Full Name:_____

CS 15-213, Spring 2002

Exam 2

March 28, 2002

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 54 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (4):	
2 (12):	
3 (13):	
4 (7):	
5 (6):	
6 (12):	
TOTAL (54):	

In this problem, you will compare the performance of direct mapped and 4-way associative caches for the initialization of 2-dimensional array of data structures. Both caches have a size of 1024 bytes. The direct mapped cache has 64-byte lines while the 4-way associative cache has 32-byte lines.

You are given the following definitions

```
typedef struct{
    float irr[3];
    short theta;
    short phi;
} photon_t;

photon_t  surface[16][16];
register int i, j, k;
```

Also assume that

- `sizeof(short) = 2`
- `sizeof(float) = 4`
- `surface` begins at memory address `0`
- Both caches are initially empty
- The array is stored in row-major order
- Variables `i`,`j`,`k` are stored in registers and any access to these variables does not cause a cache miss

A. What fraction of the writes in the following code will result in a miss in the direct mapped cache?

```
for (i = 0; i < 16; i ++)
{
    for (j = 0; j < 16; j ++)
    {
        for(k = 0; k < 3; k ++)
        {
            surface[i][j].irr[k] = 0.;
        }
        surface[i][j].theta = 0;
        surface[i][j].phi = 0;
    }
}
```

Miss rate for writes to surface:_____%

B. Using code in part A, what fraction of the writes will result in a miss in the 4-way associative cache?

Miss rate for writes to surface: _____%

```
for (i = 0; i < 16; i ++)
{
    for (j = 0; j < 16; j ++)
    {
        for (k = 0; k < 16; k ++)
        {
            surface[j][i].irr[k] = 0;
        }
        surface[j][i].theta = 0;
        surface[j][i].phi = 0;
    }
}
```

Miss rate for writes to surface:_____%

D. Using code in part C, what fraction of the writes will result in a miss in the 4-way associative cache?

Miss rate for writes to surface:_____%

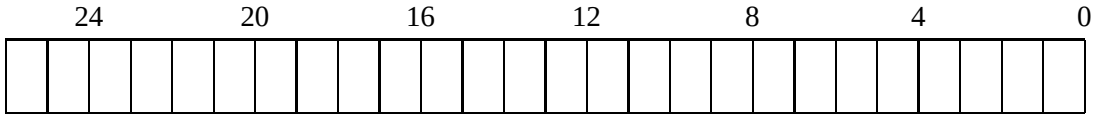
The following problem concerns various aspects of virtual memory.

Part I.

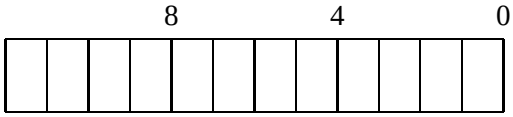
The following are attributes of the machine that you will need to consider:

- Memory is byte addressable
- Virtual Addresses are 26 bits wide
- Physical Addresses are 12 bits wide
- Pages are 512 bytes
- Each Page Table Entry contains:
  - Physical Page Number
  - Valid Bit

A. The box below shows the format of a virtual address. Indicate the bits used for the VPN (Virtual Page Number) and VPO (Virtual Page Offset).



B. The box below shows the format for a physical address. Indicate the bits used for the PPN (Physical Page Number) and PPO (Physical Page Offset)



C. **Note:** For the questions below, answers of the form 2^i are acceptable. Also, please note the units of each answer

How much *virtual* memory is addressable? _____ bytes

How much *physical* memory is addressable? _____ bytes

How many bits is each Page Table Entry? _____ bits

How large is the Page Table? _____ bytes

(4 points)

Application images for the operating system used on the machine in part I are formed with a subset of ELF. They only contain the `.text` and `.data` regions.

When a process uses `fork()` to create a new process image in memory, the operating system maintains each process' view that it has full control of the virtual address space. To the programmer, the amount of physical memory used by the two processes together is twice that which is used by a single process.

****NOTE** Read both questions below before answering**

A. How can the operating system conservatively save physical memory when creating the new process image during a call to `fork()` with respect to the `.text` and `.data` regions?

B. Imagine a process that acts in the following fashion:

```
int my_array[HUGE_SIZE];

int main() {

  /* Code to initialize my_array */

  if(fork() == 0) {
    exit(0);
  } else {

    /* Code that calculates and prints
     * the sum of the elements in my_array
     */
  }
}
```

How could the operating system be aggressive by temporarily saving memory beyond what was saved in part A in this case? *Hint 1: Note that the child doesn't change my_array, but the operating system has to be prepared for such an event since it doesn't know the future. Hint 2: Think about protection bits and page faults.*

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable, and memory accesses are to 1-byte {**not 4-byte**} words.
- Virtual addresses are 17 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 256 bytes.
- The TLB is 4-way set associative with 16 total entries.
- The cache is 2-way set associative, with a 4-byte line size and 64 total entries.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages, and the cache are as follows:

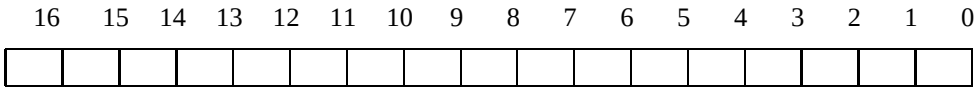
TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	C	0
	77	9	1
1	01	4	1
	32	A	1
	02	F	0
	73	0	1
2	02	3	1
	0F	B	0
	04	3	0
	26	C	0
3	00	8	1
	7A	2	1
	21	1	0
	17	E	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
000	C	0	010	1	1
001	7	1	011	8	1
002	3	1	012	3	0
003	8	1	013	E	1
004	0	0	014	6	0
005	5	0	015	C	0
006	C	1	016	7	0
007	4	1	017	2	1
008	D	1	018	9	1
009	F	0	019	A	0
00A	3	1	01A	B	0
00B	0	1	01B	3	1
00C	0	0	01C	2	1
00D	F	1	01D	9	0
00E	4	0	01E	5	0
00F	7	1	01F	B	1

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	38	1	00	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22
4	12	0	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

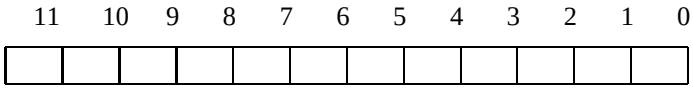
1. The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

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



2. The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

   PPO The physical page offset
   PPN The physical page number
   CO The Cache Block Offset
   CI The Cache Index
   CT The Cache Tab

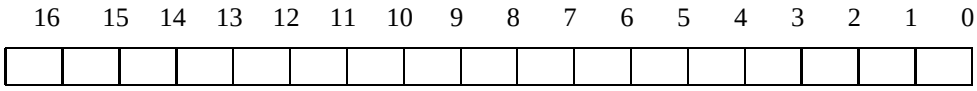


For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a cache miss, enter "-" for "Cache Byte Returned." If there is a page fault, enter "-" for "PPN" and leave part C blank.

Virtual address: `01FAD`

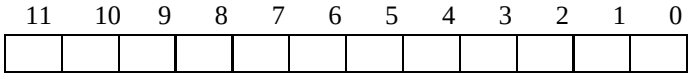
1. Virtual address format (one bit per box)



2. Address translation

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

3. Physical address format (one bit per box)



4. Physical memory reference

Parameter	Value
Block Offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Value of Cache Byte Returned	0x

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

```
int main ()
{
  int j = 0;

  for ( j = 0; j < 3; j++ )
    {
      if ( fork() == 0 )
        printf ( "%i", j + 1 );
      else
        printf ( "%i", j + 4 );

      printf ( "%i", j );
    }

  fflush ( stdout );
  return 0;
}
```

Tell how many of each number will be printed:

0 _____

1 _____

2 _____

3 _____

4 _____

5 _____

6 _____

The following program performs a shell-like file copying. It forks out a child process to copy the file. The parent process waits for the child process to finish and, at the same time, captures the SIGINT signal. The signal handler asks the user whether to kill the copying or not. If the user answers 'y', the child process will be killed. Otherwise, the signal handler just silently returns from the signal handler.

```
 1: pid_t   child_pid = 0;

 2: void  handler(int sig)
 3: {
 4:    char answer;

 5:    printf("Stop copying?\n");
 6:    scanf("%c", &answer);
 7:    if (answer == 'y' && child_pid != 0)

 8:          ____________________________________________________________;
 9: }

10: void  copy()
11: {

12:     ____________________________________________________________;

13:    if ((child_pid = fork()) == 0) {

14:          ____________________________________________________________;
15:          copy_file();
16:          exit(0);
17:    }
18:    else {

19:         ____________________________________________________________;
20:         if (wait(NULL) == child_pid) {
21:             printf("Child process %d ended.\n", child_pid);
22:         }
23:    }
24: }
```

A. Fill in the blank lines to make the code work. Please fill in "(empty)" if no code is needed.

B. What happens if the user presses Ctrl-C when the program execution is at line 6? And why?

Introduction

Below we have provided the source code for a simple C Language program. Additionally we have provided an excerpt of its disassembled assembly language source code, its Procedure Linkage Table (PLT), and snapshots of its Global Offset Table (GOT) at three different stages of its execution.

C Source Code:

```
 1. #include <stdio.h>
 2. #include <math.h>
 3.
 4. int main (int argc, char *argv)
 5. {
 6.   char *cptr;
 7.
 8.   cptr = (char *) malloc (1024);
 9.
10.   sprintf (cptr, "Hi Mom\n");
11.   printf ("%s\n", cptr);
12.   fflush (stdout);
13.
14.   return 0;
15. }
```

Questions

A. Please label each entry in the PLT as one of the following:

(a) `malloc()`
(b) `sprintf()`
(c) `printf()`
(d) `fflush()`
(e) code to invoke the dynamic linker, itself
(f) none of the above

B. Please label each entry in the first GOT as one of the following:

(a) `malloc()`
(b) `sprintf()`
(c) `printf()`
(d) `fflush()`
(e) code to invoke the dynamic linker, itself
(f) none of the above

C. Each of the three GOTs provided was extracted within gdb at three different stages of the program's execution. Please indicate when each snapshot might have been taken by indicating the last line to completely execute before the snapshot was recorded. If the snapshot was recorded before or after main() executes, please indicate this by recording "before execution", "before main" or "after main" in place of a line number.

```
int main (int argc, char *argv)
{
    080484a4 <main>:
     80484a4:       55                      push   %ebp
     80484a5:       89 e5                   mov    %esp,%ebp
     80484a7:       83 ec 14                sub    $0x14,%esp
     80484aa:       53                      push   %ebx

 char *cptr;
 cptr = (char *) malloc (1024);

     80484ab:       83 c4 f4                add    $0xfffffff4,%esp
     80484ae:       68 00 04 00 00          push   $0x400
     80484b3:       e8 d8 fe ff ff          call   8048390 <_init+0x60>
     80484b8:       89 c3                   mov    %eax,%ebx

 sprintf (cptr, "Hi Mom\n");
     80484ba:       83 c4 f8                add    $0xfffffff8,%esp
     80484bd:       68 48 85 04 08          push   $0x8048548
     80484c2:       53                      push   %ebx
     80484c3:       e8 08 ff ff ff          call   80483d0 <_init+0xa0>

 printf ("%s\n", cptr);
     80484c8:       83 c4 20                add    $0x20,%esp
     80484cb:       83 c4 f8                add    $0xfffffff8,%esp
     80484ce:       53                      push   %ebx
     80484cf:       68 50 85 04 08          push   $0x8048550
     80484d4:       e8 e7 fe ff ff          call   80483c0 <_init+0x90>

 fflush (stdout);
     80484d9:       a1 40 96 04 08          mov    0x8049640,%eax
     80484de:       83 c4 f4                add    $0xfffffff4,%esp
     80484e1:       50                      push   %eax
     80484e2:       e8 99 fe ff ff          call   8048380 <_init+0x50>

 return 0;
     80484e7:       8b 5d e8                mov    0xffffffe8(%ebp),%ebx
     80484ea:       31 c0                   xor    %eax,%eax
     80484ec:       89 ec                   mov    %ebp,%esp
     80484ee:       5d                      pop    %ebp
     80484ef:       c3                      ret
}
```

This entry belongs to _____

```
8048360:       ff 35 78 95 04 08       pushl  0x8049578
8048366:       ff 25 7c 95 04 08       jmp    *0x804957c
804836c:       00 00                   add    %al,(%eax)
804836e:       00 00                   add    %al,(%eax)
```

This entry belongs to _____

```
8048370:       ff 25 80 95 04 08       jmp    *0x8049580
8048376:       68 00 00 00 00          push   $0x0
804837b:       e9 e0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to _____

```
8048380:       ff 25 84 95 04 08       jmp    *0x8049584
8048386:       68 08 00 00 00          push   $0x8
804838b:       e9 d0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to _____

```
8048390:       ff 25 88 95 04 08       jmp    *0x8049588
8048396:       68 10 00 00 00          push   $0x10
804839b:       e9 c0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to _____

```
80483a0:       ff 25 8c 95 04 08       jmp    *0x804958c
80483a6:       68 18 00 00 00          push   $0x18
80483ab:       e9 b0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to _____

```
80483b0:       ff 25 90 95 04 08       jmp    *0x8049590
80483b6:       68 20 00 00 00          push   $0x20
80483bb:       e9 a0 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to _____

```
80483c0:       ff 25 94 95 04 08       jmp    *0x8049594
80483c6:       68 28 00 00 00          push   $0x28
80483cb:       e9 90 ff ff ff          jmp    8048360 <_init+0x30>
```

This entry belongs to _____

```
80483d0:       ff 25 98 95 04 08       jmp    *0x8049598
80483d6:       68 30 00 00 00          push   $0x30
80483db:       e9 80 ff ff ff          jmp    8048360 <_init+0x30>
```

```
0x8049574 <_GLOBAL_OFFSET_TABLE_>:      0x080495a0  _____________________
0x8049578 <_GLOBAL_OFFSET_TABLE_+4>:    0x00000000  _____________________
0x804957c <_GLOBAL_OFFSET_TABLE_+8>:    0x00000000  _____________________
0x8049580 <_GLOBAL_OFFSET_TABLE_+12>:   0x08048376  _____________________
0x8049584 <_GLOBAL_OFFSET_TABLE_+16>:   0x08048386  _____________________
0x8049588 <_GLOBAL_OFFSET_TABLE_+20>:   0x08048396  _____________________
0x804958c <_GLOBAL_OFFSET_TABLE_+24>:   0x080483a6  _____________________
0x8049590 <_GLOBAL_OFFSET_TABLE_+28>:   0x080483b6  _____________________
0x8049594 <_GLOBAL_OFFSET_TABLE_+32>:   0x080483c6  _____________________
0x8049598 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483d6  _____________________
```

B. gdb dump of .got at _____

```
0x8049574 <_GLOBAL_OFFSET_TABLE_>:      0x080495a0
0x8049578 <_GLOBAL_OFFSET_TABLE_+4>:    0x40013ed0
0x804957c <_GLOBAL_OFFSET_TABLE_+8>:    0x4000a960
0x8049580 <_GLOBAL_OFFSET_TABLE_+12>:   0x08048376
0x8049584 <_GLOBAL_OFFSET_TABLE_+16>:   0x08048386
0x8049588 <_GLOBAL_OFFSET_TABLE_+20>:   0x08048396
0x804958c <_GLOBAL_OFFSET_TABLE_+24>:   0x080483a6
0x8049590 <_GLOBAL_OFFSET_TABLE_+28>:   0x080483b6
0x8049594 <_GLOBAL_OFFSET_TABLE_+32>:   0x080483c6
0x8049598 <_GLOBAL_OFFSET_TABLE_+36>:   0x080483d6
```

C. gdb dump of .got at _____

```
0x8049574 <_GLOBAL_OFFSET_TABLE_>:      0x080495a0
0x8049578 <_GLOBAL_OFFSET_TABLE_+4>:    0x40013ed0
0x804957c <_GLOBAL_OFFSET_TABLE_+8>:    0x4000a960
0x8049580 <_GLOBAL_OFFSET_TABLE_+12>:   0x400fa530
0x8049584 <_GLOBAL_OFFSET_TABLE_+16>:   0x08048386
0x8049588 <_GLOBAL_OFFSET_TABLE_+20>:   0x400734e0
0x804958c <_GLOBAL_OFFSET_TABLE_+24>:   0x080483a6
0x8049590 <_GLOBAL_OFFSET_TABLE_+28>:   0x400328cc
0x8049594 <_GLOBAL_OFFSET_TABLE_+32>:   0x080483c6
0x8049598 <_GLOBAL_OFFSET_TABLE_+36>:   0x40068080
```