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]

**Explanation:**
An *int* contains 32 bits of information, which always fits into the 52-bit mantissa of a *double*, but not always into the 23-bit mantissa of a *float*.

(B) What value will be read after we try to store $-2^{127} - 2^{101}$ in a *float*? (Circle one) [4 pt]

- $-2^{127}$
- $-\text{NaN}$
- $-\infty$
- $-2^{127} - 2^{101}$

$-2^{127} - 2^{101} = -2^{127} \times 1.000000000000000000000012$.

Exp = 127 is a representable normalized exponent ($E = 0b11111110$)

All of the bits following the binary point in the Mantissa fit into the M field (23 bits).

(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]

```c
int aligned(void* p, int K) {
    return !(long)p%K; // other variants accepted, e.g.
    // (long)p%K == 0
    // (long)p == (long)p/K*K
    // !(long)p & (K-1))
}
```

(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
               Unaligned               Unaligned

*float:* $S = 0$, $E = 0b1010~0000$, $M = 0$, so $+1.02 \times 2^{33}$. You can recognize that this is larger than $T_{Max}$ and $U_{Max}$.

*int/unsigned int:* Positive encodings are the same for both representations. Value is $5 \times 16^7 = 2^{30} + 2^{28}$.

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.

There are 4 instructions that would need to be changed: push, pop, call, ret.

*push* and *call* would now need to *increase* %rsp.

*pop* and *ret* would now need to *decrease* %rsp.
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:

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

```c
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`.

<table>
<thead>
<tr>
<th>Expression (in C)</th>
<th>Type</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp + 2</td>
<td>long*</td>
<td>0x20</td>
</tr>
<tr>
<td>*(sp - 1)</td>
<td>short</td>
<td>0x52FC</td>
</tr>
<tr>
<td>ip[5]</td>
<td>int</td>
<td>0x31F40370</td>
</tr>
<tr>
<td>&amp;ip</td>
<td>int**</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

**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.*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movl 2(%rax), %ebx</code></td>
<td></td>
</tr>
<tr>
<td><code>leal (%rax,%rax,2), %ecx</code></td>
<td></td>
</tr>
<tr>
<td><code>movsbl 4(%rax), %edi</code></td>
<td></td>
</tr>
<tr>
<td><code>subw (%rax,2), %si</code></td>
<td></td>
</tr>
<tr>
<td><code>movl 2(%rax), %ebx</code></td>
<td>0x84f7 e710</td>
</tr>
<tr>
<td><code>leal (%rax,%rax,2), %ecx</code></td>
<td>0x0000 000c</td>
</tr>
<tr>
<td><code>movsbl 4(%rax), %edi</code></td>
<td>0x0000 0000 ffff fff7</td>
</tr>
<tr>
<td><code>subw (%rax,2), %si</code></td>
<td>0x7B09</td>
</tr>
</tbody>
</table>
Consider the following x86-64 assembly and C code for the recursive function `rfun`.

```c
// 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);
}
```

```assembly
00000000004005e6 <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.]

34 bytes

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

3

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

*s

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

0x4005fb

(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.]

<table>
<thead>
<tr>
<th>Total Frames:</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Depth:</td>
<td>6</td>
</tr>
</tbody>
</table>

main -> rfun(s) -> rfun(s+1) -> rfun(s+2) -> rfun(s+3) -> rfun(s+4) -> printf()

The recursive call to rfun(s+4), which handles the null-terminator in the string does create a stack frame since we consider the return address pushed to the stack during a procedure call to be part of the callee’s stack frame.
(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.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffdb48</td>
<td>unknown</td>
<td>%rsp when main is entered</td>
</tr>
<tr>
<td>0x7fffffffdb38</td>
<td>0x400616</td>
<td>Return address to main</td>
</tr>
<tr>
<td>0x7fffffffdb30</td>
<td>unknown</td>
<td>original %rbx</td>
</tr>
<tr>
<td>0x7fffffffdb28</td>
<td>0x4005fb</td>
<td>Return address</td>
</tr>
<tr>
<td>0x7fffffffdb20</td>
<td>*s, “Y”</td>
<td>Saved %rbx</td>
</tr>
<tr>
<td>0x7fffffffdb18</td>
<td>0x4005fb</td>
<td>Return address</td>
</tr>
<tr>
<td>0x7fffffffdb10</td>
<td>*s, *(s+1), “a”</td>
<td>Saved %rbx</td>
</tr>
<tr>
<td>0x7fffffffdb08</td>
<td>0x4005fb</td>
<td>Return address</td>
</tr>
<tr>
<td>0x7fffffffdb00</td>
<td>*s, *(s+2), “y”</td>
<td>Saved %rbx</td>
</tr>
</tbody>
</table>
4. (15 points) (x86-64 Assembly) This problem considers this assembly implementation of a C function of the from 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.

Solution:

(a) cmpq $0, %rdi
(b) shlq $1, %rdi for the first addq instruction
(c) Intended answer: ret or jmp .L2 needs to be added after this first jump, i.e., immediately before .L1. But it also works to put [back] jle .L2 either before or after the .L1 or even before the jg .L1, so that also receives full credit.
(d) It takes the number in %rdi and returns the number of times it needs to be doubled before the repeated doubling produces a non-positive number. (If the original number was positive, this will be due to overflow.) An alternate description is it returns 63 minus the bit position of the left-most 1-bit in %rdi where the least-significant bit is position 0 and returning 0 if there is no 1-bit.

(e) 63
2. (12 points) (Struct Layout) Assume x86-64 and Linux and that all fields should be properly aligned.

(a) Consider this struct definition:

```c
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:

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

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

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

Solution:

(a) size is 16 bytes;
   x at offset 0 for 4 bytes,
   then padding for 4 bytes (internal fragmentation),
   then p at offset 8 for 8 bytes

(b) size is 24 bytes;
   x at offset 0 for 4 bytes,
   then padding for 4 bytes (internal fragmentation),
   then p at offset 8 for 8 bytes,
   then y at offset 16 for 4 bytes,
   then padding for 4 bytes (external fragmentation)

(c) size is 16 bytes,
   p at offset 0 for 8 bytes,
   then x at offset 8 for 4 bytes,
   then y at offset 12 for 4 bytes.
(No padding.)
Consider the following variable declarations, assuming x86_64 architecture:

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

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

Fill in the following table:

<table>
<thead>
<tr>
<th>C Expression</th>
<th>Evaluates to?</th>
<th>Resulting data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>0x10000000</td>
<td>struct_type*</td>
</tr>
<tr>
<td>n</td>
<td>0x20000000</td>
<td>struct_type*</td>
</tr>
<tr>
<td>&amp;(m-&gt;a)</td>
<td>0x10000000</td>
<td>int*</td>
</tr>
<tr>
<td>&amp;(m-&gt;b)</td>
<td>0x10000004</td>
<td>char*</td>
</tr>
<tr>
<td>&amp;(m-&gt;c)</td>
<td>0x10000008</td>
<td>double*</td>
</tr>
<tr>
<td>sizeof(struct_type)</td>
<td>16</td>
<td>size_t (or int)</td>
</tr>
<tr>
<td>sizeof(*m)</td>
<td>16</td>
<td>size_t (or int)</td>
</tr>
<tr>
<td>sizeof(m)</td>
<td>8</td>
<td>size_t (or int)</td>
</tr>
<tr>
<td>&amp;(n[0])</td>
<td>0x20000000</td>
<td>struct_type*</td>
</tr>
<tr>
<td>&amp;(n[0].a)</td>
<td>0x20000000</td>
<td>int*</td>
</tr>
<tr>
<td>&amp;(n[1].a)</td>
<td>0x20000010</td>
<td>int*</td>
</tr>
</tbody>
</table>

Some students answered "pointer" or "address" as the resulting data type. This was not specific enough to receive full credit.

Typos: the struct declaration should have been a `typedef` so that `struct_type`. As it was written, `struct_type` would be a variable of the unnamed struct type.
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: ______5______    Cache Tag: ______14______

b) How many sets will the cache have? ______8______

c) Assume that everything except the array x is stored in registers, and that the array 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.

```c
#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:** ___2/3______     **Total Number of Hits:** ___512______

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:** ___5/6______     **Total Number of Hits:** ___640______

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
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:

\[
\text{int } x[2][64];
\]

i. How many sets will the cache have?

\[\frac{256}{16} = 16 \text{ sets}\]

ii. How many bits will be required for the cache block offset?

4 bits

iii. If the physical addresses are 22 bits, how many bits are in the cache tag?

\[22 - 4 - 4 = 14 \text{ bits}\]

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:

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

Miss Rate: ___100%__________  Total Number of Misses: ___128__________

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: ___25%__________  Total Number of Misses: ___32__________
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.**

\[(0, \text{Miss}), (8, \text{Miss}), (0, \text{Hit}), (16, \text{Miss}), (8, \text{Miss})\]

\[
\begin{align*}
\text{a.} & \quad \text{The block size is greater than 8 bytes} \\
\text{b.} & \quad \textbf{The block size is less or equal to 8 bytes} \\
\text{c.} & \quad \text{This cache has only two sets} \\
\text{d.} & \quad \text{This cache has more than 8 sets} \\
\text{e.} & \quad \textbf{This cache is 2-way set associative} \\
\text{f.} & \quad \text{The cache is 4-way set associative} \\
\text{g.} & \quad \text{Using an 8 bit address, the tag would be 4 bits} \\
\text{h.} & \quad \textbf{Using an 8 bit address, the tag would be greater than 4 bits} \\
\text{i.} & \quad \text{None of the above}
\end{align*}
\]

*Block sizes will range from 1 to 8 bytes, thus the number of sets will range from 1 to 8. All combos of block sizes and number of sets will use only 3 bits, leaving 5 bits for the tag.*
5. Processes – 10 pts

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

`exec` replaces the current process’ code with the code for a different program. It is used in the fork-exec model to get a child process to execute a program different than its parent. The example we used in class was when the user types a command like `ls` at the command line, the bash shell first forks a child process running bash and then calls `exec(ls)` in the child to get it to run the `ls` program. Note: process and program are two very different things! `exec` does not create processes, fork does that.

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

<table>
<thead>
<tr>
<th>TLB contents</th>
<th>Stack Pointer</th>
<th>Instruction Cache Contents</th>
<th>Heap Contents</th>
<th>Register Contents</th>
<th>Stack Contents</th>
<th>Condition Codes</th>
</tr>
</thead>
</table>

The wording on this question was a little vague. Heap and Stack contents would need to be preserved on a context switch – a process should not lose any of these! But they do not necessarily need to be explicitly saved at the time of context switch. We counted it as o.k. if you circled heap or stack.

C) Given the following C program:

```c
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,
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.

```c
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dorothy</td>
<td>dorothy</td>
</tr>
<tr>
<td>scarecrow</td>
<td>lion</td>
</tr>
<tr>
<td>tinman</td>
<td>toto</td>
</tr>
<tr>
<td>tinman</td>
<td>scarecrow</td>
</tr>
<tr>
<td>lion</td>
<td>tinman</td>
</tr>
<tr>
<td>toto</td>
<td>tinman</td>
</tr>
</tbody>
</table>

(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.
(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.

**Page fault:** triggered by access to virtual address not in memory, returns to the instruction that caused the fault.

**Trap:** used to for syscalls to do something protected by the kernel, returns to after the calling instruction.

(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.

*Address translation is a form of indirection, it allows us to implement fork without copying the whole process's memory, and exec without loading the whole program into memory at once.*

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

- [X] Its own process ID
- [ ] Size of physical memory
- [X] 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) _____ OS Which physical page a virtual page is mapped to.

(ii) _____ HW Which cache line is evicted for a conflict in a set-associative cache.

(iii) _____ OS Which page is evicted from physical memory during a page fault.

(iv) _____ HW Translation from virtual address to physical address.

(v) _____ P Whether data is stored in the stack or the heap.

(vi) _____ P Data layout optimized for spatial locality
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? _____10_____
- Virtual Page Number (VPN)? _____22_____ Physical Page Number (PPN)? ____18_____
- TLB index? ______2_________ TLB tag? ______20_________

b) How many entries in this page table?

$$2^{22}$$

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.

```c
#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) $$\frac{1}{32}$$
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]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page offset width</td>
<td>12</td>
</tr>
<tr>
<td>PPN width</td>
<td>7</td>
</tr>
<tr>
<td>Entries in a page table</td>
<td>2^12</td>
</tr>
<tr>
<td>TLBT width</td>
<td>12</td>
</tr>
</tbody>
</table>

Because TLB is fully associative, TLBT width matches VPN. There are 2^{VPN width} entries in PT.

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

Take advantage of spatial locality and try to avoid page faults as much as possible. Disk access is also super slow, so we want to pull a lot of data when we do access it.

(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]

<table>
<thead>
<tr>
<th>Component</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page table</td>
<td>A</td>
</tr>
<tr>
<td>Swap space</td>
<td>S</td>
</tr>
<tr>
<td>TLB</td>
<td>A/N</td>
</tr>
<tr>
<td>Cache</td>
<td>S</td>
</tr>
</tbody>
</table>

When the page is placed in physical memory, the new PPN is written into the page table entry. Swap space will get updated if a dirty page is kicked out of physical memory.

For this class, we say that the page fault handler updates the TLB because it is more efficient. In reality not all do (OS does not have access to hardware-only TLB; instr gets restarted). To update a PTE (in physical mem), you check the cache, so it gets updated on a cache miss.

(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; i < 513; i++) {
    if (i%2)
        *p = 0;
    else
        sum += *p;
    p++;
}
```

<table>
<thead>
<tr>
<th>TLBT</th>
<th>PPN</th>
<th>Valid</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7F0</td>
<td>0x31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x7F2</td>
<td>0x15</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x004</td>
<td>0x1D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x7F1</td>
<td>0x2D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```
i = 513
```

Only the current page (VPN = TLBT = 0x7F0) has write access. Once we hit the next page (TLBT = 0x7F1), we will encounter a segfault once we try to write to the page. We are using pointer arithmetic to increment our pointer by 8 bytes at a time. One page holds 2^{12}/2^3 = 512 longs, so we first access TLBT 0x7F1 when i = 512. However, the code is set up so that we only write on odd values of i, so the answer is i = 513.
18au Final

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]

```c
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: Wrong allocation size / buffer overflow – N*sizeof(float) |
| Line of code with fixes: double *z = (double *) calloc(N, sizeof(double)); |

(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]

<table>
<thead>
<tr>
<th>Request</th>
<th>return value</th>
<th>block addr</th>
<th>block size</th>
<th>internal fragmentation in this block</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = malloc(12);</td>
<td>0x610</td>
<td>0x_60C_</td>
<td><em>32</em> bytes</td>
<td><em>20</em> bytes</td>
</tr>
</tbody>
</table>

Block starts a header size before the payload (returned addr). Minimum block size in explicit free 
list is set by header+footer+2 pointers = 24 bytes, then aligned to 16-bytes: 32 bytes. Internal 
fragmentation is size of block – payload size = 32 – 12 = 20 bytes.

(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]

```c
#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;
}
```

<table>
<thead>
<tr>
<th>Group 1: &amp;bar &amp;foo foo</th>
<th>Group 2: &amp;foo main str</th>
<th>Group 3: bar &amp;str &amp;ZERO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7) &amp;foo/&amp;bar (Stack)</td>
<td>6) foo (Heap)</td>
<td>5) &amp;ZERO/&amp;str (Static Data)</td>
</tr>
<tr>
<td>6) foo</td>
<td>5) &amp;ZERO/&amp;str (Static Data)</td>
<td></td>
</tr>
<tr>
<td>5) &amp;ZERO/&amp;str (Static Data)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4) str (Literals)</td>
<td>3) main (Code)</td>
<td>2) bar (351)</td>
</tr>
<tr>
<td>3) main (Code)</td>
<td>2) bar (351)</td>
<td>1) ZERO (0)</td>
</tr>
</tbody>
</table>
4. Memory Allocation (11 points total)

```c
#include <stdlib.h>
float pi = 3.14;

int main(int argc, char *argv[]) {
    int year = 2019;
    int* happy = malloc(sizeof(int*));
    happy++;
    free(happy);
    return 0;
}
```

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:** 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) [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.

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) [2 pts] 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.
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).

i. Floating-point operations often produce small rounding errors that can compound over many operations.

ii. Pointer arithmetic is scaled by the size of the pointed-to object.

iii. Using uninitialized data can lead to garbage results that depend on whatever happened to be in that memory previously.

iv. 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).

i. Throwing an ArrayIndexOutOfBoundsException if an array index is too large.

ii. Throwing a NullPointerException if the e in e.m() evaluates to null.

iii. 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.

```java
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.

**Solution:**

(a) i. yes  (b) i. yes  
ii. no     ii. no  
iii. no    iii. yes 
iv. yes

(c) A subclass of Foo could override sameAs7 with a method that uses the this pointer and the first argument to whyNotBoth could be an instance of this subclass. With the optimization, the callee will not have the proper this pointer it needs.