

## Question 1: Structs

For this question, assume a 64-bit machine and the following C struct definition.

```
typedef struct { K:
  char* title; 8 // title (e.g. "HW SW INTERFACE")
  char dept[3]; 1 // dept (e.g. "CSE")
  short num; 2 // course number (e.g. 351)
  int enrolled; 4 // students enrolled
} course; Kmax = 8
```

(A) How much memory, in bytes, does an instance of course use? How many of those bytes are *internal* fragmentation and *external* fragmentation?

sizeof(course)	Internal	External
<b>24 bytes</b>	<b>3 bytes</b>	<b>4 bytes</b>

**Alignment requirements listed above in red next to the struct fields. A course instance:**

<b>title</b>	<b>dept</b>		<b>num</b>		<b>enrolled</b>	
--------------	-------------	--	------------	--	-----------------	--

**0 8 11 12 14 16 20 24**

**The unused bytes around num count as internal fragmentation, the unused bytes after enrolled count as external fragmentation.**

(B) Assume that an instance **course** *c* is allocated on the stack and an array **char** *ar*[] is allocated 40 bytes below *c* (i.e.  $\&ar + 0x28 == (\mathbf{char}^*)\&c$ ). Fill in the blanks below with the new ASCII characters stored in *c.dept* after the following loop is executed. Hint: recall that the values 0x30 to 0x39 correspond to the ASCII characters '0' to '9'.

```
for (int i = 0; i < 52; ++i) {
  ar[i] = i;
}
```

*c.dept*[0]: **'0'**

*c.dept*[1]: **'1'**

*c.dept*[2]: **'2'**

Starting from the beginning of ar, we store the values 0 to 39 before we reach the struct c. The values 40 to 47 overwrite the bytes of c.title (address 0x2f2e2d2c2b2a2928, assuming little-endian). c.dept then gets overwritten with the values 48 = 0x30 = '0', 49 = 0x31 = '1', and 50 = 0x32 = '2'.

### Question 2: Caching

We have 256 KiB of RAM and a 4-KiB L1 data cache that is 2-way set associative with 32-byte blocks and random replacement, write-back, and write allocate policies.

(A) Calculate the TIO address breakdown:

Tag bits	Index bits	Offset bits
7	6	5

18 address bits.  $\log_2 32 = 5$  offset bits.  $2^{12}$ -B cache = 128 blocks. 2 blocks/set  $\rightarrow 64 = 2^6$  sets.

(B) The code snippet below accesses two arrays of doubles. Assuming i is stored in a register and the cache starts *cold*, give the memory access pattern (read or write to which elements/addresses) and compute the **miss rate**.

```
#define SIZE 128
double src[SIZE]; // &src = 0x08000 (physical addr)
double dst[SIZE]; // &dst = 0x0E000 (physical addr)
for (int i = 0; i < SIZE; i += 1) {
    dst[i] = src[i];
    src[i] = i;
}
```

Per Iteration:	Access 1:	Access 2:	Access 3:
(circle) $\rightarrow$	R / W to	R / W to	R / W to
(fill in) $\rightarrow$	src[i]	dst[i]	src[i]
<p>src[i] and dst[i] map into the same set because their index fields match. However, our cache is 2-way set associative, so they do not conflict. Each block holds 32 B = 4 doubles, so for the 4 iterations in the same cache block, we get MMH HHH HHH HHH for a miss rate of <math>2/12 = 1/6</math>.</p>			<p><b>Code Miss Rate:</b></p> <p style="text-align: center;">1/6</p>

(C) For each of the proposed (independent) changes, draw  $\uparrow$  for “increased”,  $-$  for “no change”, or  $\downarrow$  for “decreased” to indicate the effect on the **miss rate from Part B** for the code above:

Use float instead  $\downarrow$  Double the cache size  $-$

Half the associativity \_\_\_↑\_\_\_ No-write allocate \_\_\_↑\_\_\_

Using floats means we access each block twice as much ( $MR = 1/12$ ). Doubling cache size doubles the number of sets, but  $src[i]$  and  $dst[i]$  still map to the same set. Direct-mapped would cause  $src[i]$  and  $dst[i]$  to generate conflict misses. No-write allocate means we *don't* bring in the block for  $dst$  into the cache on access 2, so future access 2s continue to be Misses.

(D) Assume it takes 160 ns to get a block of data from main memory. If our L1 data cache has a hit time of 5 ns and a miss rate of 5%, what is our average memory access time (AMAT)?

$$AMAT = HT + MR \times MP = 5 \text{ ns} + 0.05 \times 160 \text{ ns} = 5 + 8 \text{ ns}$$

13 ns

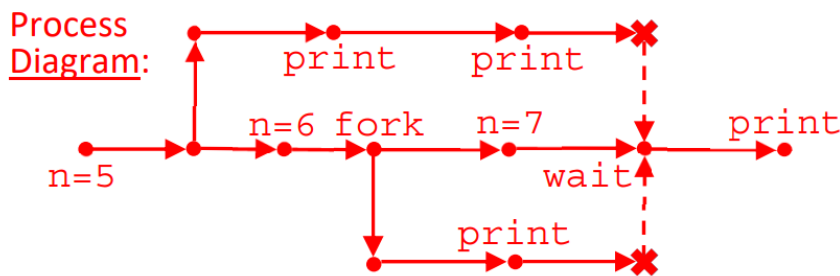
### Question 3: Processes

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes:

```
void concurrent(void) {  
    int n = 5;  
    if (fork()) {  
        n++;  
        if (fork()) {  
            n++;  
            wait();  
        }  
        printf("%d, ", n);  
        exit(0);  
    } else {  
        printf("%d, ", n);  
    }  
    printf("%d, ", n);  
    exit(0);  
}
```

7 Possible Outcomes:

- 1) 5, 5, 6, 7,
- 2) 5, 5, 7, 6,
- 3) 5, 6, 5, 7,
- 4) 5, 6, 7, 5,
- 5) 6, 5, 5, 7,
- 6) 6, 5, 7, 5,
- 7) 6, 7, 5, 5,



(B) For the following examples of exception causes, write “S” for synchronous or “A” for asynchronous from the perspective of the user process. [4 pt]

System call   S                        Divide by zero   S    
 Segmentation fault   S                Key pressed   A  

Everything but a key press is caused by an assembly instruction *within* your program.

(C) Fill in the following blanks with “A” for always, “S” for sometimes, and “N” for never if the following would be different when **context switching** to a *different* process?

Process ID   A   Program   S   PTBR   A   Condition Codes   S  

Every process has a unique ID and its own page table, but could be running different instances of the same program. Each process has its own execution state (including the condition codes), but it is possible that the condition codes have the same *values* at the instance we switch.

(D) Is the following statement True or False? Provide a *brief* justification: a single process can execute multiple programs simultaneously.

Circle one:    True / **False**  
Justification: One process is dedicated to running one program at a time. The program defines the instructions, initial memory state, etc. of the process, so two programs can't exist within the same process at once.

### Question 4: Virtual Memory

Our system has the following setup:

- 15-bit virtual addresses and 2 KiB of RAM with 256-byte pages
- A 4-entry fully-associative TLB with LRU replacement
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)

(A) Compute the following values:

page offset width **8 bits** # of TLB sets **1 set**

# of virtual pages  **$2^7$  pages** minimum width of PTBR **11 bits**

**Page offset is  $\log_2 256 = 8$  bits wide. # of virtual pages is  $2^{n-p} = 2^7$ . The TLB is fully associative, so only has 1 set. The page table lives in physical memory, so the PTBR must hold its physical address, which need to be at least 11 bits wide to address all 2 KiB of RAM.**

(B) Assuming that the TLB is in the state shown (permission bits: 1 = allowed, 0 = disallowed), give example addresses that will fulfill the following scenarios:

TLBT	PPN	Valid	D	R	W	X
0x20	0xc	1	0	1	0	0
0x7f	0xa	1	0	1	1	0
0x7e	0xf	1	0	1	1	0
0x04	0xe	1	0	1	1	1

A value in `%rip` that causes a TLB Hit and no exception:

**Want TLB entry with  $V=1, X=1 \rightarrow$  VPN 0x04. Any address between 0x0400-0x04FF**

A *write* address that causes a TLB Hit and segmentation fault:

**Want TLB entry with  $V=1, W=0 \rightarrow$  VPN 0x20. Any address between 0x2000-0x20FF**

**Find the desired entry in the TLB. Because the TLB is fully-associative, the TLB tag is exactly the virtual page number (VPN). Any page offset within this page will access that TLB entry.**

## Question 5: Memory Allocation

```

1 #include <stdlib.h>
2 float pi = 3.14;
3
4 int main(int argc, char *argv[]) {
5     int year = 2019;
6     int* happy = malloc(sizeof(int*));
7     happy++;
8     free(happy);
9     return 0;
10 }

```

(A) Consider the C code shown above. Assume that the malloc call succeeds and happy and year are stored in memory (not in a register). Fill in the following blanks with “<” or “>” or “UNKNOWN” to compare the *values* returned by the following expressions just before return 0.

&year \_\_\_\_>\_\_ &main

happy \_\_\_\_<\_\_ &happy

&pi \_\_\_\_<\_\_ happy

(B) The code above has two memory-related errors. Use the line numbers in the code to describe what the errors are and where they occur.

**Error #1: On line 6 we are requesting more memory than we need. We should be requesting size of int (4 bytes), not size of int\* (8 bytes). Alternatively we could have meant to declare happy to be of type int\*\* (a pointer to a pointer to an int) so that we would have needed 8 bytes to hold a pointer to an int.**

**Error #2: On line 8 we are calling free on a pointer that was not the one returned to us by malloc. In line 7 we are incrementing happy (a pointer to an int that was returned to us by malloc).**

(C) (Not related to code at top of page) Give one advantage that next fit placement policy has over a first fit placement policy in an implicit free list implementation.

**Next fit searches the list starting where the previous search finished. This should often be faster than first fit because it avoids re-scanning unhelpful blocks. First fit always starts searching at the beginning of the list. In an implicit free list this is particularly bad because the “free” list actually contains all allocated blocks as well as free blocks. So starting from the beginning of the list is likely to traverse many allocated blocks each time.**

(D) List two reasons why it would be hard to write a garbage collector for the C

programming language.

Reason #1: **Pointers in C can point to a location other than the beginning of a block of memory on the heap.**

Reason #2: **In C you can “hide” pointers e.g. by casting them to longs.**

## Question 6: Java vs. C

This is an open-ended question, just make sure to avoid repeating the same point but worded differently or listing the same point but worded in opposite ways for both questions.

(A) Describe two distinct ways or things that you think C does better than Java.

**Possible Answer Topics: Pointer Manipulation, Memory Usage/Management, Compiles Faster/ Runs Faster, Closer to Machine Code (Lower Level), Programmer has more direct control**

(B) Describe two distinct ways or things that Java does better than C.

**Possible Answer Topics: Safety (e.g. type casting, bounds checking, exception handling, simpler code), Higher Level (less responsibility for the programmer), Data Structures (also OOP, Classes), More Portable, References vs. Pointers**