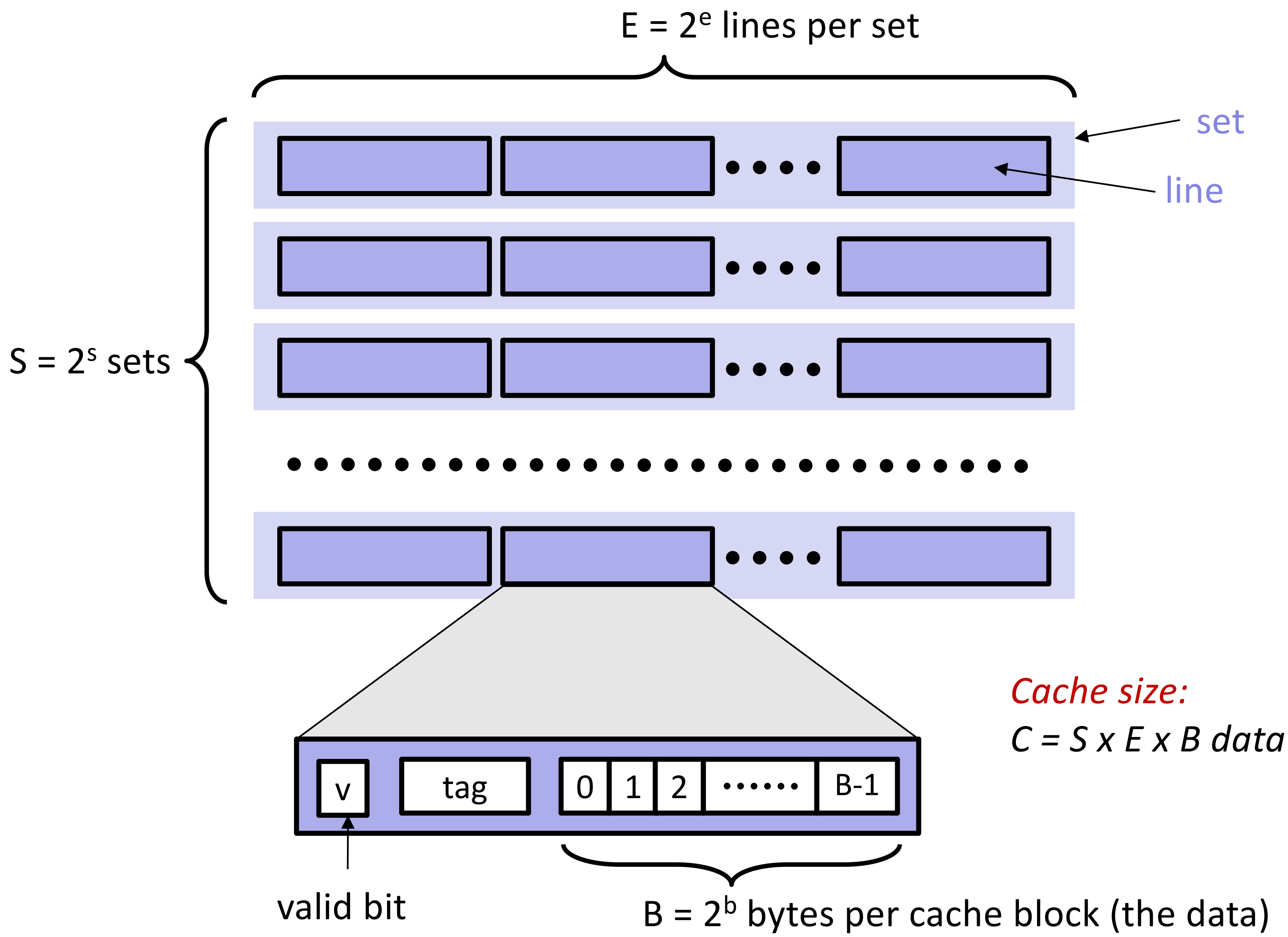
[2][1] = 2 * 4 + 1 = 9

[i][j] = i * width of array + j

Caching Concepts

- **Dimensions: S, E, B**
  - S: Number of sets
  - E: Associativity – number of lines per set
  - B: Block size – number of bytes per block (1 block per line)
- **Given Values for S,E,B,m**
  - Find which address maps to which set
  - Is it a Hit/Miss? Is there an eviction?
  - Hit rate/Miss rate
- **Types of misses**
  - Which types can be avoided?
  - What cache parameters affect types/number of misses?
- **Understanding of Locality**
- **Week 5, part 2**

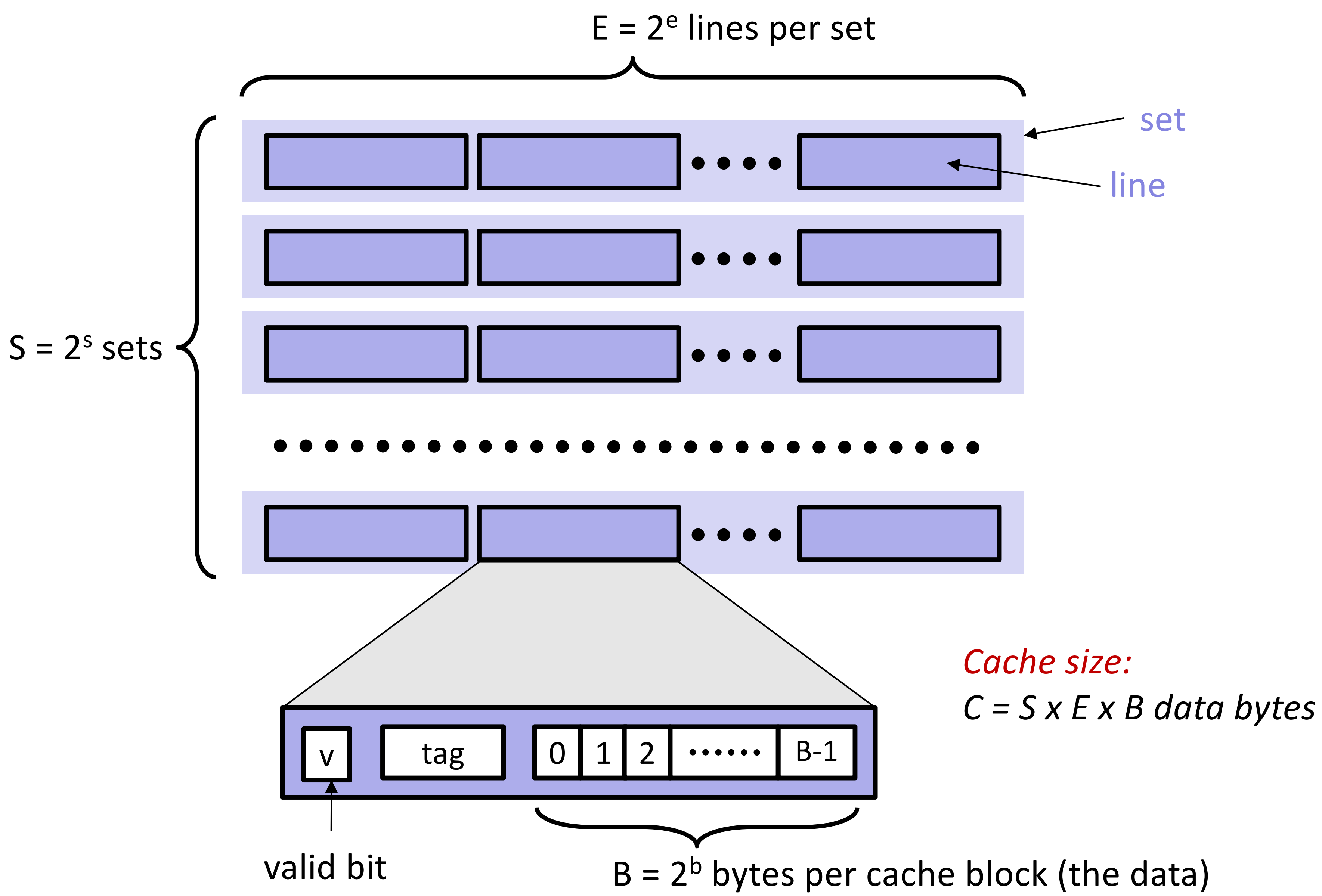
General Cache Organization (S, E, B)



Cache size:

C = S x E x B data bytes

General Cache Organization (S, E, B)



Cache size:

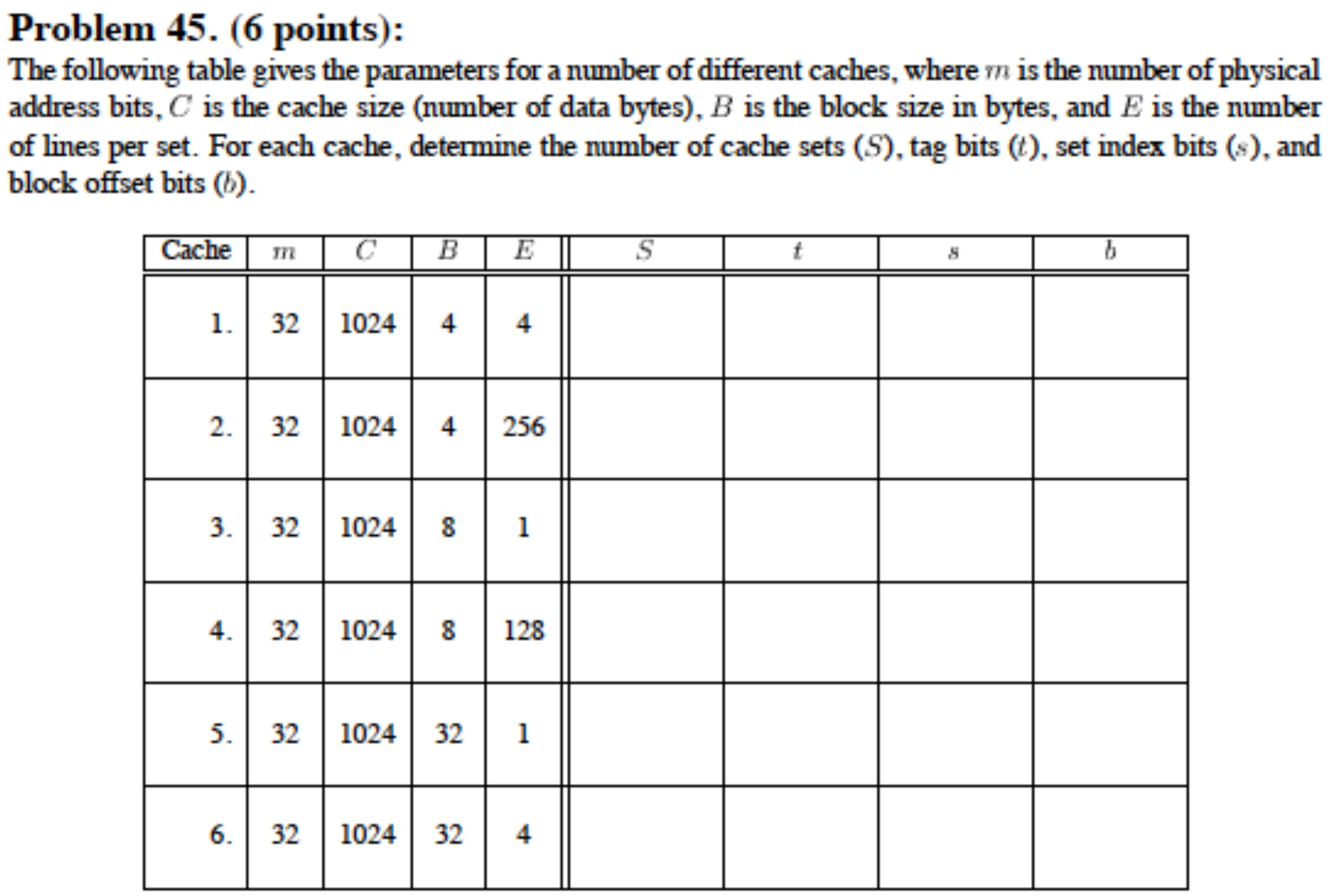
C = S x E x B data bytes

Locality Example

- **Question: Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?**

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

Caching



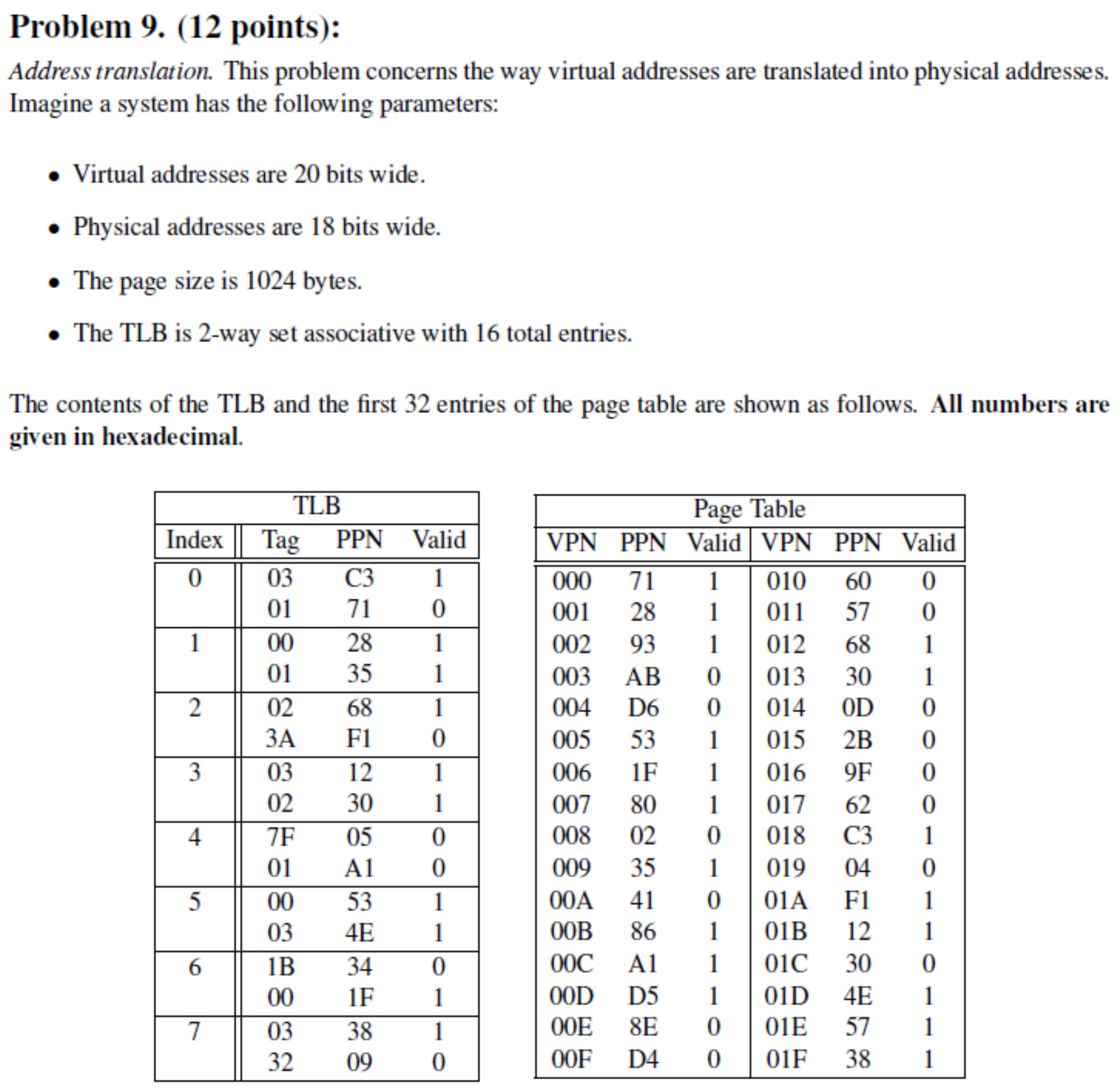
Problem 45. (6 points):

The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, and E is the number of lines per set. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

Virtual Memory

Final Exam Question



Problem 9. (12 points):

Address translation. This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 20 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 1024 bytes.
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal.**

TLB

Index	Tag	PPN	Valid
0	03	C3	1
	01	71	0
1	00	28	1
	01	35	1
2	02	68	1
	3A	F1	0
3	03	12	1
	02	30	1
4	7F	05	0
	01	A1	0
5	00	53	1
	03	4E	1
6	1B	34	0
	00	1F	1
7	03	38	1
	32	09	0

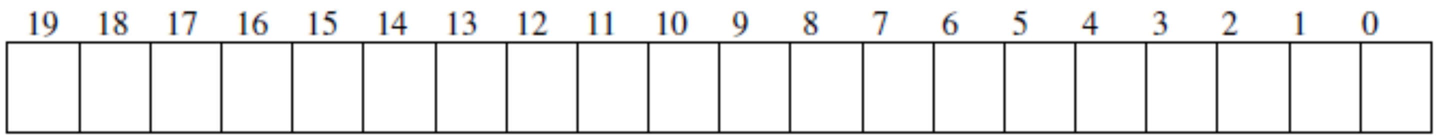
Page Table

VPN	PPN	Valid	VPN	PPN	Valid
000	71	1	010	60	0
001	28	1	011	57	0
002	93	1	012	68	1
003	AB	0	013	30	1
004	D6	0	014	0D	0
005	53	1	015	2B	0
006	1F	1	016	9F	0
007	80	1	017	62	0
008	02	0	018	C3	1
009	35	1	019	04	0
00A	41	0	01A	F1	1
00B	86	1	01B	12	1
00C	A1	1	01C	30	0
00D	D5	1	01D	4E	1
00E	8E	0	01E	57	1
00F	D4	0	01F	38	1

Part 1

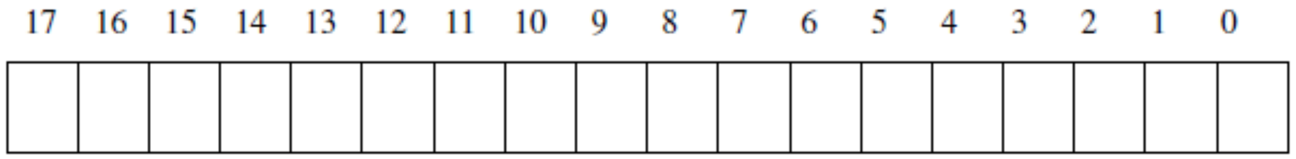
1. The diagram below shows the format of a virtual address. Please indicate the following fields by labeling the diagram:

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag



2. The diagram below shows the format of a physical address. Please indicate the following fields by labeling the diagram:

PPO	The physical page offset
PPN	The physical page number



Part 2

For the given virtual addresses, please indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs. If there is a page fault, enter "-" for "PPN" and leave the physical address blank.

Virtual address: `078E6`

1. Virtual address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Virtual address: `04AA4`

1. Virtual address (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

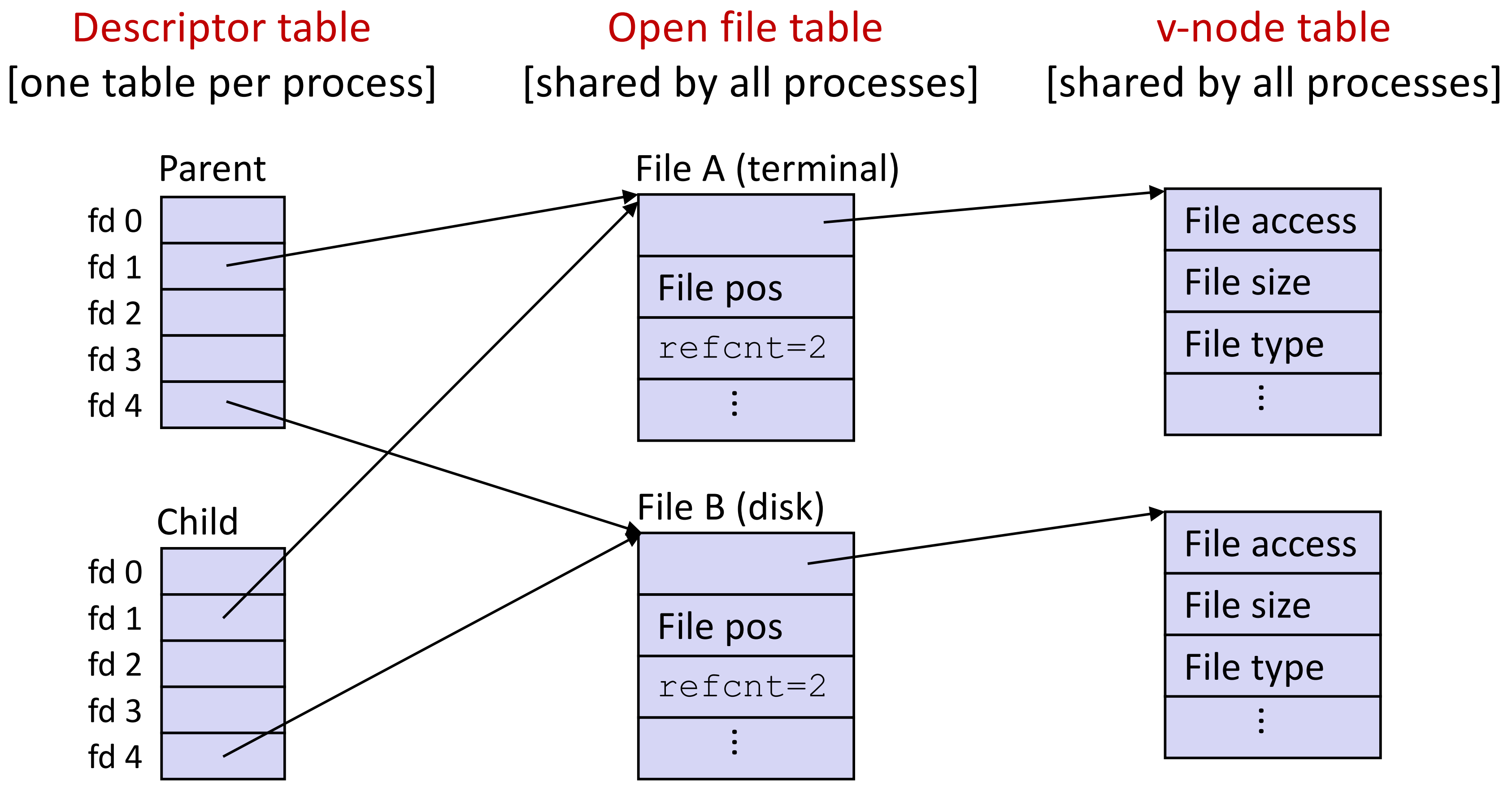
Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	PPN	0x

3. Physical address(one bit per box)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

How Processes Share Files: `fork`

- **A child process inherits its parent's open files**
- ***After*** **`fork`:**
  - Child's table same as parent's, and +1 to each refcnt

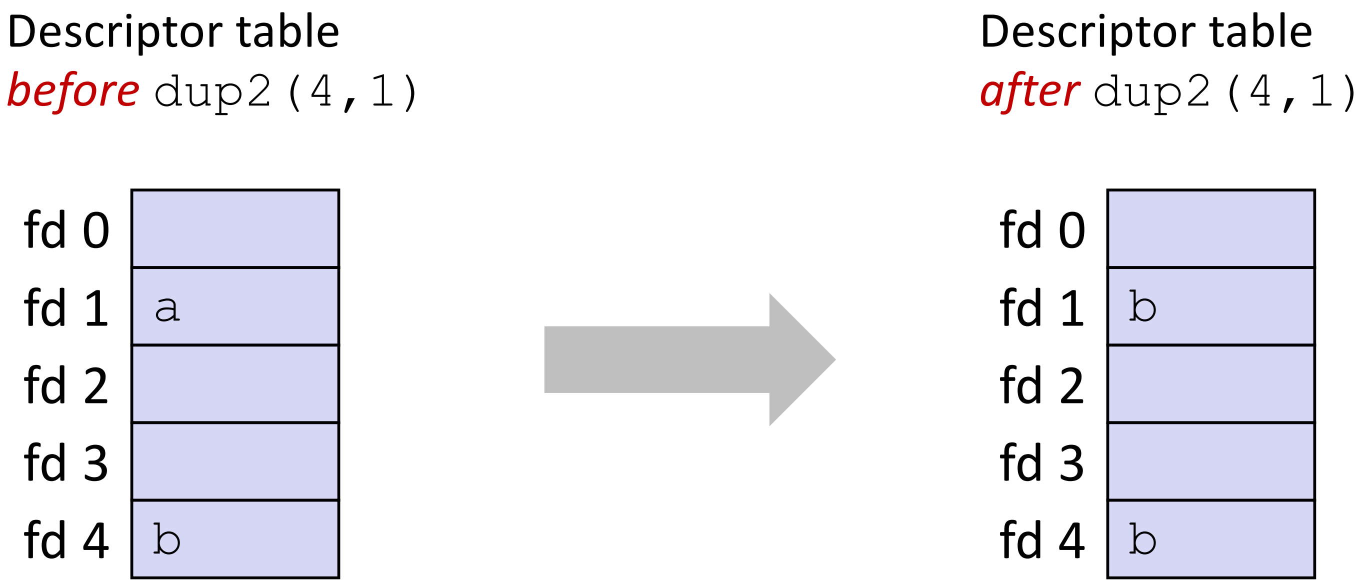


I/O Redirection

- **Question: How does a shell implement I/O redirection?**

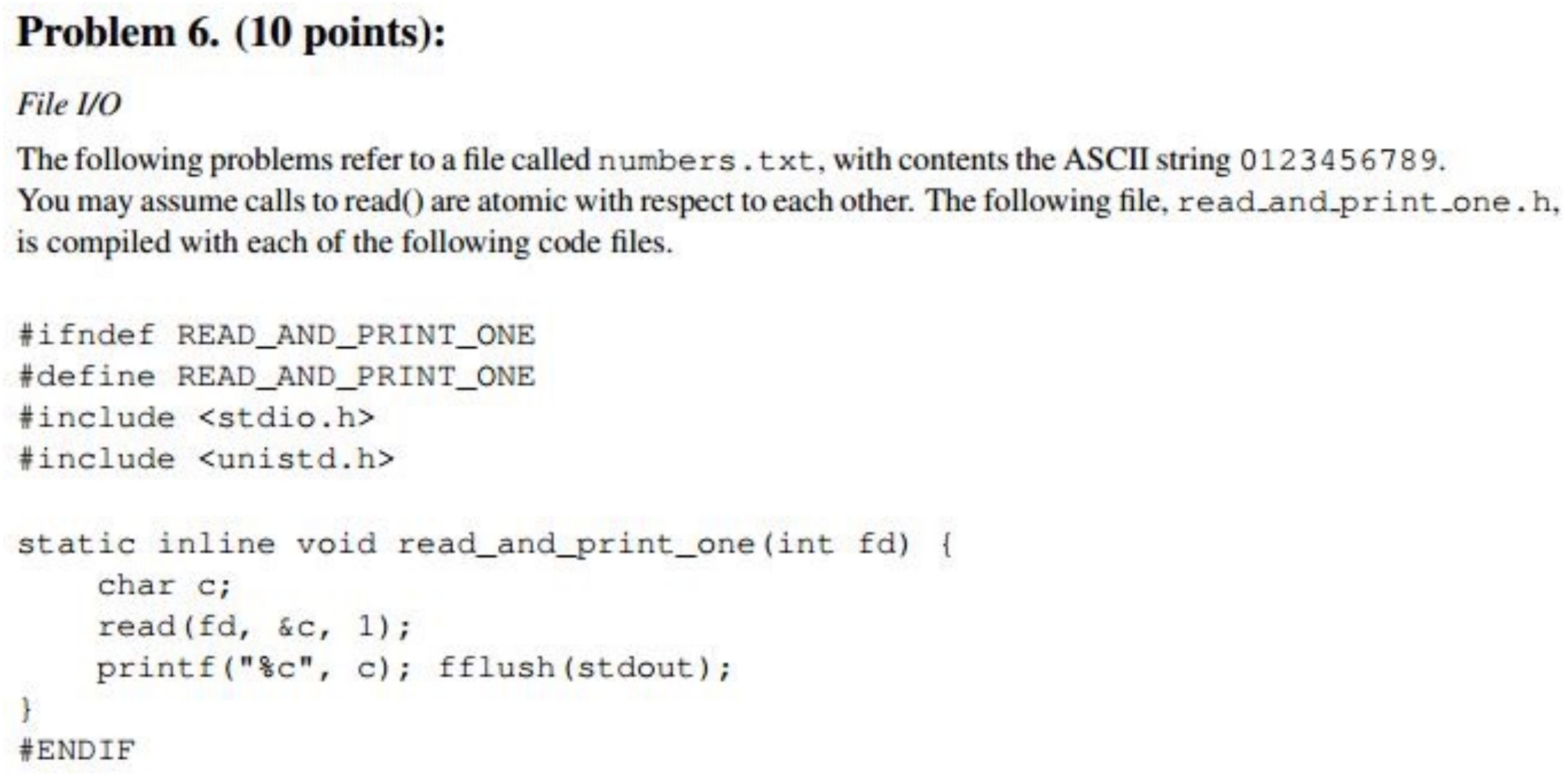
```
linux> ls > foo.txt
```

- **Answer: By calling the `dup2(oldfd, newfd)` function**
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**



I/O Redirection

- **Final Exam Question**



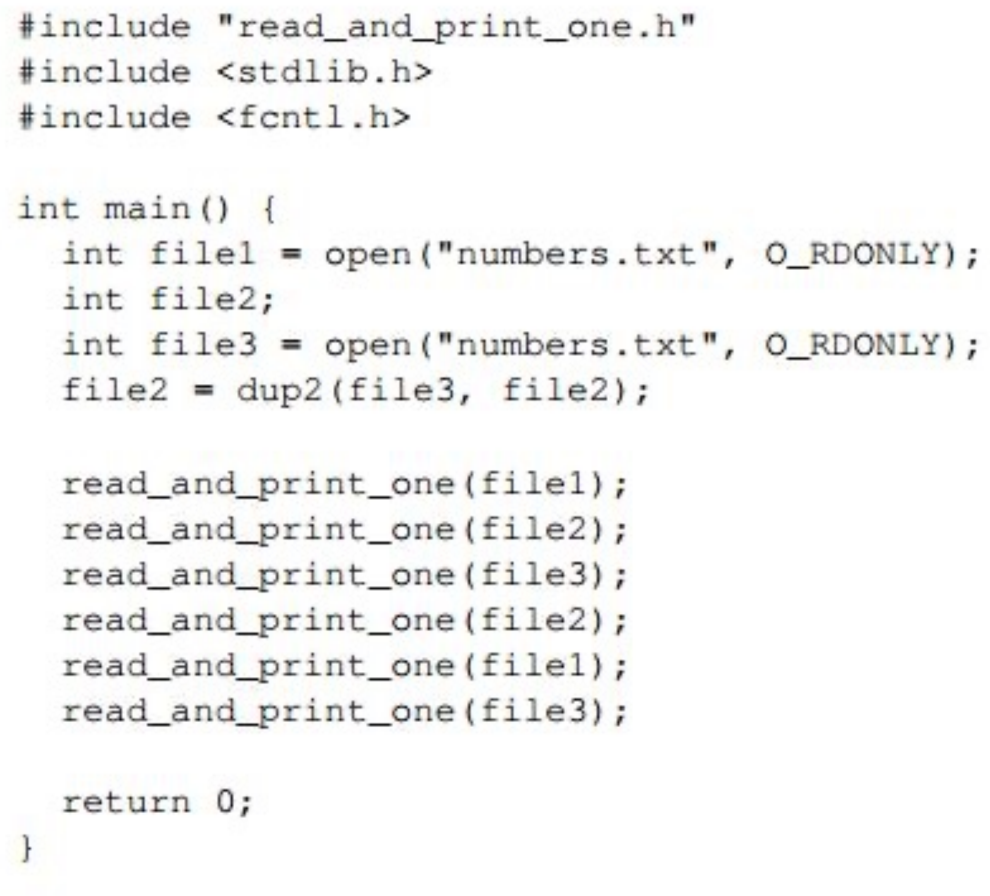
Problem 6. (10 points):

File I/O

The following problems refer to a file called `numbers.txt`, with contents the ASCII string `0123456789`. You may assume calls to read() are atomic with respect to each other. The following file, `read_and_print_one.h`, is compiled with each of the following code files.

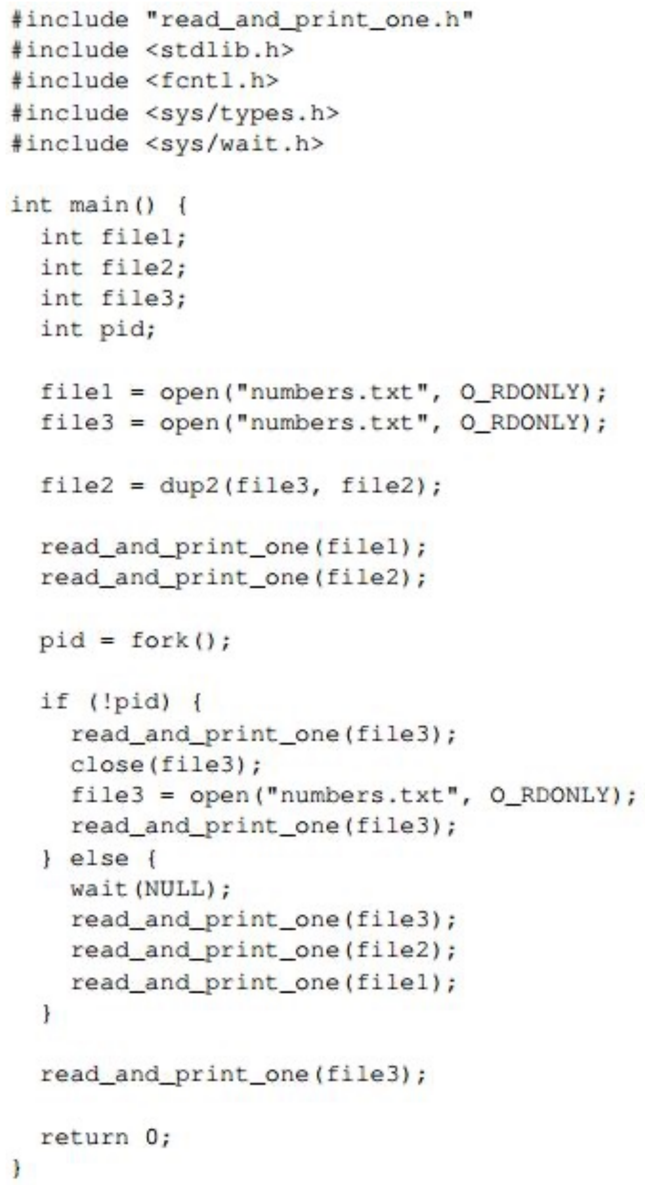
```
#ifndef READ_AND_PRINT_ONE
#define READ_AND_PRINT_ONE
#include <stdio.h>
#include <unistd.h>

static inline void read_and_print_one(int fd) {
    char c;
    read(fd, &c, 1);
    printf("%c", c); fflush(stdout);
}
#ENDIF
```

- **A. List all outputs of the following code.**



```
#include "read_and_print_one.h"
#include <stdlib.h>
#include <fcntl.h>

int main() {
  int file1 = open("numbers.txt", O_RDONLY);
  int file2;
  int file3 = open("numbers.txt", O_RDONLY);
  file2 = dup2(file3, file2);

  read_and_print_one(file1);
  read_and_print_one(file2);
  read_and_print_one(file3);
  read_and_print_one(file2);
  read_and_print_one(file1);
  read_and_print_one(file3);

  return 0;
}
```

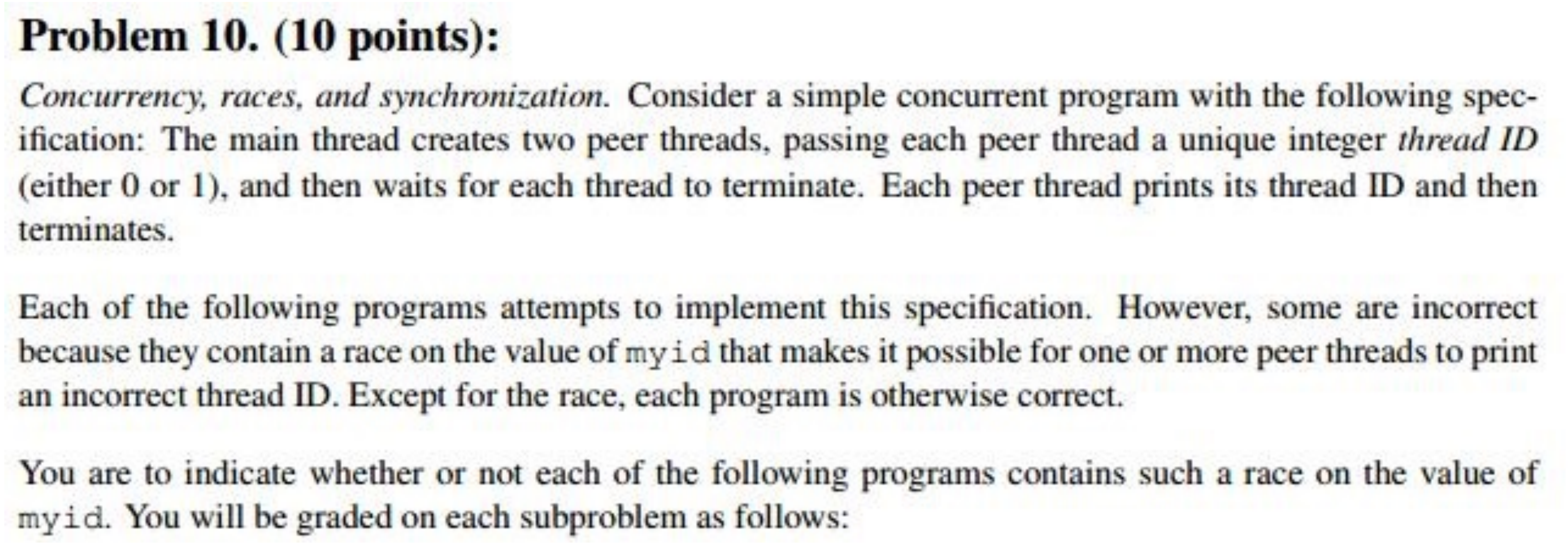
■ B. List all outputs of the following code.



```c
#include "read_and_print_one.h"
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int file1;
  int file2;
  int file3;
  int pid;

  file1 = open("numbers.txt", O_RDONLY);
  file3 = open("numbers.txt", O_RDONLY);

  file2 = dup2(file3, file2);

  read_and_print_one(file1);
  read_and_print_one(file2);

  pid = fork();

  if (!pid) {
    read_and_print_one(file3);
    close(file3);
    file3 = open("numbers.txt", O_RDONLY);
    read_and_print_one(file3);
  } else {
    wait(NULL);
    read_and_print_one(file3);
    read_and_print_one(file2);
    read_and_print_one(file1);
  }

  read_and_print_one(file3);

  return 0;
}
```

Threading

- **Final Exam Question**

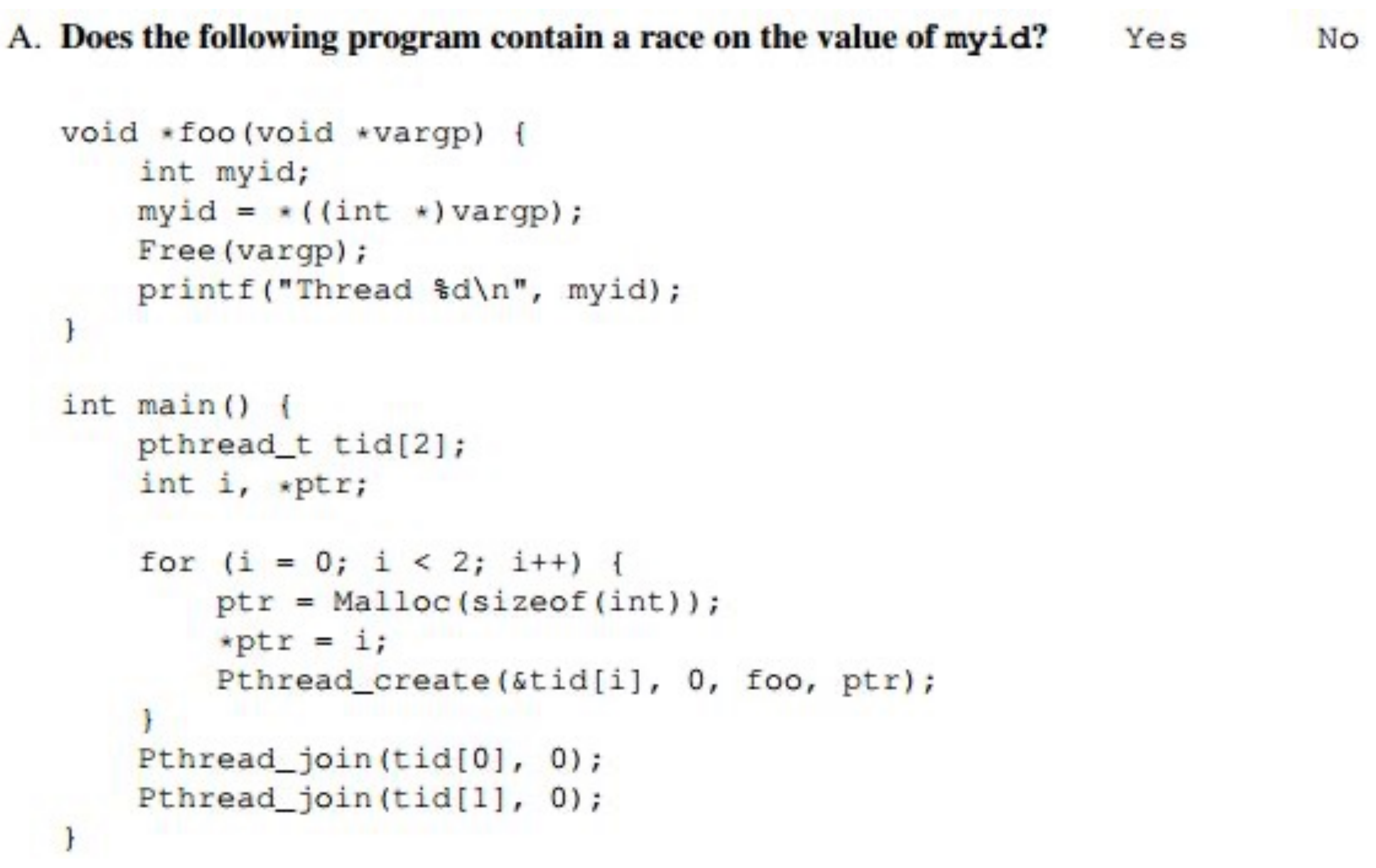


Problem 10. (10 points):

Concurrency, races, and synchronization. Consider a simple concurrent program with the following specification: The main thread creates two peer threads, passing each peer thread a unique integer *thread ID* (either 0 or 1), and then waits for each thread to terminate. Each peer thread prints its thread ID and then terminates.

Each of the following programs attempts to implement this specification. However, some are incorrect because they contain a race on the value of `myid` that makes it possible for one or more peer threads to print an incorrect thread ID. Except for the race, each program is otherwise correct.

You are to indicate whether or not each of the following programs contains such a race on the value of `myid`. You will be graded on each subproblem as follows:



A. **Does the following program contain a race on the value of `myid`?** Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    Free(vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i, *ptr;

    for (i = 0; i < 2; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], 0, foo, ptr);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

B. **Does the following program contain a race on the value of `myid`?** Yes No

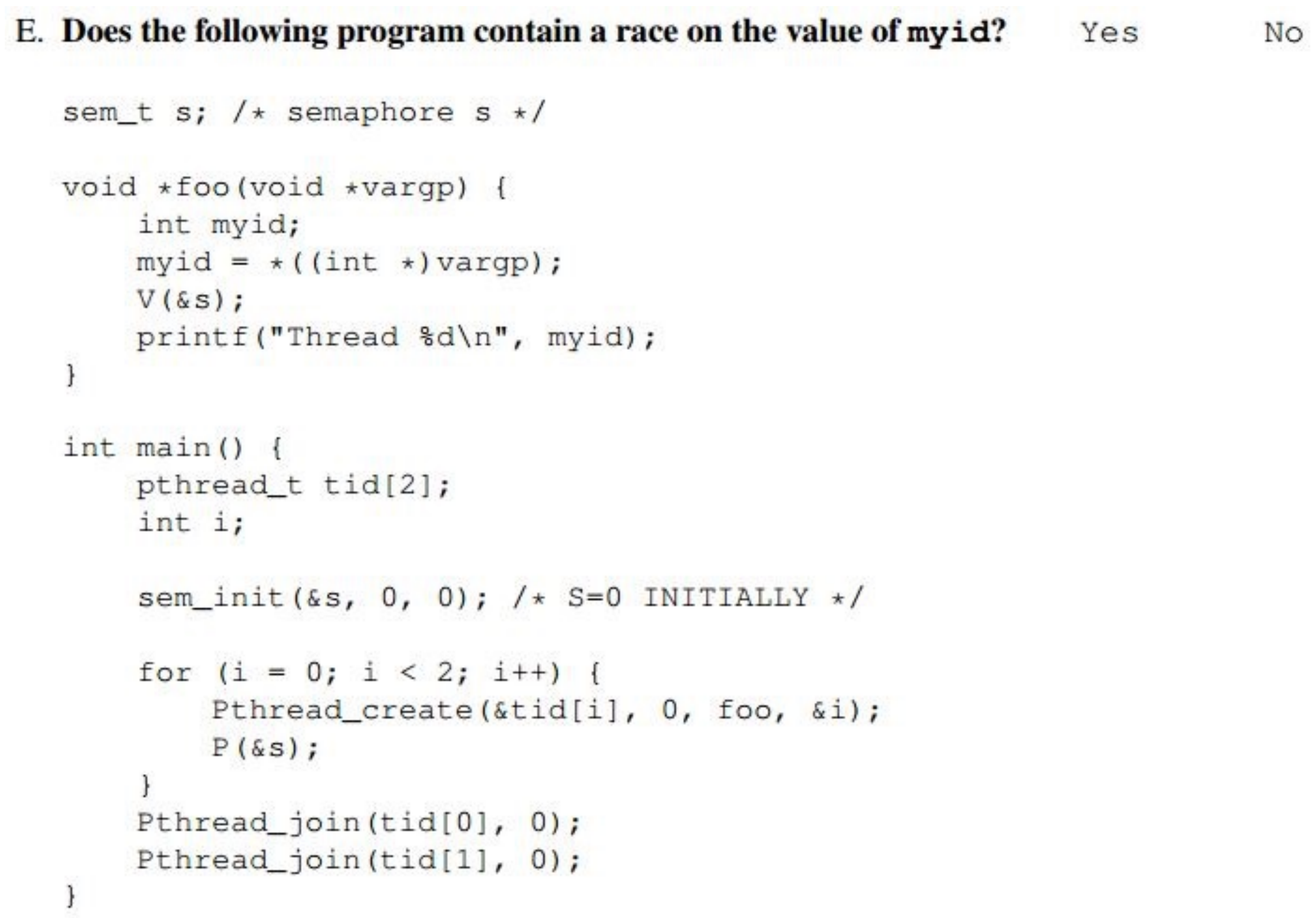
```
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], NULL, foo, &i);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
}
```

C. **Does the following program contain a race on the value of `myid`?** Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = (int)vargp;
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], 0, foo, i);
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

D. **Does the following program contain a race on the value of `myid`?** Yes No

```
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    P(&s);
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 1); /* S=1 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

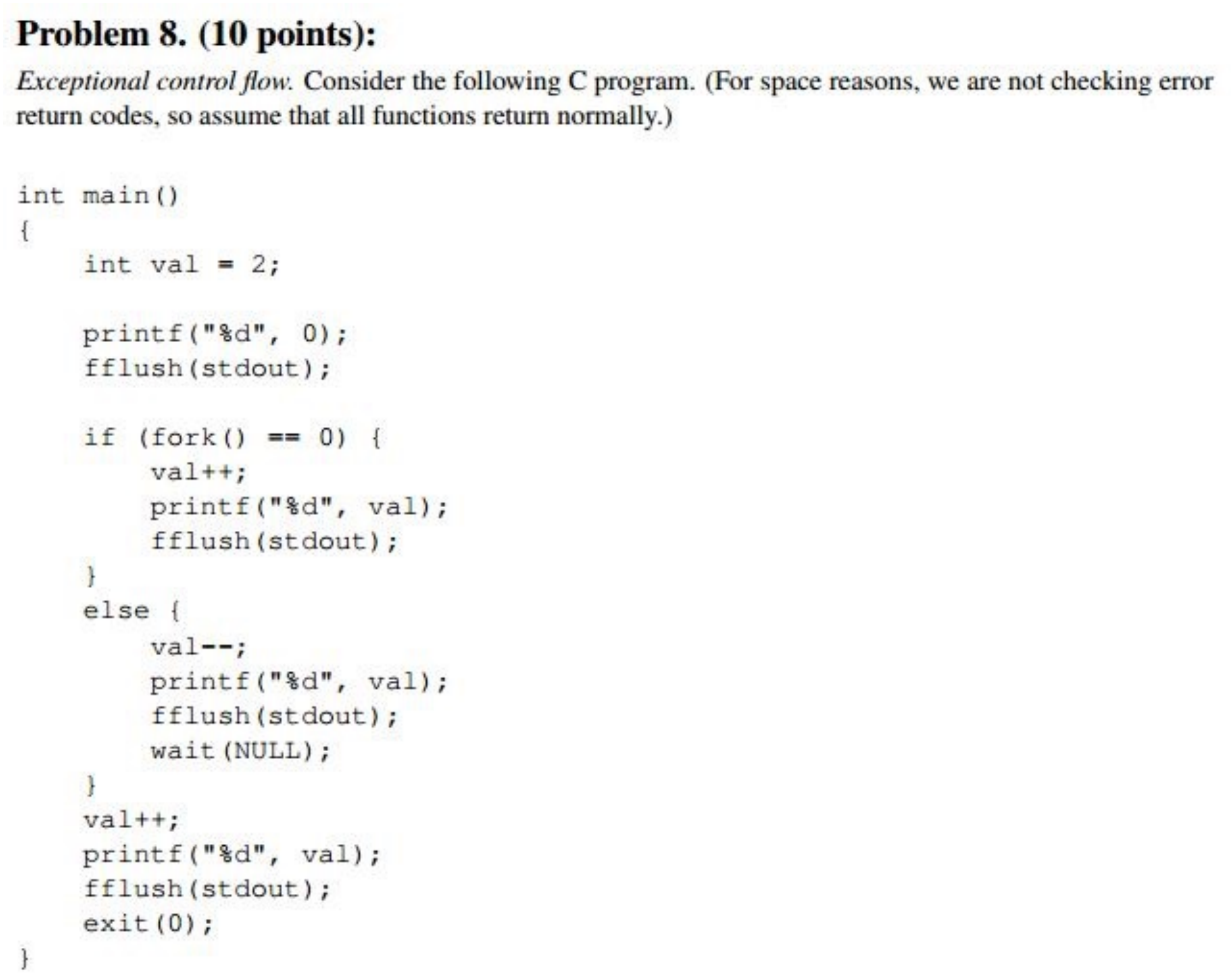


E. **Does the following program contain a race on the value of `myid`?** Yes No

```
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 0); /* S=0 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
        P(&s);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

Processes and Signals

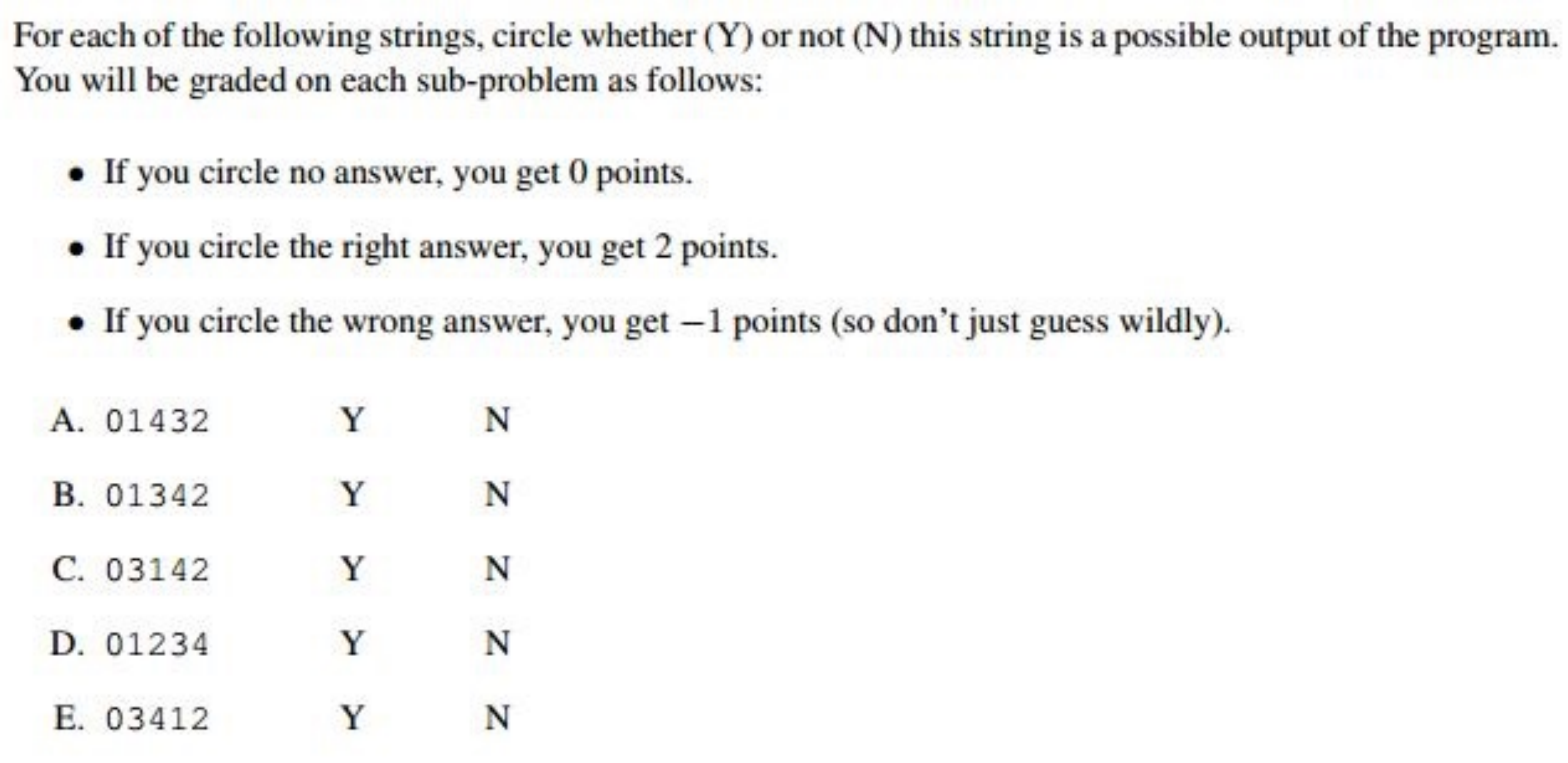
- **Final Exam Question**



Problem 8. (10 points):

Exceptional control flow. Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main()
{
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```



For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program. You will be graded on each sub-problem as follows:

- If you circle no answer, you get 0 points.
- If you circle the right answer, you get 2 points.
- If you circle the wrong answer, you get -1 points (so don't just guess wildly).

A.	`01432`	Y	N
B.	`01342`	Y	N
C.	`03142`	Y	N
D.	`01234`	Y	N
E.	`03412`	Y	N

Hyperthreading

- The following instruction sequences belong to the execution of three concurrent threads. The three threads execute on a 4-way hyperthreaded processor.

	🟥	🟦	🟩
1	`movq (%rdi), %rax`	`addq %rax, %rcx`	`movq (%rdi), %rax`
2	`movq (%rsi), %rdx`	`addq %rcx, %rdx`	`movq %rax, (%rsi)`
3	`movq %rdx, (%rdi)`	`movq (%rdx), %rbx`	`movq (%rdi), %rbx`
4	`movq %rax, (%rsi)`	`movq %rbx, (%rsi)`	`movq %rbx, (%rsi)`
5	`ret`	`ret`	`ret`

- The scheduler fills the four issue slots with four instructions with the goal of generating as many memory operations as possible. Which of the following combination are valid?

(A)	🟥 1	🟥 2	🟦 3	🟦 4
(B)	🟥 1	🟥 2	🟦 4	🟩 1
(C)	🟥 1	🟥 2	🟩 1	🟩 3
(D)	🟩 3	🟦 1	🟥 1	🟥 2

Sequential Consistency

- Consider the execution of the following concurrent processes on two different processors, and `A` and `B` are originally cached by both processors with initial value of 0.

```
P1:          A = 0
             .......
             A = 1;
             if (B ==0) ...
```

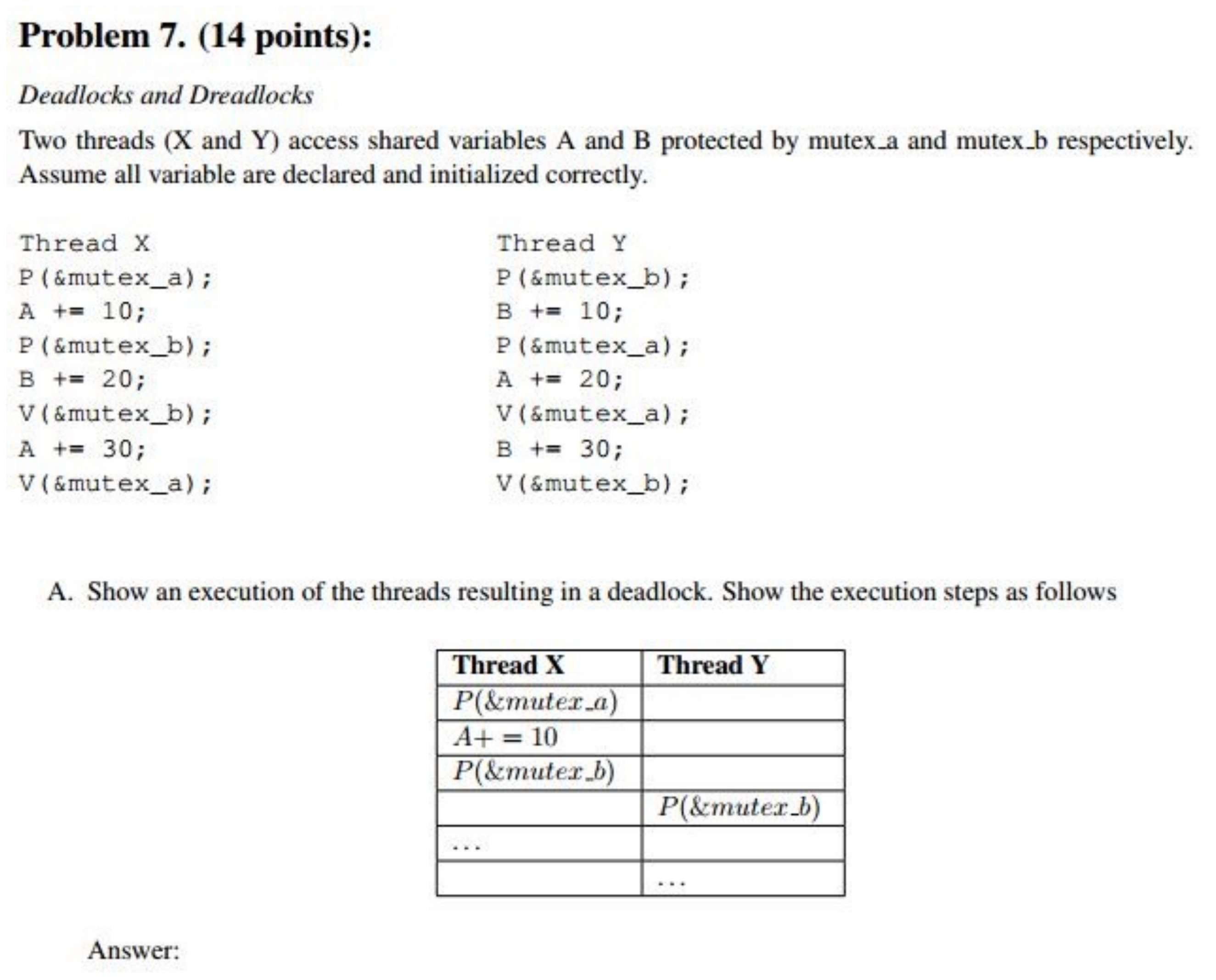
```
P2:          B = 0
             .......
             B = 1;
             if (A ==0) ...
```

- Under sequential consistency which of the following outcomes are possible?

(A)	NT	NT
(B)	T	T
(C)	T	NT
(D)	NT	T

Deadlock

- **Final Exam Question**



Problem 7. (14 points):

Deadlocks and Dreadlocks

Two threads (X and Y) access shared variables A and B protected by mutex_a and mutex_b respectively. Assume all variable are declared and initialized correctly.

```
Thread X                          Thread Y
P(&mutex_a);                      P(&mutex_b);
A += 10;                          B += 10;
P(&mutex_b);                      P(&mutex_a);
B += 20;                          A += 20;
V(&mutex_b);                      V(&mutex_a);
A += 30;                          B += 30;
V(&mutex_a);                      V(&mutex_b);
```

A. Show an execution of the threads resulting in a deadlock. Show the execution steps as follows

Thread X	**Thread Y**
$P(\&mutex\_a)$	
$A+=10$	
$P(\&mutex\_b)$	
	$P(\&mutex\_b)$
...	
	...

Answer:

B. There are different approaches to solve the deadlock problem. Modify the code above to show **two** approaches to prevent deadlocks. You can declare new mutex variables if required. Do not change the order or amount of the increments to A and B. Rather, change the locking behavior around them. The final values of A and B must still be guaranteed to be incremented by 60.

Answer:

Questions?