University of Washington – Computer Science & Engineering

Autumn 2025 Instructors: Justin Hsia, Amber Hu 2025-12-10

CSE351 FINAL

Last Name:	Perfect	
First Name:	Perry	
Student ID Number:	1234567	
Name of person to your Left | Right	Sidney Student	Leslie Learner
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. **(please sign)**		

Do not turn the page until 12:30.

Instructions

- This exam contains 14 pages, including this cover page. Show scratch work for 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 rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices). You are allowed two pages (US letter, 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 before starting. 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.

Question	M1	M2	M3	M4	M5	F6	F7	F8	F9	F10	Total
Possible Points	8	4	17	24	18	12	18	14	20	15	**150**

Question M1: Number Representation [8 pts]

Suppose we're writing a program that records **monetary transactions**. Each transaction is recorded in the following 32-bit representation scheme (U = unsigned, S = signed/two's complement):

```
        +----------+----------+----------+----------+
        |   Year   |  Month   |    Day   |  Amount  |
        +----------+----------+----------+----------+
bits:     31-23(U)   22-19(U)   18-14(U)    13-0(S)
```

(A) What is the **maximum** (*i.e.*, most positive) **transaction amount**? You may leave your answer unsimplified. [2 pt]

TMax = $2^{n-1}-1$, n = 14 for Amount

$2^{13}-1$ = 8,191

(B) Our year field will encode "years since 2000" (*e.g.*, 0x0 → year 2000, 0x1 → year 2001). **What is the *first* year after 2000 that we *cannot* represent?** [2 pt]

Max Year encoding (9 bits) is 0x1FF = 511 → year 2511

2512

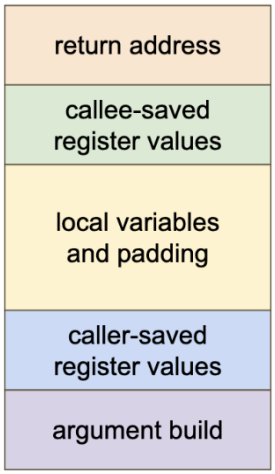
(C) Fill in the C expression that completes the function to **extract the month field value from txn:** [4 pt]

```
// The returned unsigned int should have the least 4 significant
// bits set to the month and the rest set to 0.
unsigned int get_month(unsigned int txn) {

    return (txn >> 19) & 0xF;
}   // (txn & mask) >> 19 also accepted with mask 0x780000-0x7FFFFF
    // (txn << 9) >> 28 also accepted
```

Question M2: Design Question [4 pts]

We introduced the different portions of a stack frame. *Briefly* explain why the **caller-saved registers** must be found in the position/order given.



Why after the *local variables and padding*?

Caller-saved registers are saved right before making a function call in order to preserve their most up-to-date values (and make it easier to restore later via a pop). Local variables should already be available for computations prior to the function call.

Why before the *argument build*?

Argument build must be last (right above the callee's return address) so that arguments 7+ can be found by the callee at consistent offsets.

Student ID Number: ___1234567___

Question M3: Pointers & Memory [17 pts]

For this problem, we are using a 64-bit x86-64 machine (**little endian**). A partial view of the current state of memory (values in hex) is shown below, along with a snippet of the ASCII table.

```
// "uw" is shown in memory
char* cp = "uw";
long  lg = 0x351;
short ar[5];
```

Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
0xA0	33	00	75	77	00	74	6F	70
0xA8	00	CE	BE	9F	50	24	9D	D6
0xB0	5A	96	8E	5E	EF	5C	DF	9E
0xB8	D9	81	3E	43	34	DE	D7	CB
0xC0	B1	6B	AA	FB	BD	0B	7E	67

Partial ASCII table:

'n'	'o'	'p'	'q'	'r'	's'	't'	'u'	'v'	'w'	'x'	'y'	'z'
0x6E	0x6F	0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77	0x78	0x79	0x7A

(A) **How many bytes are allocated** by the declarations and initializations in the C code given above? [4 pt]

8 (char*) + 3 (string literal) + 8 (long) + 5*2 (short)

29 bytes

(B) **What is the output of `printf("%s", cp+3)`** – printing `cp+3` as a C string? [3 pt]

Start reading at address `0xA2+3=0xA5` and move to increasing addresses until null character (`0x00`) encountered at `0xA8`.

top

(C) Assume `ar` is stored at address `0xF0` (not shown in memory), and consider the following C expressions. **Provide the C type and value** (no leading zeros). [10 pt]

C Expression	C Type	Hex Value
&ar[-8]	**short***	0x **E0**
*cp + 5	**char**	0x **7A**

`&ar[-8]` uses pointer arithmetic, so we scale by 2, making the subtraction 8*2 = 16 = 0x10. `&ar[0]` is given as `0xF0`.

`*cp + 5` uses normal arithmetic, since we are grabbing the `char` pointed to (`'u' = 0x75`). Don't forget that we are using hex, so `0x75 + 5 = 0x7A`, not `0x80`.

Question M4: Procedures & The Stack [24 pts]

The recursive function `run_back_sum()` and its x86-64 disassembly are shown below. This function recursively computes the sum of an array of `long`s and replaces the array values with the running partial sum, in reverse order. For example, `run_back_sum({1, 2, 3}, 3)` returns 6 and modifies the array to be `{6, 5, 3}`.

```
long run_back_sum (long* addr, int len) {
  if (len == 0)
    return 0;
  long elem = *addr;
  *addr = elem + run_back_sum(addr+1, len-1);
  return *addr;
}
```

```
#:  0000000000401136 <run_back_sum>:
1     401136:   b8 00 00 00 00     mov    $0x0,%eax
2     40113b:   85 f6              test   %esi,%esi
3     40113d:   75 01              jne    401140 <run_back_sum+0xa>
4     40113f:   c3                 ret
5     401140:   55                 push   %rbp
6     401141:   53                 push   %rbx
7     401142:   48 83 ec 08        sub    $0x8,%rsp
8     401146:   48 89 fb           mov    %rdi,%rbx
9     401149:   48 8b 2f           mov    (%rdi),%rbp
10    40114c:   83 ee 01           sub    $0x1,%esi
11    40114f:   48 8d 7f 08        lea    0x8(%rdi),%rdi
12    401153:   e8 de ff ff ff     call   401136 <run_back_sum>
13    401158:   48 01 e8           add    %rbp,%rax
14    40115b:   48 89 03           mov    %rax,(%rbx)
15    40115e:   48 83 c4 08        add    $0x8,%rsp
16    401162:   5b                 pop    %rbx
17    401163:   5d                 pop    %rbp
18    401164:   c3                 ret
```

(A) What is the **return address to `run_back_sum`** that gets stored on the stack? Answer in hex. [2 pt]

Address of the instruction *after* the `call` instruction.

`0x` **401158**

(B) The 18 assembly instructions are numbered on the left. Below, **list *only* the instruction numbers that access memory**, separated by commas. [4 pt]

`ret` and `pop` pop values off the stack; #9 has Mem src.

`call` and `push` push values onto the stack; #14 has Mem dst.

READ from memory: **4, 9, 16, 17, 18**

WRITE to memory: **5, 6, 12, 14**

(C) **What values are saved across each recursive call?** Answer in terms of the C code. [4 pt]

addr (in %rbx)	**elem = *addr** (in %rbp)

(D) **How big is a `run_back_sum` stack frame** on a recursive (*i.e.*, not base case) call? [2 pt]

ret addr + saved %rbp + saved %rbx + sub 8

32 bytes

(E) Which stack frame portions exist in a recursive call to `run_back_sum`? **Write "Y" or "N".** [2 pt]

From the Part D answer, %rbp and %rbx are callee-saved registers and the sub to %rsp is padding for stack frame size alignment.

Return address	__Y__
Callee-saved register values	__Y__
Local variables and padding	__Y__
Caller-saved register values	__N__
Argument build	__N__

(F) Assume `main` calls `run_back_sum({4, 2}, 2)`. **How many `run_back_sum` stack frames are created?** [2 pt]

({4, 2}, 2) → ({2}, 1) → (<off end>, 0).

3 frames

(G) Assume `main` calls `run_back_sum({4, 2}, 2)`, with the array at address `0x100`. **Fill in the memory snapshot below** as this call to `run_back_sum` returns to `main`. The number of boxes is arbitrary; for unknown words, write "`0x unknown`". [8 pt]

From Part D, we see that the recursive stack frames are ordered ret addr, saved %rbp, saved %rbx, 8 bytes padding. It is important to note that %rbp is *saved* first (instr #5), despite being *set* second (instr #9).

From Part C, we know that %rbp holds elem and %rbx holds addr.

0x7fffffffe008	<ret addr to main>	({4, 2}, 2)
0x7fffffffe000	0x **unknown** (old %rbp)	
0x7fffffffdff8	0x **unknown** (old %rbx)	
0x7fffffffdff0	0x **unknown** (padding)	
0x7fffffffdfe8	0x **401158**	({2}, 1)
0x7fffffffdfe0	0x **4** (old %rbp)	
0x7fffffffdfd8	0x **100** (old %rbx)	
0x7fffffffdfd0	0x **unknown** (padding)	
0x7fffffffdfc8	0x **401158**	(<off end>, 0)

Question M5: C & Assembly [18 pts]

Answer the questions below about the following x86-64 assembly function:

```
mystery:
        cmpl    $1, %esi                # Line 1
        jg      .L7                     # Line 2
        ret                             # Line 3
.L7:    movslq  %esi, %rax              # Line 4
        leaq    -1(%rdi,%rax), %rax     # Line 5
        movzbl  (%rax), %edx            # Line 6
        movzbl  (%rdi), %ecx            # Line 7
        movb    %cl, (%rax)             # Line 8
        movb    %dl, (%rdi)             # Line 9
        subl    $2, %esi                # Line 10
        addq    $1, %rdi                # Line 11
        call    mystery                 # Line 12
        ret                             # Line 13
```

`mystery` takes two arguments and contains an `if`-statement. **Fill in the C function skeleton below**, using the provided variable names `arr`, `len`, and `c`. Hint: Recall that `arr[i]` ⇔ `*(arr+i)`.

```
void mystery (char* arr, int len) {
    if ( len > 1 ) {                        // Lines 1-2
        char c = arr[len-1];                // Lines 4-6
        arr[len-1] = arr[0];                // Lines 7-8
        arr[0] = c;                         // Line  9
        mystery(arr + 1, len - 2);          // Lines 10-12
    }
}
```

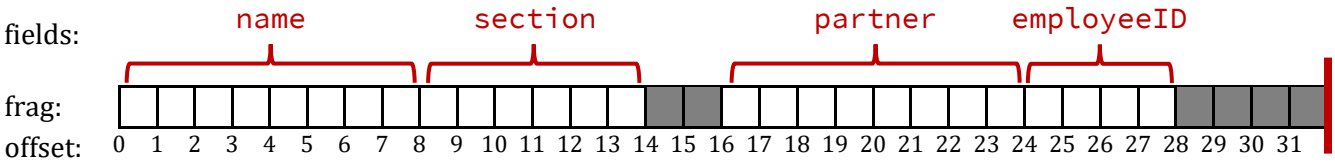
- Arguments: `%rdi` (`arr`) used in memory operand in `movb` (Line 9), so `char*`. `%esi` (`len`) is only form of `%rsi` used in code (and name `len` hints it's an integer).
- Lines 1-2: We jump *into* the body when `len` – 1 > 0 → `len > 1`.
- Lines 4-6: `c` is given as the assignment target, which is `%edx` in the assembly. Line 4 puts `%esi` (`len`) into `%rax`, which is then updated to `arr + len – 1` in Line 5. This address is then dereferenced in Line 6, with `*(arr+len-1)` = `arr[len-1]`, using `char*` scaling factor of 1.
- Lines 7-8: `*arr` = `arr[0]` is read in Line 7 and then immediately stored back at the address in `%rax`, which is still `arr[len-1]` from Line 5.
- Line 9: `%dl` is the lowest byte of `%edx`, which still corresponds to `c` (Line 6).

Question F6: Structs [12 pts]

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

```
struct TA {
  char* name;           // e.g., "Rose"
  char section[6];      // e.g., "AE/CE"
  struct TA* partner;
  int employeeID;
};
```

(A) The boxes below represent the bytes in an instance of `struct TA`, starting with offset 0 on the left. **Label each field above the boxes with a bracket and name. Shade in the boxes representing fragmentation.** Draw a thick vertical line at the end of the struct instance; if there are extra unused boxes at the end, leave them blank. [6 pt]



(B) **Can we reorder the fields in `struct TA` to reduce the size of an instance?** If yes, provide a field ordering; if not, briefly explain why not. [4 pt]

No. $K_{max} = 8$ from name/partner and the fields sum to 8+6+8+4 = 26 bytes without fragmentation, so this will always get rounded up to at least 32 due to alignment requirements.

(C) *Briefly* give one reason why adding external fragmentation in struct instances is good/beneficial. [2 pt]

Some acceptable answers (others may exist):

- Allows for arrays of structs to be properly aligned one after the other.
- Any following data is more likely to be aligned for caching/performance reasons.
- Provides "nice" struct instance sizes (*e.g.*, not 17 bytes).

Question F7: Caching [18 pts]

We have an L1 data cache with the following settings:

- 128-byte cache size
- direct-mapped
- 32-byte blocks
- LRU replacement
- write-back and write allocate policies

The width of a physical address is ≥ 8 bits, but its exact length is not important. This code snippet swaps the elements of **two *adjacent* arrays** of `doubles` of size `LEN`.

```
#define LEN 6
double temp;
double A[LEN];  // &A[0] = 0x60
double B[LEN];  // &B[0] = 0x90
for (int i = 0; i < LEN; i++) {
  temp = A[i];
  A[i] = B[i];
  B[i] = temp;
}
```

Assume `i` and `temp` are stored in registers and no elements of either array are in the cache at the start. Answer the following questions to analyze the code:

(A) **How many array elements per cache block**? [1 pt]

32 bytes per block and 8 bytes per `double`.

4

(B) **How wide** (in bits) **are the block offset and index fields**? [2 pt]

$k = \log_2 K = \log_2 32 = 5, s = \log_2((C/K)/E) = \log_2 4 = 2.$

index: **2** bits	block offset: **5** bits

(C) **Which set and offset** (in binary) do `A[0]` and `B[0]` map into? [4 pt]

`&A[0] = 0x60 = 0b 0|11|0 0000`

`&B[0] = 0x90 = 0b 1|00|1 0000`

A[0] set: `0b` **11**	A[0] offset: `0b` **0 0000**
B[0] set: `0b` **00**	B[0] offset: `0b` **1 0000**

(D) **What is the memory access pattern** of *one* iteration of the `for`-loop (*e.g.*, RA for "read from A", WB for "write to B")? Separate accesses with commas (*e.g.*, "RA, WB"). [2 pt]

read `A[i]` (RHS), read `B[i]` (RHS), write `A[i]` (LHS), write `B[i]` (LHS)

RA, RB, WA, WB

(E) **How many cache misses** occur in the execution of this *entire* code snippet? [3 pt]

The arrays `A` and `B` span 3 *adjacent* cache blocks, with `B` starting in the middle of the 2nd block. This means that each block maps into a different set: 3, 0, and 1, respectively. Therefore, they never conflict, and we only have compulsory misses, one per block.

3

(F) For each of the proposed (independent) changes, draw ↑ for "increased", — for "no change", or ↓ for "decreased" to indicate the **effect on the number of misses from Part E**: [6 pt]

3 blocks now map to sets 1, 0, 1, but can co-exist in set 1.	Make 2-way set associative	—
Only spans 2 cache blocks, reducing compulsory misses.	Double the block size (64 B)	↓
No write misses in code (always read from index first).	No-write allocate	—

Question F8: Memory Allocation [14 pts]

(A) 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* (assume just before `return 0`) is **smallest/lowest**. [6 pt]

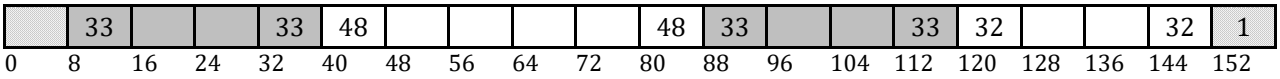
```
#include <stdlib.h>
long TWO = 2;
struct node* head = NULL;

int main() {
  int* ip = malloc(88);
  char* str = "351";
  free(ip);
  return 0;
}
```

Group 1:	ip	&str	**&TWO**
Group 2:	**head**	*str	TWO
Group 3:	&head	&ip	**str**

7) `&ip/&str` (Stack)
6) `ip` (Heap)
5) `&head/&TWO` (Static Data)
4) `str` (Literals)
3) `*str` (`'3' = 0x33`)
2) `TWO` (2)
1) `head` (`NULL = 0`)

(B) We are using a dynamic memory allocator on a 64-bit machine with an explicit free list, 8-byte boundary tags, and 16-byte alignment. There is no prec-used? tag, so both allocated and free heap blocks have a footer. Answer the following questions, given the current state of the heap shown below.



i. What is the **minimum block size** of this allocator? [2 pt]

alloc: head + foot + payload (>0); free: head + foot + 2 ptr; multiple of 16

32 bytes

ii. If the shown allocated blocks were the result of calls to `malloc(16)` and `malloc(12)`, how much **internal fragmentation** is there in the heap? [2 pt]

Internal frag is everything but payload, so 32 + 32 – 12 – 16 = 36.

36 bytes

iii. Assuming a *best-fit allocation strategy*, what would be the **returned value from a call to `malloc(24)`**? [2 pt]

Necessary block size: 24 + 8 + 8 = 40, rounded up to 48 for alignment.
Allocation strategy is irrelevant as only one free block is big enough.
Return value of `malloc()` is the address of the *payload*.

48

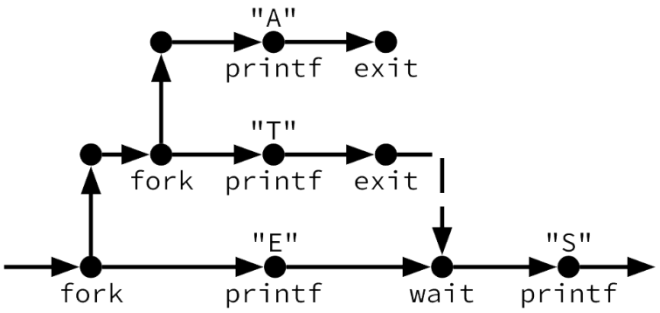
iv. Assuming the `malloc` from (iii) hasn't taken place, which argument to `free()` **yields the largest free block afterward**? [2 pt]

Want the largest coalesced block, so choose the one in-between free blocks.
`free()` takes address of payload.

96

Question F9: Processes [20 pts]

(A) For the following process diagram, **determine whether or not each output is possible** (circle one). For impossible outputs, **circle the first invalid letter in the sequence**. *Spaces have been added for readability, but should not be considered when determining the validity of outputs.* [7 pt]



i. "A T E S" Possible / Impossible

ii. "E A S T" Possible / Impossible

iii. "S E A T" Possible / Impossible

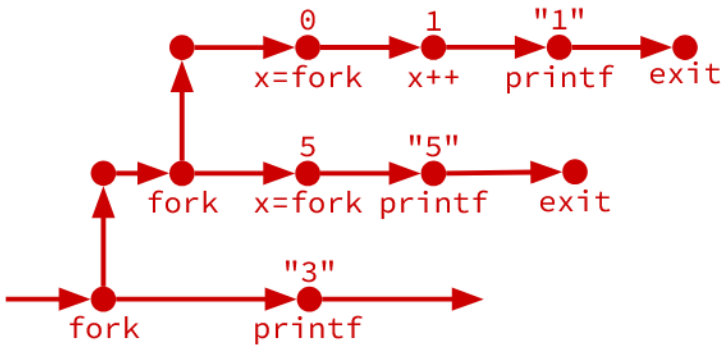
iv. "T E S A" Possible / Impossible

Restriction imposed by `wait`: `"S"` must come after `"T"`. Note that `"A"` is not similarly restricted because `wait` does not interact with/does not know about grandchildren.

(B) Following Part A's formatting, **draw a process diagram for the following code snippet**. Assume that the original process has a PID of 3 and that PIDs are assigned sequentially (*i.e.*, the next processes will have PIDs 4, 5, 6, etc.). [9 pt]

```
int x;
if (fork()) {
  printf("%d", getpid());
} else {
  x = fork();
  if (x)
    printf("%d", x);
  else {
    x++;
    printf("%d", x);
  }
  exit(0);
}
```

Draw here:



(C) *Briefly* explain how the following enable each process in Part B to maintain a distinct x value. [4 pt]

`fork`: Creates a new, separate process from the parent, with separate address space. Each process' x variable lives in a different address space.
Page tables: Each process, despite having parent-child relationships, are prevented from accessing each other's address spaces because they each have separate page tables that map only to pages for that particular process.

Question F10: Virtual Memory [15 pts]

Our system has the following setup:

- 16-bit virtual addresses and 4 KiB of RAM with 256-byte pages
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)
- A 4-entry, 2-way set associative TLB with LRU replacement in the state shown (permission bits: 1 = allowed, 0 = disallowed)

TLBT	Valid	D	R	W	X	PPN
0x05	1	1	1	1	0	0xA
0x7F	1	1	1	1	0	0xB
0x7F	0	0	1	1	0	0xE
0x02	1	0	1	0	1	0xD

(A) Compute the following system values: [6 pt]

page offset width 8 bits____ # of physical address bits 12 bits__

# of PTEs in a page table $2^8 = 256$ TLBT field width 7 bits____

Page offset is $\log_2 256 =$ **8 bits** wide. # of physical address bits is $\log_2$ (4 KiB) = **12 bits**.
1 PTE for every VPN, so 2^{16} bytes of virtual address space / 256-byte page = 2^8 = **256 PTEs**.
VPN width is $n - p = 8$ bits, 2 sets in TLB means TLB tag width is $8 - \log_2 2 =$ **7 bits**.

(B) Given the current TLB state, what will happen if we try to **write to the address `0x05ED`**? Circle one item in each row. [3 pt]

TLB:	(TLB Hit) / TLB Miss / Neither	Split address: `0b0000010`|`1`|`1110 1101`
Page Table:	Page Table Hit / Page Fault / (Neither)	TLBI = 0b1, TLBT = 0x02 → TLB Hit!
Data Access:	Cache Access / (Segmentation Fault)	Page table never accessed on TLB Hit. Segfault because no write access to page.

(C) Given the current TLB state, give the **smallest address that results in accessing physical page 10**. [3 pt]

PPN 10 (`0xA`) found in top line of TLB Set `0` with TLBT of `0x05`. We get our VPN by putting these together: `0b0000101|0`. Smallest address has page offset of all `0`'s, leading to: `0b0000 1010|0000 0000 = 0x0A00`.

`0x 0A00`

(D) Circle one of the options below for each row: [3 pt]

Smallest storage capacity:	Caches	Disk	RAM	(Registers)
Slowest access time:	Caches	(Disk)	RAM	Registers
Updated on a context switch:	Caches	Disk	RAM	(Registers)

Smallest capacity is highest in the memory hierarchy. Slowest access time is lowest in the memory hierarchy. Register values are specific to the currently running process, so must be updated on a context switch; the others use physical addressing and are protected by address translation.