University of Washington — Computer Science and Engineering

Spring 2026 - Instructor: Ruth Anderson - 2026-06-10

CSE 351 FINAL

Last Name:	
First Name:	
UWNetID:	@uw.edu
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade and a report to CSSC. (please sign)	

Do not turn the page until 12:30.

Instructions

- **Fill out this page of the exam NOW**. Writing after time has been called will not be allowed and will be reported to CSSC.
- This exam contains 18 pages, including this cover page. Show scratch work for potential partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the exam.
- The exam is closed book (no laptops, tablets, wearable devices). You are allowed two pages (double-sided) of handwritten notes. Scientific calculators are allowed.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully. Skip questions that are taking a long time.
- Read *all* questions first and start where you feel the most confident.
- Relax. You are here to learn. You can do it!

Question	1	2	3	4	5	6	7	8	9	10	Total
Possible Points	9	16	8	8	10	9	6	8	8	9	91

Q1: Structs and Arrays [9 pts]

Assume a 64-bit machine and the following C struct definitions:

```
typedef struct {
    char type[4];
    int horsepower;
} engine;

typedef struct {
    char brand[6];
    engine eng;
    short year;
    int** previous_services;
} car;
```

A) How many bytes of internal and external fragmentation does an instance of `car` use? What is its total size?

Internal Fragmentation:	External Fragmentation:	Total Size:
bytes	bytes	bytes

B) Is it possible to reorder the fields in `car` to reduce the size of an instance? If so, provide the reordering. If not, briefly explain why. (**DO NOT** reorder the fields in `engine`!)

Possible to reduce size? (CIRCLE ONE)	Yes	No
New Ordering / Justification:		

C) Below are two different options for implementing an array `arr` with multiple dimensions in C:

Option 1:

```
int arr[5][5] = {{/* some values */}, {/* some values */}, . . . };
```

Option 2:

```
int row0[5] = {/* some values */};
int row1[5] = {/* some values */};
int row2[5] = {/* some values */};
int row3[5] = {/* some values */};
int row4[5] = {/* some values */};
int* arr[5] = {row0, row1, row2, row3, row4};
```

Consider each line of code below, with the intended behavior in the comments. Which implementation option(s) would produce the intended behavior?

1)
```
// var1 holds the element at arr[2][0]
int var1 = arr[1][5];
```

Correct behavior (CIRCLE ONE):	Option 1	Option 2	Both	Neither

Justification (1-2 sentences):

2)
```
// var2 holds the element at arr[0][2]
int var2 = (arr + 2);
```

Correct behavior (CIRCLE ONE):	Option 1	Option 2	Both	Neither

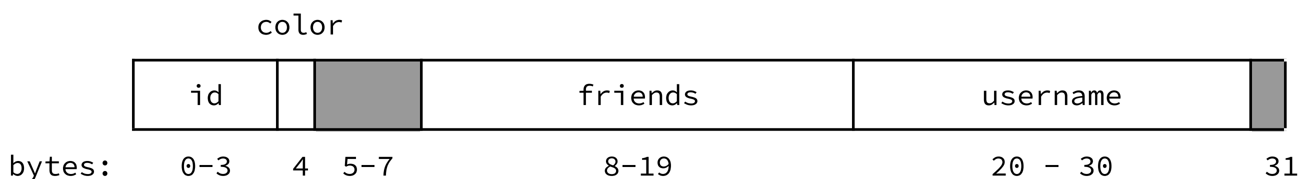
Justification (1-2 sentences):

Q2: Caches [16 pts]

You're a game developer creating a new online multiplayer game called Penguin Club. Each user has the following struct associated with their penguin:

```
struct penguin{
     int id; // 32 bit unique user id
     char color; // 8 bit color id
     int friends[3]; // array of maximum 3 friend user ids
     char username[11]; // 11 character username
};
```

It has the following layout (the shaded boxes represent fragmentation):



Suppose this game is in beta mode and only has 4 players right now. The players are stored in an array:

```
struct penguin players[4];  // starts at address 0x0
```

Assume the game servers run on the following cache, which starts **cold**:

Capacity: 64B
Associativity: 1
Block Size: 16B
Physical Memory Capacity: 1 KiB

A) Fill in the **number of bits** required to represent the following:

Address: _____ Cache Block Tag: _____ Cache Set Index: _____ Cache Block Offset: ___

B) Fill in the following information for `players[3].color` **in hex**:

Address: **0x** ____________________ Cache Block Tag: **0x** __________________

Cache Set Index: **0x** ______________ Cache Block Offset: **0x** ______________________

C) Assume the following lines of code compile and run correctly during the game:

```
1    players[0].friends[1] = players[2].id;
2    players[1].username = “x86lover94”;
3    players[2].color = 0x4f;
```

Answer questions about the access pattern on the following page →

Fill in the information for each access from the three lines of code on the previous page. Give the set each access maps to and color in exactly ONE option per access.

1. Line 1:
    a. `players[2].id` maps to set: _______
    ◯ hit ◯ cold miss ◯ conflict miss ◯ capacity miss

    b. `players[0].friends[1]` maps to set: _______
    ◯ hit ◯ cold miss ◯ conflict miss ◯ capacity miss

2. Line 2: `players[1].username` maps to set: _______
    ◯ hit ◯ cold miss ◯ conflict miss ◯ capacity miss

3. Line 3: `players[2].color` maps to set: _______
    ◯ hit ◯ cold miss ◯ conflict miss ◯ capacity miss

D) One of your friends suggests storing players differently. Instead of an array of `penguin` structs, they suggest a single struct containing arrays of an appropriate size:

```
struct penguin_arrays{
     int id[4];
     char color[4];
     int friends[4][3];
     char username[4][11];
};
```

Option 1: `struct penguin players[4]; //original array of structs`
Option 2: `struct penguin_arrays players; //your friend's struct of arrays`

1) In **one sentence**, describe an access pattern that would cause Option 1 to be more cache-efficient than Option 2:

2) In **one sentence**, describe an access pattern that would cause Option 2 to be more cache-efficient than Option 1:

Q3: Java & C [8 pts]

You have another friend who is trying to convince you to write Penguin Club in Java instead of C. She has written an equivalent implementation of the Penguin struct in Java shown below along with the original C version:

C Version:

```c
struct penguin {
    int id;
    char color;
    int friends[3];
    char username[11];
};

int main() {
    struct penguin p = {
        592,
        'b',
        {123, 456, 789},
        "PenguinPal"
    };
    int pal = p.friends[1];
};
```

Java Version:

```java
class Penguin {
    int id;
    char color;
    int[] friends;
    String username;

    public Penguin(int id, char color, int[]
friends, String username) {
        this.id = id;
        this.color = color;
        this.friends = friends;
        this.username = username;
    }
}

public class Main {
    public static void main(String[] args) {
        Penguin p = new Penguin(
            1,
            'b',
            new int[] {123, 456, 789},
            "PenguinPal"
        );
        int pal = p.friends[1];
    }
}
```

A) Is the **access time** of `int pal = p.friends[1];` **less** in the C implementation or the Java implementation? Why? CIRCLE ONE.

Less time with C struct | They are the Same | Less time with Java object

Explanation (1-2 sentences):

B) Give one **benefit** of writing the `friends` array in C and one **benefit** of writing the `friends` array in Java. **Do NOT use access time as your benefit.** Explain your answer.

Benefit of C implementation of the `friends` array (1-3 sentences):

Benefit of Java implementation of the `friends` array (1-3 sentences):

C) Which takes up **less** total space in memory, a `struct penguin` in C or a `Penguin` in Java? Why? CIRCLE ONE.

Less space with C struct | They are the Same | Less space with Java object

Explanation (1-2 sentences):

Q4: Dynamic Memory Allocation [8 pts]

A) Consider the following C code snippet. Assume that the length of `scores` is `NUM_LEVELS` and that the call to `malloc` succeeds.

```
 1  #include <stdlib.h>
 2  #include <stdio.h>
 3
 4  int NUM_LEVELS = 5;
 5  // Takes an array of scores for each level and computes a bonus
 6  // score for each level, and prints them along with the total.
 7  void print_bonus_scores(int* scores) {
 8      int* bonus = (int*) malloc(NUM_LEVELS * sizeof(int));
 9      int total = bonus[0];
10
11      for (int i = 0; i < NUM_LEVELS; i++) {
12          bonus[i] = scores[i] * 2;
13          total += bonus[i];
14      }
15
16      free(bonus);
17      for (int i = 0; i < NUM_LEVELS; i++) {
18          printf("Level %d bonus: %d\n", i + 1, bonus[i]);
19      }
20      printf("Total: %d\n", total);
21  }
```

This snippet has two <u>memory related bugs</u>. For each bug, identify the bug type, the line number, and describe the fix in one sentence.

Choose the bug type from the following list:

Bad pointer arithmetic	Dereferencng a non-pointer	Bad bounds checking
Using uninitialized memory	Memory leak	Accessing a freed block

	Bug 1	Bug 2
Bug type:		
Line number:		
Describe the fix (one sentence):		

B) Consider the C code shown here. Assume that the `malloc` call succeeds and that all variables are stored in memory (not registers). In the following groups of expressions, **CIRCLE the one** whose returned *value* is **smallest/lowest** immediately before the last line of main (`return 0`) is executed.

```
#include <stdlib.h>
long glob = 5;
char* str = "hello";

int main() {
    int* ip = malloc(16);
    int ONE = 1;
    char* msg = "bye";
    free(ip);
    return 0;
}
```

Group 1:	&ONE	ip	str	&main
Group 2:	glob	ONE	*str	&ip
Group 3:	&str	&ONE	ip	&ip
Group 4:	&msg	&glob	ip	&ONE

Q5: Processes [10 pts]

Consider the C function `doStuff()` below. Remember that `waitpid` pauses the caller's execution until the specified child exits. You can ignore the second and third arguments to `waitpid`.

```
void doStuff() {
  int x = fork(); // PID of child process is 1
  printf("%d ", x);
  int y = fork();

  if (y) {
    waitpid(y, NULL, 0);
    printf("%d ", x + 2);
  } else {
    printf("%d ", x + 4);
    exit(0);
  }
  exit(0);
}
```

A) For each of the following outputs, CIRCLE whether the output is possible or not possible. If an output is not possible, **CIRCLE the first invalid number** in the output.

"1 3 5 0 2 4 "	Possible	Not possible
"1 5 0 4 2 3 "	Possible	Not possible
"0 5 2 4 1 3 "	Possible	Not possible
"1 5 3 2 0 4 "	Possible	Not possible
"0 1 4 5 3 2 "	Possible	Not possible
"1 3 2 0 4 5 "	Possible	Not possible

**** Did you remember to **CIRCLE the first invalid number** on all impossible outputs?*

B) Consider a process *B*, which was interrupted at time t_1, then context switched back into at time t_2. Note that process *B* is not running between t_1 and t_2. CIRCLE ALL the items that **could** be different at times t_1 and t_2:

Contents of the Virtual Address Space of Process *B*	%rsp	Cache	Page Table Base Register	Management bits of Process B's page table

Q6: Virtual Memory [9 pts]

A) We're translating virtual page number x into a physical page number. For each scenario, fill in the next step in the translation process using one option from the table below. Multiple options may be possible per part, we only expect you to list **one**. Options may be used more than once.

(a) Check the Caches	(b) Check the Page Table
(c) Check the TLB	(d) Give control to the OS
(e) Update the TLB	(f) Re-run the instruction

1) If we get a TLB hit on x, then we ________
2) If we get a TLB miss on x, then we ________
3) If we find a valid entry for x in the Page Table, then we ________
4) If we don't find a valid entry for x in the Page Table, then we ________

B) For the following, fill in each of the blanks with one of the options listed below.

Disk   Page Table   TLB   RAM/Main Memory

1) The TLB acts as a cache for __________________
2) RAM/Main Memory acts as a cache for __________________
3) The Caches act as a cache for __________________

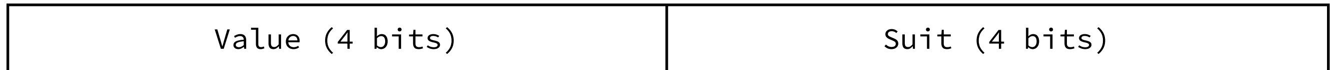
C) Why do we need to flush the TLB on a context switch?

Brief Explanation (2-3 sentences):

Q7: Number Representation and Bitwise Operators [6 pts]

In lecture, we saw several ways to encode a card from a standard deck of 52 playing cards, where each card can have one of 13 values and one of 4 suits. Here we consider another encoding:

`Card:`



Value (4 bits)	Suit (4 bits)

Here, `Value` is stored as an **unsigned** integer value, and `Suit` is stored using a representation called **one-hot encoding** (shown below). However, assume `C` treats `Card` as a **signed type**.

`Suit` field

Hearts	0b 0001
Diamonds	0b 0010
Clubs	0b 0100
Spades	0b 1000

For questions A) and B), **USE ONLY** bitwise operators (`&` , `|` , `^` , `~`), bitshifts (`<<` , `>>`), logical operators (`&&` , `||`, `!`), and constants (e.g. `351` , `0x123`).

A) Complete the function `value`, which returns the `Value` of a `Card`.

```
int value(Card c) {
      return (int)(________________________________________) ;
}
```

B) Complete the `suitIfEqual` function, which compares the `Suit`s of two `Card`s. If the cards share the same suit, it returns that suit as a one-hot encoding, with all 0s in the `Value` field. If the suits are different, it returns `0`.

```
int suitIfEqual(Card a, Card b) {

      return (int)(________________________________________) ;
}
```

Q8: Pointers and Memory [8 pts]

Consider the disassembly for the function `greet` located at the bottom of this page.
<u>Read the questions before reading the assembly code at the bottom of the page!</u>

A) What are the values (in **hex**) stored in each register after the following x86-64 instructions are executed? **Use the appropriate bit widths.** Write N/A if the value cannot be determined. Assume registers are initialized as shown below.

```
movswl  0x5(%rbx), %ecx
leaw    0x3(%rbx,%rdx,4), %si
```

Register	Value (in hex):
%rbx	0x0000 0000 0000 0108
%rdx	0x0000 0000 0000 0002
%ecx	**0x**
%si	**0x**

B) Complete the C code below to fulfill the behaviors described in the comments.

```
long* p = 0x100;

// set v1 = 0x7c83
short v1 = ((short*) p)[_______________];

// set v2 = 0x110
char* v2 = (char*) ((_______________________) p + 4);
```

<u>For the questions above, refer to the disassembly below!</u>

```
0000000000000100 <greet>:
  100:   48 83 ec 08              sub    $0x8,%rsp
  104:   80 7e 24 04              mov    %edi,0x4(%rsp)
  108:   83 7c 24 04 00           cmpl   $0x0,0x4(%rsp)
  10d:   7e 84                    jle    115 <greet+0x15>
  10f:   8b 44 24 04              mov    0x4(%rsp),%eax
  113:   83 e8 01                 sub    $0x1,%eax
  116:   eb 03                    jmp    11b <greet+0x1b>
  118:   b8 00 00 00 00           mov    $0x0,%eax
  11d:   48 83 c4 08              add    $0x8,%rsp
  121:   c3                       retq
```

Q9: C & Assembly [8 pts]

Consider the following x86-64 Assembly:

```
mystery:

      movl    $0, %eax
      cmpl    $1, %edi
      jle     .L4
.L1:
      testb   $1, %dil
      jne     .L2
      movslq  %edi, %rdi
      leal    1(%rdi,%rdi,2), %ecx
      movl    %ecx, %edi
      jmp     .L3
.L2:
      sarl    $1, %edi
.L3:
      addl    $1, %eax
      cmpl    $1, %edi
      jg      .L1
.L4:
      ret
```

A) Fill in the blanks in the following C function skeleton:

```
__________ mystery (__________ n) {

   int steps = 0;

   while (______________________) {

      if ( n % 2 == 0 ) {

         __________________________;

      } else {

         __________________________;

      }
      steps++;
   }

   return ____________ ;
}
```

Q10: Procedures & The Stack [9 pts]

Consider the following C code and disassembly for a recursive implementation of **binary search**. The function `algo()` takes in a sorted array, the range of indices to be searched between, and the value that is being searched for. It returns the index of the value if found or a -1 otherwise.

```c
int algo(int* arr, int lo, int hi, int val)
{
    if (lo > hi) {
        return -1;
    }
    int mid = lo + (hi - lo) / 2;
    if (arr[mid] == val) {
        return mid;
    }
    if (val < arr[mid]) {
        return algo(arr, lo, mid - 1, val);
    }
    return algo(arr, mid + 1, hi, val);
}
```

```c
int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int res = algo(arr, 0, 4, 7);
    printf("result: %d\n", res);
    return 0;
}
```

```
0000000000401126 <algo>:
401126:   cmp    %edx,%esi
401128:   jg     401170 <algo+0x3a>
40112a:   sub    $0x8,%rsp
40112e    mov    %edx,%r8d
401131:   sub    %esi,%r8d
401134    mov    %r8d,%eax
401137:   shr    $1,%eax
401139:   add    %esi,%eax
40113b:   movslq %eax,%r8
40113e:   mov    (%rdi,%r8,4),%r8d
401142:   cmp    %ecx,%r8d
401145:   je     401151 <algo+0x2b>
401147:   jg     401156 <algo+0x30>
401149:   lea    0x1(%rax),%esi
40114c:   call   401126 <algo>
401151:   add    $0x8,%rsp
401155:   ret
401156:   lea    -0x1(%rax),%edx
401159:   call   401126 <algo>
40116e:   jmp    401151 <algo+0x2b>
401170:   mov    $0xffffffff,%eax
401175:   ret
```

A) List up to four possible return addresses that can be stored on the stack when `algo()` makes a **recursive** call to itself. You do **not** need to use all the given blanks if less than four are possible. **Answer in hex**.

0x________________ **0x**________________ **0x**________________ **0x**________________

B) Which stack frame portions exist in a recursive call to `algo()` that will also make a recursive call to itself? CIRCLE ONE PER ROW.

1) Return Address?	Exists	Does not exist
2) Callee-Saved Register Values?	Exists	Does not exist
3) Padding and Local Variables?	Exists	Does not exist
4) Caller-Saved Register Values?	Exists	Does not exist
5) Argument Build?	Exists	Does not exist

C) `algo()` can be rewritten to use a loop instead of recursion. Would using a loop reduce `algo()`'s memory usage, either in terms of total bytes of memory used, or number of memory reads/writes? Briefly justify your choice in 1-2 sentences.

Reduces memory usage? (CIRCLE ONE)	Yes	No
Justification (1-2 sentences):		

This page intentionally left blank! Enjoy!!!