

18au Final

Question M1: Numbers [16 pts]

- (A) *Briefly* explain why we know that there may be data loss **casting** from `int` to `float`, but there won't be casting from `int` to `double`. [4 pt]

<u>Explanation:</u>

- (B) What value will be read after we try to store $-2^{127} - 2^{104}$ in a `float`? (Circle one) [4 pt]

-2^{127} `-NaN` $-\infty$ $-2^{127} - 2^{104}$

- (C) Complete the following C function that returns whether or not a pointer `p` is aligned for data type size `K`. Hint: be careful with data types! [4 pt]

```
int aligned(void* p, int K) {  
    return _____;  
}
```

- (D) Take the 32-bit numeral `0x50000000`. Circle the number representation below that has the *most positive* value for this numeral. [4 pt]

Floating Point Two's Complement Unsigned Two's AND Unsigned

Question M2: Design Question [4 pts]

- (A) If the Stack grew upwards (*e.g.* we switched the positions of the Stack and Heap), which assembly instructions would need their behaviors changed? Name two and briefly describe the changes.

<u>Instruction 1:</u> <u>Change:</u>
<u>Instruction 2:</u> <u>Change:</u>

17sp Midterm

4. Pointers, Memory & Registers (14 points)

Assuming a 64-bit x86-64 machine (little endian), you are given the following variables and initial state of memory (values in hex) shown below:

Address	+0	+1	+2	+3	+4	+5	+6	+7
0x00	AB	EE	1E	AC	D5	8E	10	E7
0x08	F7	84	32	2D	A5	F2	3A	CA
0x10	83	14	53	B9	70	03	F4	31
0x18	01	20	FE	34	46	E4	FC	52
0x20	4C	A8	B5	C3	D0	ED	53	17

```
int* ip = 0x00;  
short* sp = 0x20;  
long* yp = 0x10;
```

- a) Fill in the type and value for each of the following C expressions. If a value cannot be determined from the given information answer UNKNOWN.

Expression (in C)	Type	Value (in hex)
<code>yp + 2</code>		
<code>*(sp - 1)</code>		
<code>ip[5]</code>		
<code>&ip</code>		

- b) Assuming that all registers start with the value 0, except `%rax` which is set to 0x4, fill in the values (in hex) stored in each register after the following x86 instructions are executed. Remember to give enough hex digits to fill up the width of the register name listed.

```
movl 2(%rax), %ebx  
leal (%rax,%rax,2), %ecx  
movsbl 4(%rax), %edi  
subw (,%rax,2), %si
```

Register	Value (in hex)
<code>%rax</code>	0x0000 0000 0000 0004
<code>%ebx</code>	
<code>%ecx</code>	
<code>%rdi</code>	
<code>%si</code>	

18wi Final

Question 6: Procedures & The Stack [24 pts.]

Consider the following x86-64 assembly and C code for the recursive function rfun.

```
// Recursive function rfun
long rfun(char *s) {
    if (*s) {
        long temp = (long)*s;
        s++;
        return temp + rfun(s);
    }
    return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
    char *s = "Yay!";
    long r = rfun(s);
    printf("r: %ld\n", r);
}
```

```
0000000004005e6 <rfun>:
 4005e6: 0f b6 07                movzbl (%rdi),%eax
 4005e9: 84 c0                   test  %al,%al
 4005eb: 74 13                   je    400600 <rfun+0x1a>
 4005ed: 53                      push  %rbx
 4005ee: 48 0f be d8            movsbq %al,%rbx
 4005f2: 48 83 c7 01            add  $0x1,%rdi
 4005f6: e8 eb ff ff ff        callq 4005e6 <rfun>
 4005fb: 48 01 d8              add  %rbx,%rax
 4005fe: eb 06                 jmp  400606 <rfun+0x20>
 400600: b8 00 00 00 00        mov  $0x0,%eax
 400605: c3                     retq
 400606: 5b                     pop  %rbx
 400607: c3                     retq
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pts.]

(B) The compiler automatically creates labels it needs in assembly code. How many labels are used in `rfun` (including the procedure itself)? [2 pts.]

(C) In terms of the C function, what value is being saved on the stack? [2 pts.]

(D) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)? [2 pts.]

(E) Assume `main` calls `rfun` with `char *s = "Yay!"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack? [2 pts.]

Total Frames:	Max Depth:
---------------	------------

(F) Assume main calls rfun with `char *s = "Yay!"`, as shown in the C code. After main calls rfun, we find that the return address to main is stored on the stack at address `0x7fffffffdb38`. On the first call to rfun, the register `%rdi` holds the address `0x4006d0`, which is the address of the input string "Yay!" (i.e. `char *s == 0x4006d0`). Assume we stop execution prior to executing the `movsbq` instruction (address `0x4005ee`) during the **fourth** call to rfun. [14 pts.]

For each address in the stack diagram below, fill in both the **value** and a **description** of the entry.

The **value** field should be a hex value, an expression involving the C code listed above (e.g., a variable name such as `s` or `r`, or an expression involving one of these), a literal value (integer constant, a string, a character, etc.), "unknown" if the value cannot be determined, or "unused" if the location is unused.

The **description** field should be one of the following: "Return address", "Saved %reg" (where reg is the name of a register), a short and descriptive comment, "unused" if the location is unused, or "unknown" if the value is unknown.

Memory Address	Value	Description
0x7fffffffdb48	unknown	%rsp when main is entered
0x7fffffffdb38	0x400616	Return address to main
0x7fffffffdb30	unknown	original %rbx
0x7fffffffdb28		
0x7fffffffdb20		
0x7fffffffdb18		
0x7fffffffdb10		
0x7fffffffdb08		
0x7fffffffdb00		

18sp Midterm

4. (15 points) (x86-64 Assembly) This problem considers this assembly implementation of a C function of the form `long mystery(long x) { ... }`

```
mystery:
    movq    $0, %rax
    testq   %rdi, %rdi
    jle    .L2
.L1:
    addq   %rdi, %rdi
    addq   $1, %rax
    testq   %rdi, %rdi
    jg     .L1
.L2:
    ret
```

In parts (a)-(c) we ask you to modify the assembly code in ways that have *no effect* on the answers it produces, i.e., it should perform the same overall computation after any of your changes.

- (a) Give a use of a `cmpq` instruction that could be used instead of either of the `testq` instructions.

- (b) Give a use of a `shlq` instruction could be used instead of one of the `addq` instructions and indicate which instruction it is replacing.

- (c) Suppose we replace the `jle .L2` with `jg .L1`. Insert an additional instruction to complete this change correctly: indicate what instruction you are adding and where.

Now we ask about what `mystery` is actually computing.

- (d) Complete this description of what `mystery` computes with 1–2 English sentences: “It takes the number in `%rdi` and returns...”.

- (e) What is the largest number `mystery` could possibly return? Answer in base-10.

18sp Final

2. (12 points) (Struct Layout) Assume x86-64 and Linux and that all fields should be properly aligned.

(a) Consider this `struct` definition:

```
struct S {
    int x;
    int * p;
};
```

Do all of the following:

- Indicate what `sizeof(struct S)` would evaluate to.
- Draw the layout of the struct, indicating the size and offset of each field.
- For any padding, indicate whether it is in *internal* or *external* fragmentation.

(b) Repeat the previous problem for this `struct` definition:

```
struct S {
    int x;
    int * p;
    int y;
};
```

(c) Repeat the previous problem for this `struct` definition:

```
struct S {
    int * p;
    int x;
    int y;
};
```

15wi Final

6 Pointers, arrays and structs (10 points)

Consider the following variable declarations, assuming x86_64 architecture:

```
typedef struct {  
    int a;  
    char b;  
    double c;  
} struct_type;
```

```
struct_type* m;  
struct_type n[2];
```

Errata: the struct declaration was meant to include a typedef. As it was written, "struct_type" would be a variable of the unnamed struct type, rather than a new name for the struct.

Fill in the following table:

C Expression	Evaluates to?	Resulting data type
m	0x10000000	
n	0x20000000	
&(m->a)		
&(m->b)		
&(m->c)		
sizeof(struct_type)		
sizeof(*m)		
sizeof(m)		
&(n[0])		
&(n[0].a)		
&(n[1].a)		

17sp Final

1. Caches (11 points)

You are using a byte-addressed machine where physical addresses are 22-bits. You have a 4-way associative cache of total size 1 KiB with a cache block size of 32 bytes. It uses LRU replacement and write-back policies.

a) Give the number of bits needed for each of these:

Cache Block Offset: _____ Cache Tag: _____

b) How many sets will the cache have? _____

c) Assume that everything except the array \mathbf{x} is stored in registers, and that the array \mathbf{x} starts at address 0x0. Give the hit rate (as a fraction or a %) for the following code, assuming that the cache starts out empty. Also give the total number of hits.

```
#define LEAP 1
#define SIZE 256
int x[SIZE][8];
... // Assume x has been initialized to contain values.
... // Assume the cache starts empty at this point.
for (int i = 0; i < SIZE; i += LEAP) {
    x[i][0] += x[i][4];
}
```

Hit Rate: _____ Total Number of Hits: _____

d) If we increase the cache block size to 64 bytes (and leave all other factors the same) what would the hit rate be?

Hit Rate: _____ Total Number of Hits: _____

e) For each of the changes proposed below, indicate how it would affect the hit rate of the code above in part c) *assuming that all other factors remained the same* as they were in the original cache:

Change associativity from
4-way to 2-way: increase / no change / decrease

Change **LEAP** from
1 to 4: increase / no change / decrease

Change cache size from
1 KiB to 2 KiB: increase / no change / decrease

14au Final

2. Caches – 35 pts total (14/A, 6/B, 15C)

- A. You are given a direct-mapped cache of total size 256 bytes, with cache block size of 16 bytes. The system's page size is 4096 bytes. The following C array has been declared and initialized to contain some values:

```
int x[2][64];
```

- i. How many sets will the cache have?
- ii. How many bits will be required for the cache block offset?
- iii. If the physical addresses are 22 bits, how many bits are in the cache tag?
- iv. Assuming that all data except for the array **x** are stored in registers, and that the array **x** starts at address 0x0. Give the miss rate (as a fraction or a %) and total number of misses for the following code, assuming that the cache starts out empty:

```
int sum = 1;
int i;
for (i = 0; i < 64; i++) {
    sum += x[0][i] + x[1][i];
}
```

Miss Rate: _____ Total Number of Misses: _____

- v. What if we maintain the same total cache size and cache block size, but increase the associativity to 2-way set associative. Now what will be the miss rate and total number of misses of the above code, assuming that the cache starts out empty?

Miss Rate: _____ Total Number of Misses: _____

2. (cont.)

B. Given the following access results in the form (address, result) on an empty cache of total size 16 bytes, what can you infer about this cache's properties? Assume LRU replacement policy. **Circle all that apply.**

(0, Miss), (8, Miss), (0, Hit), (16, Miss), (8, Miss)

- a. The block size is greater than 8 bytes
- b. The block size is less or equal to 8 bytes
- c. This cache has only two sets
- d. This cache has more than 8 sets
- e. This cache is 2-way set associative
- f. The cache is 4-way set associative
- g. Using an 8 bit address, the tag would be 4 bits
- h. Using an 8 bit address, the tag would be greater than 4 bits
- i. None of the above

15au Final

5. Processes – 10 pts

A) What is `exec()` used for? Give an example of when it is used.

B) On a context switch, circle all of the following that would be saved:

TLB contents	Stack Pointer	Instruction Cache Contents	Heap Contents	Register Contents	Stack Contents	Condition Codes
-----------------	------------------	----------------------------------	------------------	----------------------	-------------------	--------------------

C) Given the following C program:

```
void sunny() {
    int n = 1;

    if (fork() == 0) {
        n = n << 1
        printf("%d, ", n);
        n = n << 1
    }
    if(fork() == 0) {
        n = n + 700;
    }
    printf("%d, ", n);
};
```

Which of the following outputs are possible for this function (circle all that apply):

- a. 2, 4, 1, 701, 704,
- b. 1, 2, 4, 704, 701,
- c. 2, 704, 4, 701, 1,
- d. 701, 2, 704, 4, 1,
- e. 1, 704, 2, 4, 701,
- f. 2, 1, 704, 4, 701,

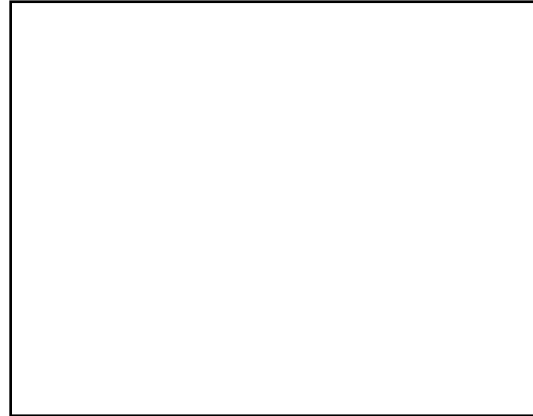
16sp Final

6. Programs, processes, and processors (oh my!) (25 pts)

- (a) Consider the following C code on the left (running on Linux), then give *one* possible output of running it. Assume that `printf` flushes its output immediately.

```
void oz() {
    char * name = "toto\n";
    printf("dorothy\n");
    if (fork() == 0) {
        name = "wizard\n";
        printf("scarecrow\n");
        fork();
        printf("tinman\n");
        exit(0);
        printf("witch\n");
    } else {
        printf("lion\n");
    }
    printf(name);
}
```

Possible output:



- (b) "*Pay no attention to the man behind the curtain.*" We have seen several different mechanisms used to create illusions or abstractions for running programs:

- A. Context switch
- B. Virtual memory
- C. Virtual method tables (vtables)
- D. Caches
- E. Timer interrupt
- F. Stack discipline
- G. None of the above, or impossible.

For each of the following, indicate which mechanism above (A-F) enables the behavior, or G if the behavior is impossible or untrue.

- (i) _____ Allows operating system kernel to run to make scheduling decisions.
- (ii) _____ Prevents buffer overflow exploits.
- (iii) _____ Allows multiple instances of the same program to run concurrently.
- (iv) _____ Lets programs use more memory than the machine has.
- (v) _____ Makes recently accessed memory faster.
- (vi) _____ Multiple processes appear to run concurrently on a single processor.
- (vii) _____ Enables programs to run different code depending on an object's type.
- (viii) _____ Allows an x86-64 machine to execute code for a different ISA.

Name: _____

(c) Give an example of a *synchronous* exception, what could trigger it, and where the exception handler would return control to in the original program.

(d) In what way does address translation (virtual memory) help make *exec* fast? Explain in less than 2 sentences. *Hint*: it may help to write down what happens during *exec*.

(e) Which of the following *can* a running process determine, assuming it does *not* have access to a timer? (*check all that apply*)

- Its own process ID
- Size of physical memory
- Size of the virtual address space
- L1 cache associativity
- When context switches happen

(f) For each of the following, fill in what is responsible for making the decision: hardware ("HW"), operating system ("OS"), or program ("P").

- (i) _____ Which physical page a virtual page is mapped to.
- (ii) _____ Which cache line is evicted for a conflict in a set-associative cache.
- (iii) _____ Which page is evicted from physical memory during a page fault.
- (iv) _____ Translation from virtual address to physical address.
- (v) _____ Whether data is stored in the stack or the heap.
- (vi) _____ Data layout optimized for spatial locality

17sp Final

3. Virtual Memory (9 points)

Assume we have a virtual memory detailed as follows:

- 256 MiB Physical Address Space
- 4 GiB Virtual Address Space
- 1 KiB page size
- A TLB with 4 sets that is 8-way associative with LRU replacement

For the following questions it is fine to leave your answers as powers of 2.

a) How many bits will be used for:

Page offset? _____

Virtual Page Number (VPN)? _____ Physical Page Number (PPN)? _____

TLB index? _____ TLB tag? _____

b) How many entries in this page table?

c) We run the following code with an empty TLB. Calculate the TLB miss rate for data (ignore instruction fetches). Assume **i** and **sum** are stored in registers and **cool** is page-aligned.

```
#define LEAP 8
int cool[512];
... // Some code that assigns values into the array cool
... // Now flush the TLB. Start counting TLB miss rate from here.
int sum;
for (int i = 0; i < 512; i += LEAP) {
    sum += cool[i];
}
```

TLB Miss Rate: (fine to leave you answer as a fraction) _____

16au Final

Question F7: Virtual Memory [10 pts]

Our system has the following setup:

- 24-bit virtual addresses and 512 KiB of RAM with 4 KiB pages
- A 4-entry TLB that is fully associative with LRU replacement
- A page table entry contains a valid bit and protection bits for read (R), write (W), execute (X)

(A) Compute the following values: [2 pt]

Page offset width _____ PPN width _____
Entries in a page table _____ TLBT width _____

(B) Briefly explain why we make the page size so much larger than a cache block size. [2 pt]

(C) Fill in the following blanks with “A” for always, “S” for sometimes, and “N” for never if the following get updated during a **page fault**. [2 pt]

Page table _____ Swap space _____ TLB _____ Cache _____

(D) The TLB is in the state shown when the following code is executed. Which iteration (value of *i*) will cause the **protection fault (segfault)**? Assume *sum* is stored in a register.

Recall: the hex representations for TLBT/PPN are padded as necessary. [4 pt]

```
long *p = 0x7F0000, sum = 0;
for (int i = 0; 1; i++) {
    if (i%2)
        *p = 0;
    else
        sum += *p;
    p++;
}
```

TLBT	PPN	Valid	R	W	X
0x7F0	0x31	1	1	1	0
0x7F2	0x15	1	1	0	0
0x004	0x1D	1	1	0	1
0x7F1	0x2D	1	1	0	0

i =

Question F10: Memory Allocation [18 pts]

- (A) In the following code, briefly identify the TWO memory errors. They can be fixed by changing ONE line of code. [6 pt]

```
int N = 64;
double *func(double A[][], double x[]) {
    double *z = (double *) malloc(N * sizeof(float));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            z[i] = A[i][j] + z[i] * x[j];
        }
    }
    return z;
}
```

Error 1:

Error 2:

Line of code with fixes:

- (B) We are using a dynamic memory allocator on a **64-bit machine** with an **explicit free list**, **4-byte boundary tags**, and **16-byte alignment**. Assume that a footer is always used. [6 pt]

<u>Request</u>	<u>return value</u>	<u>block addr</u>	<u>block size</u>	<u>internal fragmentation in this block</u>
<code>p = malloc(12);</code>	<code>0x610</code>	<code>0x_____</code>	<code>_____ bytes</code>	<code>_____ bytes</code>

- (C) 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 the **lowest/smallest**. [6 pt]

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";

int main(int argc, char *argv[]) {
    int *foo = malloc(888);
    int bar = 351;
    free(foo);
    return 0;
}
```

Group 1:	<code>&bar</code>	<code>&foo</code>	<code>foo</code>
Group 2:	<code>&foo</code>	<code>main</code>	<code>str</code>
Group 3:	<code>bar</code>	<code>&str</code>	<code>&ZERO</code>

19sp Final

4. Memory Allocation (11 points total)

```
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) [3 pts] 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) [4 pts] 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:

Error #2:

- c) [2 pts] (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.

- d) [2 pts] List two reasons why it would be hard to write a garbage collector for the C programming language.

Reason #1:

Reason #2:

18sp Final

8. (11 points) (Java)

- (a) For each course topic below that we studied using C, answer *yes* if the topic is *directly relevant* in Java as well (else answer *no*).
- Floating-point operations often produce small rounding errors that can compound over many operations.
 - Pointer arithmetic is scaled by the size of the pointed-to object.
 - Using uninitialized data can lead to garbage results that depend on whatever happened to be in that memory previously.
 - Keeping your working set small helps improve the performance of memory operations in general, without concern for the exact parameters of a machine's memory hierarchy.
- (b) Some of the safety checks that are performed in Java but not in C require extra data in memory, i.e., fields that are part of Java data but not part of the analogous C data. For each of the following, answer *yes* if Java needs such extra data to perform the operation (else answer *no*).
- Throwing an `ArrayIndexOutOfBoundsException` if an array index is too large.
 - Throwing a `NullPointerException` if the `e` in `e.m()` evaluates to `null`.
 - Throwing a `ClassCastException` if the `e` in `(Foo)e` does not evaluate to an instance of `Foo`.
- (c) Consider this Java code, which is part of a larger program.

```
class Foo {
    int x;
    boolean sameAsX(int y) { return y == this.x; }
    boolean sameAs7(int y) { return y == 7; }
}
class Bar {
    boolean whyNotBoth(Foo f, int z) {
        return f.sameAsX(z) && f.sameAs7(z);
    }
}
```

Your friend suggests that when compiling the method call `f.sameAs7(z)` above, the compiler can optimize out the instruction that passes `this` as a procedure argument since the `sameAs7` method in `Foo` does not use it. Explain in roughly 1–2 sentences why this “optimization” is wrong.