Data representation exercises

Many exercises that seem less appropriate this year, or which cover topics that we haven't covered in class, are marked with ⚠️. However, we may have missed some.

DATAREP-1. Sizes and alignments

QUESTION DATAREP-1A. True or false: For any non-array type X, the size of X (`sizeof(X)`) is greater than or equal to the alignment of type X (`alignof(X)`).

Show solution

True.

This also mostly true for arrays. The exception is zero-length arrays: `sizeof(X[0]) == 0`, but `alignof(X[0]) == alignof(X)`.

Hide solution

Hide solution

QUESTION DATAREP-1B. True or false: For any type X, the size of `struct Y { X a; char newc; }` is greater than the size of X.

Show solution

True

Hide solution

Hide solution

QUESTION DATAREP-1C. True or false: For any types `A1`...`An` (with `n` ≥ 1), the size of `struct Y` is greater than the size of `struct X`, given:

```
struct X {
    A1 a1;
    ...
    An an;
};
```

```c
struct Y {
    A1 a1;
    ...
    An an;
    char newc;
};
```

Show solution

False (example: `A1 = int`, `A2 = char`)

Hide solution

Hide solution

QUESTION DATAREP-1D. True or false: For any types `A1`...`An` (with `n` ≥ 1), the size of `struct Y` is greater than the size of `union X`, given:

```c
union X {
    A1 a1;
    ...
    An an;
};
```

```c
struct Y {
    A1 a1;
    ...
    An an;
};
```

Show solution

False (if `n = 1`)

Hide solution

Hide solution

QUESTION DATAREP-1E. Assume that structure `struct Y { ... }` contains *K* `char` members and *M* `int` members, with *K*≤*M*, and nothing else. Write an expression defining the **maximum** `sizeof(struct Y)`.

Show solution

4M + *4K*

Hide solution

Hide solution

QUESTION DATAREP-1F. You are given a structure `struct Z { T1 a; T2 b; T3 c; }` that contains no padding. What does `(sizeof(T1) + sizeof(T2) + sizeof(T3)) % alignof(struct Z)` equal?

Show solution

0

Hide solution

Hide solution

QUESTION DATAREP-1G. Arrange the following types in increasing order by size. Sample answer: "1 < 2 = 4 < 3" (choose this if #1 has smaller size than #2, which has equal size to #4, which has smaller size than #3).

1. `char`
2. `struct minipoint { uint8_t x; uint8_t y; uint8_t z; }`
3. `int`
4. `unsigned short[1]`
5. `char**`
6. `double[0]`

Show solution

#6 < #1 < #4 < #2 < #3 = #5

Hide solution

Hide solution

DATAREP-2. Expressions

QUESTION DATAREP-2A. Here are eight expressions. Group the expressions into four pairs so that the two expressions in each pair have the same value, and each pair has a different value from every other pair. There is one unique answer that meets these constraints. `m` has the same type and value everywhere it appears (there's one unique value for `m` that meets the problem's constraints). Assume an x86-32 machine: a **32-bit** architecture in which pointers are 32 bits long.

1. `sizeof(&m)`
2. `-1`
3. `m & -m`
4. `m + ~m + 1`
5. `16 >> 2`
6. `m & ~m`
7. `m`
8. `1`

Show solution

1—5; 2—7; 3—8; 4—6

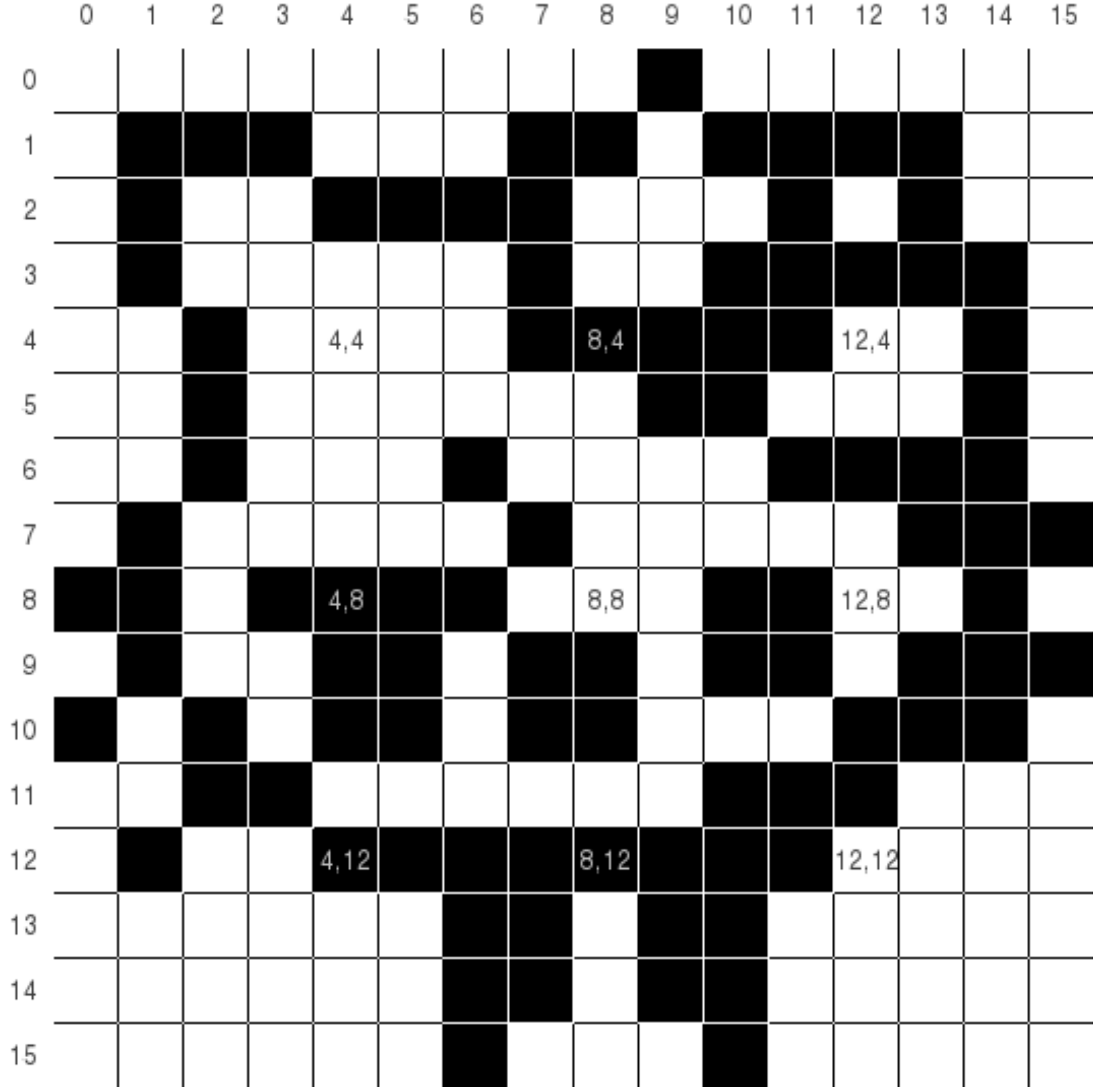
1—5 is easy. `m + ~m + 1 == m + (-m) == 0`, and `m & ~m == 0`, giving us 3—8. Now what about the others? `m & -m` (#3) is either 0 or a power of 2, so it cannot be -1 (#2). The remaining possiblities are `m` and 1 . If `(m & -m) == m`, then the remaining pair would be 1 and -1, which clearly doesn't work. Thus `m & -m` matches with 1, and `m == -1`.

Hide solution

Hide solution

DATAREP-3. Hello binary

This problem locates 8-bit numbers horizontally and vertically in the following 16x16 image. Black pixels represent 1 bits and white pixels represent 0 bits. For horizontal arrangements, the most significant bit is on the left as usual. For vertical arrangements, the most significant bit is on top.



Examples: The 8-bit number 15 (hexadecimal 0x0F, binary 0b00001111) is located horizontally at 3,4, which means X=3, Y=4.

- The pixel at 3,4 is white, which has bit value 0.
- 4,4 is white, also 0.
- 5,4 is white, also 0.
- 6,4 is white, also 0.
- 7,4 is black, which has bit value 1.
- 8,4, 9,4, and 10,4 are black, giving three more 1s.
- Reading them all off, this is 0b00001111, or 15.

15 is also located horizontally at 7,6.

The 8-bit number 0 is located vertically at 0,0. It is also located horizontally at 0,0 and 1,0.

The 8-bit number 134 (hexadecimal 0x86, binary 0b10000110) is located vertically at 8,4.

QUESTION DATAREP-3A. Where is 3 located vertically? (All questions refer to 8-bit numbers.)

Show solution

9,6

Hide solution

Hide solution

QUESTION DATAREP-3B. Where is 12 located horizontally?

Show solution

5,5

Hide solution

Hide solution

QUESTION DATAREP-3C. Where is 255 located vertically?

Show solution

14,3

Hide solution

Hide solution

DATAREP-4. Hello memory

Shintaro Tsuji wants to represent the image of Question DATAREP-3 in computer memory. He stores it in an array of 16-bit unsigned integers:

```
unsigned short cute[16];
```

Row Y of the image is stored in integer `cute[Y]`.

QUESTION DATAREP-4A. What is `sizeof(cute)`, 2, 16, 32, or 64?

Show solution

32

Hide solution

Hide solution

QUESTION DATAREP-4B. `printf("%d\n", cute[0]);` prints `16384`. Is Shintaro’s machine big-endian or little-endian?

Show solution

Little-endian

Hide solution

Hide solution

DATAREP-5. Hello program

Now that Shintaro has represented the image in memory as an array of `unsigned short` objects, he can manipulate the image using C. For example, here’s a function.

```
void swap(void) {
    for (int i = 0; i < 16; ++i) {
        cute[i] = (cute[i] << 8) | (cute[i] >> 8);
    }
}
```

Running `swap` produces the following image:

Shintaro has written several other functions. Here are some images (A is the original):


A


B


C


D


E


F


G


H


I


J

For each function, what image does that function create?

QUESTION DATAREP-5A.

```
void f0() {
    for (int i = 0; i < 16; ++i) {
        cute[i] = ~cute[i];
    }
}
```

Show solution

H. The code flips all bits in the input.

Hide solution

Hide solution

QUESTION DATAREP-5B.

```
void f1() {
    for (int i = 0; i < 16; ++i) {
        cute[i] = ((cute[i] >> 1) & 0x5555) | ((cute[i] << 1) & 0xAAAA);
        cute[i] = ((cute[i] >> 2) & 0x3333) | ((cute[i] << 2) & 0xCCCC);
        cute[i] = ((cute[i] >> 4) & 0x0F0F) | ((cute[i] << 4) & 0xF0F0);
        cute[i] =  (cute[i] >> 8)           |  (cute[i] << 8);
    }
}
```

Show solution

D

Hide solution

Hide solution

QUESTION DATAREP-5C.

```
void f2() {
    char* x = (char*) cute;
    for (int i = 0; i < 16; ++i) {
        x[2*i] = i;
    }
}
```

Show solution

> J
>
> Hide solution
>
> Hide solution

For "fun"

The following programs generated the other images. Can you match them with their images?

```
void f3() {
    for (int i = 0; i < 16; ++i) {
        cute[i] &= ~(7 << i);
    }
}

void f4() {
    swap();
    for (int i = 0; i < 16; ++i) {
        cute[i] <<= i/4;
    }
    swap();
}

void f5() {
    for (int i = 0; i < 16; ++i) {
        cute[i] = -1 * !!(cute[i] & 64);
    }
}

void f6() {
    for (int i = 0; i < 8; ++i) {
        int tmp = cute[15-i];
        cute[15-i] = cute[i];
        cute[i] = tmp;
    }
}

void f7() {
    for (int i = 0; i < 16; ++i) {
        cute[i] = cute[i] & -cute[i];
    }
}

void f8() {
    for (int i = 0; i < 16; ++i) {
        cute[i] ^= cute[i] ^ cute[i];
    }
}

void f9() {
    for (int i = 0; i < 16; ++i) {
        cute[i] = cute[i] ^ 4080;
    }
}
```

Show solution

`f3`—I; `f4`—B; `f5`—C; `f6`—F; `f7`—G; `f8`—A; `f9`—E

Hide solution

Hide solution

DATAREP-6. Memory regions

Consider the following program:

```c
struct ptrs {
    int** x;
    int* y;
};

struct ptrs global;

void setup(struct ptrs* p) {
    int* a = malloc(sizeof(int));
    int* b = malloc(sizeof(int));
    int* c = malloc(sizeof(int));
    int* d = malloc(sizeof(int));
    int* e = malloc(sizeof(int) * 2);
    int** f = malloc(4 * sizeof(int*));
    int** g = malloc(sizeof(int*));

    *a = 0;
    *b = 0;
    *c = (int) a;
    *d = *b;
    e[0] = 29;
    e[1] = (int) &d[100000];

    f[0] = b;
    f[1] = c;
    f[2] = 0;
    f[3] = 0;

    *g = c;

    global.x = f;
    global.y = e;

    p->x = g;
    p->y = &e[1];
}

int main(int argc, char** argv) {
    stack_bottom = (char*) &argc;
    struct ptrs p;
    setup(&p);
    m61_collect();
    do_stuff(&p);
}
```

This program allocates objects `a` through `g` on the heap and then stores those pointers in some stack and global variables. (It then calls our conservative garbage collector from class⚠️, but that won't matter until the next problem.) We recommend you draw a picture of the state `setup` creates.

QUESTION DATAREP-6A. Assume that `(uintptr_t) a == 0x8300000`, and that `malloc` returns increasing addresses. Match each address to the most likely expression with that address value. The expressions are evaluated within the context of `main`. You will not reuse an expression.

	Value		Expression
1.	0x8300040	A.	`&p`
2.	0x8049894	B.	`(int*) *p.x[0]`
3.	0x8361AF0	C.	`&global.y`
4.	0x8300000	D.	`global.y`
5.	0xBFAE0CD8	E.	`(int*) *p.y`

Show solution

1—D; 2—C; 3—E; 4—B; 5—A

Since `p` has automatic storage duration, it is located on the stack, giving us 5—A. The `global` variable has static storage duration, and so does its component `global.y`; so the pointer `&global.y` has an address that is below all heap-allocated pointers. This gives us 2—C. The remaining expressions go like this:

`global.y == e`;
`p.y == &e[1]`, so `*p.y == e[1] == (int) &d[100000]`, and `(int *) *p.y == &d[100000]`;
`p.x == g`, so `p.x[0] == g[0] == *g == c`, and `*p.x[0] == *c == (int) a`.

Address #4 has value 0x8300000, which by assumption is `a`'s address; so 4—B. Address #3 is much larger than the other heap addresses, so 3—E. This leaves 1—D.

Hide solution

Hide solution

DATAREP-7. Garbage collection

Here is the top-level function for the conservative garbage collector we wrote in class. (⚠️ *2018 note: We haven't done this.* ⚠️)

```c
void m61_collect(void) {
    char* stack_top = (char*) &stack_top;

    // The entire contents of the heap start out unmarked
    for (size_t i = 0; i != nmr; ++i) {
        mr[i].marked = 0;
    }

    // Mark all reachable objects, starting with the roots (the stack)
    m61_markaccessible(stack_top, stack_bottom - stack_top);

    // Free everything that wasn't marked
    for (size_t i = 0; i != nmr; ++i) {
        if (mr[i].marked == 0) {
            m61_free(mr[i].ptr);
            --i;                   // m61_free moved different data into this
                                   // slot, so we must recheck the slot
        }
    }
}
```

This garbage collector is not correct because it doesn't capture all memory roots.

Consider the program from the previous section, and assume that an object is *reachable* if `do_stuff` can access an address within the object via variable references and memory dereferences *without casts or pointer arithmetic*. Then:

QUESTION DATAREP-7A. Which *reachable* objects will `m61_collect()` free? Circle all that apply.

a b c d e f g None of these

Show solution

`b`, `f`.

The collector searches the stack for roots. This yields just the values in `struct ptrs p` (the only pointer-containing variable with automatic storage duration at the time `m61_collect` is called). The objects directly pointed to by `p` are `g` and `e`. The collector then recursively marks objects pointed to by these objects. From `g`, it finds `c`. From `e`, it finds nothing. Then it checks one more time. From `c`, it finds the value of `a`! Now, `a` is actually not a pointer here—the type of `*c` is `int`—so by the definition above, `a` is not actually reachable. But the collector doesn't know this.

Putting it together, the collector marks `a`, `c`, `e`, and `g`. It won't free these objects; it will free the others (`b`, `d`, and `f`). But `b` and `f` are reachable from `global`.

Hide solution

Hide solution

QUESTION DATAREP-7B. Which *unreachable* objects will `m61_collect()` *not* free? Circle all that apply.

`a` `b` `c` `d` `e` `f` `g` None of these

Show solution

`a`

Hide solution

Hide solution

QUESTION DATAREP-7C. Conservative garbage collection in C is often slower than precise garbage collection in languages such as Java. Why? Circle all that apply.

1. C is generally slower than other languages.
2. Conservative garbage collectors must search all reachable memory for pointers. Precise garbage collectors can ignore values that do not contain pointers, such as large character buffers.
3. C programs generally use the heap more than programs in other languages.
4. None of the above.

Show solution

#2

Hide solution

Hide solution

DATAREP-8. Memory errors

The following function constructs and returns a lower-triangular matrix of size *N*. The elements are random 2-dimensional points in the unit square. The matrix is represented as an array of pointers to arrays.

```
struct point2 {
    double d[2];
};
typedef point2* point2_vector;

point2_vector* make_random_lt_matrix(size_t N) {
    point2_vector* m = (point2_vector*) malloc(sizeof(point2_vector) * N);
    for (size_t i = 0; i < N; ++i) {
        m[i] = (point2*) malloc(sizeof(point2) * (i + 1));  /* LINE A */
        for (size_t j = 0; j <= i; ++j) {
            for (int d = 0; d < 2; ++d) {
                m[i][j].d[d] = drand48();                    /* LINE B */
            }
        }
    }
    return m;
}
```

This code is running on an x86-**32** machine (`size_t` is **32 bits**, not 64). You may assume that the machine has enough free physical memory and the process has enough available virtual address space to satisfy any memory allocation request.

QUESTION DATAREP-8A. Give a value of *N* so that, while `make_random_lt_matrix(N)` is running, no `new` fails, but a memory error (such as a null pointer dereference or an out-of-bounds dereference) happens on Line A. The memory error should happen specifically when `i == 1`.

(This problem is probably easier when you write your answer in hexadecimal.)

Show solution

We are asked to produce a value of *N* so that no memory error happens on Line A when `i == 0`, but a memory error *does* happen when `i == 1`. So reason that through. What memory errors could happen on Line A if `malloc()` returns non-`nullptr`? There's only one memory operation, namely the dereference `m[i]`. Perhaps this dereference is out of bounds.

If no memory error happens when `i == 0`, then a `m[0]` dereference must not cause a memory error. So the `m` object must contain at least 4 bytes. But a memory error *does* happen on Line A when `i == 1`. So the `m` object must contain *less* than 8 bytes. How many bytes were allocated for `m`? `sizeof(point2_vector) * N == sizeof(point2 *) * N == 4 * N`. So we have:

- `(4 * N)` ≥ 4
- `(4 * N)` < 8

It seems like the only possible answer is `N == 1`. But no, this doesn't cause a memory error, because the loop body would never be executed with `i == 1`!

The key insight is that the multiplications above use 32-bit unsigned computer arithmetic. Let's write `N` as `X + 1`. Then these inequalities become:

- 4 ≤ `4 * (X + 1)` = `4 * X + 4` < 8
- 0 ≤ `(4 * X)` < 4

(Multiplication distributes over addition in computer arithmetic.) What values of `X` satisfy this inequality? It might be easier to see if we remember that multiplication by powers of two is equivalent to shifting:

- 0 ≤ `(X << 2)` < 4

The key insight is that this shift eliminates the top two bits of `X`. There are exactly *four* values for `X` that work: `0`, `0x40000000`, `0x80000000`, and `0xC0000000`. For any of these, `4 * X == 0` in 32-bit computer arithmetic, because 4×`X` = 0 (mod 2^{32}) in normal arithmetic.

Plugging `X` back in to `N`, we see that `N ∈ {0x40000001, 0x80000001, 0xC0000001}`. These are the only values that work.

Partial credit was awarded for values that acknowledged the possibility of overflow.

Hide solution

Hide solution

QUESTION DATAREP-8B. Give a value of *N* so that no `new` fails, and no memory error happens on Line A, but a memory error *does* happen on Line B.

Show solution

If no memory error happens on Line A, then N < 2^{30} (otherwise overflow would happen as seen above). But a memory error does happen on Line B. Line B dereferences m[i][j], for 0 ≤ j ≤ i; so how big is m[i]? It was allocated on Line A with size `sizeof(point2)

- (i + 1) == 2 * sizeof(double) * (i + 1) == 16 * (i + 1). If i + 1 ≥ 2³² / 16 = 2²⁸, this multiplication will overflow. Since i < N, we can finally reason that any N greater than or equal to 2²⁸ = 0x10000000 and less than 2³⁰ = 0x40000000` will cause the required memory error.

Hide solution

Hide solution

DATAREP-9. Data representation

Assume a 64-bit x86-64 architecture unless explicitly told otherwise.

Write your assumptions if a problem seems unclear, and write down your reasoning for partial credit.

QUESTION DATAREP-9A. Arrange the following values in increasing numeric order. Assume that `x` is an `int` with value 8192.

1. `EOF`
2. `x & ~x`
3. `(signed char) 0x47F`
4. `x | ~x`
5. `1000`
6. `(signed char) 65535`
7. The size of the stdio cache
8. `-0x80000000`

A possible answer might be “a < b < c = d < e < f < g < h.”

Show solution

h < a = d = f < b < c < e < g

Hide solution

Hide solution

For each of the remaining questions, write one or more arguments that, when passed to the provided function, will cause it to return the integer **61** (which is **0x3d hexadecimal**). Write the expected number of arguments of the expected types.

QUESTION DATAREP-9B.

```
int f1(int n) {
    return 0x11 ^ n;
}
```

Show solution

0x2c == 44

Hide solution

Hide solution

QUESTION DATAREP-9C.

```
int f2(const char* s) {
    return strtol(s, nullptr, 0);
}
```

Show solution

`"61"`

Hide solution

Hide solution

QUESTION DATAREP-9D. Your answer should be different from the previous answer.

```
int f3(const char* s) {
    return strtol(s, nullptr, 0);
}
```

Show solution

`" 0x3d"`, `" 61 "`, etc.

Hide solution

Hide solution

QUESTION DATAREP-9E. For this problem, you will also need to define a global variable. Give its type and value.

```
f4:
    andl $5, %edi
    leal (%rsi,%rdi,2), %eax
    movzbl y(%rip), %ecx
    subl %ecx, %eax
    retq
```

Show solution

This code was compiled from:

```
int f4(int a, int b) {
    extern unsigned char y;
    return (a & 5) * 2 + b - y;
}
```

A valid solution is `a`=0, `b`=61, `unsigned char y`=0.

Hide solution

Hide solution

DATAREP-10. Sizes and alignments

Assume a 64-bit x86-64 architecture unless explicitly told otherwise.

Write your assumptions if a problem seems unclear, and write down your reasoning for partial credit.

QUESTION DATAREP-10A. Use the following members to create a struct of size 16, using each member exactly once, and putting `char a` first; or say "impossible" if this is impossible.

1. `char a;` (we've written this for you)
2. `unsigned char b;`
3. `short c;`
4. `int d;`

```
struct size_16 {
    char a;




};
```

Show solution

Impossible

Hide solution

Hide solution

QUESTION DATAREP-10B. Repeat Part A, but create a struct with size 12.

```
struct size_12 {
    char a;





};
```

Show solution

abdc, acbd, acdb, adbc, adcb, ...

Hide solution

Hide solution

QUESTION DATAREP-10C. Repeat Part A, but create a struct with size 8.

```
struct size_8 {
    char a;





};
```

Show solution

abcd

Hide solution

Hide solution

QUESTION DATAREP-10D. Consider the following structs:

```
struct x {
    T x1;
    U x2;
};
struct y {
    struct x y1;
    V y2;
};
```

Give definitions for T, U, and V so that there is one byte of padding in `struct x` after `x2`, and two bytes of padding in `struct y` after `y1`.

Show solution

Example: T = `short[2]`, U = `char`, V = `int`

Hide solution

Hide solution

DATAREP-11. Dynamic memory allocation

QUESTION DATAREP-11A. True or false?

1. `free(nullptr)` is an error.
2. `malloc(0)` can never return `nullptr`.

Show solution

False, False

Hide solution

Hide solution

QUESTION DATAREP-11B. Give values for `sz` and `nmemb` so that `calloc(sz, nmemb)` will always return `nullptr` (on a 32-bit x86 machine), but `malloc(sz * nmemb)` might or might not return null.

Show solution

`(size_t) -1, (size_t) -1`—anything that causes an overflow

Hide solution

Hide solution

Consider the following 8 statements. (`p` and `q` have type `char*`.)

a. `free(p);`
b. `free(q);`
c. `p = q;`
d. `q = nullptr;`
e. `p = (char*) malloc(12);`
f. `q = (char*) malloc(8);`
g. `p[8] = 0;`
h. `q[4] = 0;`

QUESTION DATAREP-11C. Put the statements in an order that would execute without error or evoking undefined behavior. Memory leaks count as errors. Use each statement **exactly once.** Sample answer: "abcdefgh."

Show solution

cdefghab (and others). Expect "OK"

Hide solution

Hide solution

QUESTION DATAREP-11D. Put the statements in an order that would cause one double-free error, and no other error or undefined behavior (except possibly one memory leak). Use each statement exactly once.

Show solution

efghbcad (and others). Expect "double-free + memory leak"

Hide solution

Hide solution

QUESTION DATAREP-11E. Put the statements in an order that would cause one memory leak (one allocated piece of memory is not freed), and no other error or undefined behavior. Use each statement exactly once.

Show solution

efghadbc (and others). Expect "memory leak"

Hide solution

Hide solution

QUESTION DATAREP-11F. Put the statements in an order that would cause one boundary write error, and no other error or undefined behavior. Use each statement exactly once.

Show solution

> eafhcgbd (and others). Expect "out of bounds write"
>
> Hide solution
>
> Hide solution

DATAREP-12. Pointers and debugging allocators

You are debugging some students' `m61` code from Problem Set 1. The codes use the following metadata:

```
struct meta { ...
    meta* next;
    meta* prev;
};

meta* mhead;    // head of active allocations list
```

Their linked-list manipulations in `m61_malloc` are similar.

```
void* m61_malloc(size_t sz, const char* file, int line) {
    ...
    meta* m = (meta*) ptr;
    m->next = mhead;
    m->prev = nullptr;
    if (mhead) {
        mhead->prev = m;
    }
    mhead = m;
    ...
}
```

But their linked-list manipulations in `m61_free` differ.

Alice’s code:

```
void m61_free(void* ptr, ...) { ...
    meta* m = (meta*) ptr - 1;
    if (m->next != nullptr) {
        m->next->prev = m->prev;
    }
    if (m->prev == nullptr) {
        mhead = nullptr;
    } else {
        m->prev->next = m->next;
    }
    ...
}
```

Bob’s code:

```
void m61_free(void* ptr, ...) { ...
    meta* m = (meta*) ptr - 1;
    if (m->next) {
        m->next->prev = m->prev;
    }
    if (m->prev) {
        m->prev->next = m->next;
    }
    ...
}
```

Chris’s code:

```
void m61_free(void* ptr, ...) { ...
    meta* m = (meta*) ptr - 1;
    m->next->prev = m->prev;
    m->prev->next = m->next;
    ...
}
```

Donna's code:

```
void m61_free(void* ptr, ...) { ...
    meta* m = (meta*) ptr - 1;
    if (m->next) {
        m->next->prev = m->prev;
    }
    if (m->prev) {
        m->prev->next = m->next;
    } else {
        mhead = m->next;
    }
    ...
}
```

You may assume that all code not shown is correct.

QUESTION DATAREP-12A. Whose code will segmentation fault on this input? List all students that apply.

```
int main() {
    void* ptr = malloc(1);
    free(ptr);
}
```

Show solution

Chris

Hide solution

Hide solution

QUESTION DATAREP-12B. Whose code might report something like "`invalid free of pointer [ptr1], not allocated`" on this input? (Because a list traversal starting from `mhead` fails to find `ptr1`.) List all students that apply. Don't include students whose code would segfault before the report.

```
int main() {
    void* ptr1 = malloc(1);
    void* ptr2 = malloc(1);
    free(ptr2);
    free(ptr1);   // <- message printed here
}
```

Show solution

Alice

Hide solution

Hide solution

QUESTION DATAREP-12C. Whose code would improperly report something like "`LEAK CHECK: allocated object [ptr1] with size 1`" on this input? (Because the `mhead` list appears not empty, although it should be.) List all students that apply. Don't include students whose code would segfault before the report.

```
int main() {
    void* ptr1 = malloc(1);
    free(ptr1);
    m61_printleakreport();
}
```

Show solution

Bob

Hide solution

Hide solution

QUESTION DATAREP-12D. Whose linked-list code is correct for all inputs? List all that apply.

Show solution

Donna

Hide solution

Hide solution

DATAREP-13. Arena allocation

Chimamanda Ngozi Adichie is a writing a program that needs to allocate and free a lot of nodes, where a node is defined as follows:

```
struct node {
    int key;
    void* value;
    node* left;
    node* right;        // also used in free list
};
```

She uses an arena allocator variant. Here's her code.

```
struct arena_group {
    arena_group* next_group;
    node nodes[1024];
};

struct arena {
    node* frees;
    arena_group* groups;
};

node* node_alloc(arena* a) {
    if (!a->frees) {
        arena_group* g = new arena_group;
        // ... link `g` to `a->groups` ...
        for (size_t i = 0; i != 1023; ++i) {
            g->nodes[i].right = &g->nodes[i + 1];
        }
        g->nodes[1023].right = nullptr;
        a->frees = &g->nodes[0];
    }
    node* n = a->frees;
    a->frees = n->right;
    return n;
}

void node_free(arena* a, node* n) {
    n->right = a->frees;
    a->frees = n;
}
```

QUESTION DATAREP-13A. True or false?

1. This allocator never has external fragmentation.
2. This allocator never has internal fragmentation.

Show solution

True, True

Hide solution

Hide solution

QUESTION DATAREP-13B. Chimamanda's frenemy Paul Auster notices that if many nodes are allocated right in a row, every 1024th allocation seems much more expensive than the others. The reason is that every 1024th allocation initializes a new group, which in turn adds 1024 nodes to the free list. Chimamanda decides instead to allow a *single* element of the free list to represent *many contiguous free nodes*. The average allocation might get a tiny bit slower, but no allocation will be much slower than average. Here's the start of her idea:

```
node* node_alloc(arena* a) {
    if (!a->frees) {
        arena_group* g = new arena_group;
        // ... link `g` to `a->groups` ...
        g->nodes[0].key = 1024;    // g->nodes[0] is the 1st of 1024 contiguous free
nodes
        g->nodes[0].right = nullptr;
        a->frees = &g->nodes[0];
    }
    node* n = a->frees;
    // ???
    return n;
}
```

Complete this function by writing code to replace `// ???`.

Show solution

```
if (n->key == 1) {
    a->frees = n->right;
} else {
    a->frees = n + 1;
    a->frees->key = n->key - 1;
    a->frees->right = n->right;
}
```

Another solution:

```
if (n->right) {
    a->frees = n->right;
} else if (n->key == 1) {
    a->frees = NULL;
} else {
    a->frees = n + 1;
    a->frees->key = n->key - 1;
}
```

Hide solution

Hide solution

QUESTION DATAREP-13C. Write a `node_free` function that works with the `node_alloc` function from the previous question.

```
void node_free(arena* a, node* n) {



}
```

Show solution

```
void node_free(arena* a, node* n) {
    n->right = a->frees;
    n->key = 1;
    a->frees = n;
}
```

Or, if you use the solution above:

```
void node_free(arena* a, node* n) {
    n->right = a->frees;
    a->frees = n;
}
```

Hide solution

Hide solution

QUESTION DATAREP-13D. Complete the following new function.

```
// Return the arena_group containing node `n`. `n` must be a node returned by
// a previous call to `node_alloc(a)`.
arena_group* node_find_group(arena* a, node* n) {
    for (arena_group* g = a->groups; g; g = g->next_group) {




    }
    return nullptr;
}
```

Show solution

```
arena_group* node_find_group(arena* a, node* n) {
    for (arena_group* g = a->groups; g; g = g->next_group) {
        if ((uintptr_t) &g->nodes[0] <= (uintptr_t) n
            && (uintptr_t) n <= (uintptr_t) &g->nodes[1023]) {
            return g;
        }
    }
    return nullptr;
}
```

Hide solution

Hide solution

QUESTION DATAREP-13E. Chimamanda doesn't like that the `node_find_group` function from part D takes $O(G)$ time, where G is the number of allocated arena_groups. She remembers a library function that might help, `posix_memalign`:

```
int posix_memalign(void** memptr, size_t alignment, size_t size);
```

The function posix_memalign() allocates `size` bytes and places the address of the allocated memory in `*memptr`. The address of the allocated memory will be a multiple of `alignment`, which must be a power of two and a multiple of `sizeof(void*)`. ...

"Cool," she says, "I can use this to speed up `node_find_group`!" She now allocates a new group with the following code:

```
arena_group* g;
int r = posix_memalign(&g, 32768, sizeof(arena_group));
assert(r == 0); // posix_memalign succeeded
```

Given this allocation strategy, write a version of `node_find_group` that takes $O(1)$ time.

```
arena_group* node_find_group(arena* a, node* n) {






}
```

Show solution

```
arena_group* node_find_group(arena* a, node* n) {
    uintptr_t n_addr = (uintptr_t) n;
    return (arena_group*) (n_addr - n_addr % 32768);
}
```

Hide solution

Hide solution

DATAREP-14. Data representation

Sort the following expressions in ascending order by value, using the operators <, =, >. For example, if we gave you:

i. `int A = 6;`
j. `int B = 0x6;`
k. `int C = 3;`

you would write `C < A = B`.

a. `unsigned char a = 0x191;`
b. `char b = 0x293;`
c. `unsigned long c = 0xFFFFFFFF;`
d. `int d = 0xFFFFFFFF;`
e. `int e = d + 3;`
f. f = 4 GB
g. `size_t g = sizeof(*s)` (given `short *s`)
h. `long h = 256;`
i. i = `0b1000000000000000000000000000000000000` (binary)
j. `unsigned long j = 0xACE - 0x101;`

Show solution

`b < d < e = g < a < h < j < c < f < i`

Hide solution

Hide solution

DATAREP-15. Memory

For the following questions, select the part(s) of memory from the list below that best describes where you will find the object.

1. heap
2. stack
3. between the heap and the stack
4. in a read-only data segment
5. in a text segment starting at address 0x08048000
6. in a read/write data segment
7. in a register

Assume the following code, compiled without optimization.

```
#include <stdio.h>
#include <stdlib.h>
const long maxitems = 1000;
struct info {
    char name[20];
    unsigned int age;
    short height;
} s = { "sushi", 1, 9 };

int main(int argc, char* argv[]) {
    static long L = 0xbadf00d;
    unsigned long u = 0x8badf00d;
    int i, num = maxitems + 1;
    struct info *sp;
    printf("What did you do? %lx?\n", u);
    while (num > maxitems || num < 10) {
        printf("How much of it did you eat? ");
        scanf(" %d", &num);
    }
    sp = (struct info *)malloc(num * sizeof(*sp));
    for (i = 0; i < num; i++) {
        sp[i] = s;
    }
    return 0xdeadbeef;
}
```

QUESTION DATAREP-15A. The value 0xdeadbeef, when we are returning from main.

Show solution

7, in a register

Hide solution

Hide solution

QUESTION DATAREP-15B. The variable maxitems

Show solution

4, in a read-only data segment

Hide solution

Hide solution

QUESTION DATAREP-15C. The structure s

Show solution

6, in a read/write data segment

Hide solution

Hide solution

QUESTION DATAREP-15D. The structure at sp[9]

Show solution

1, heap

Hide solution

Hide solution

QUESTION DATAREP-15E. The variable u

Show solution

2, stack, or 7, in a register

Hide solution

Hide solution

QUESTION DATAREP-15F. main

Show solution

5, in a text segment starting at address 0x08048000

Hide solution

Hide solution

QUESTION DATAREP-15G. printf

Show solution

3, between the heap and the stack

Hide solution

Hide solution

QUESTION DATAREP-15H. argc

Show solution

2, stack, or 7, in a register

Hide solution

Hide solution

QUESTION DATAREP-15I. The number the user enters

Show solution

2, stack

Hide solution

Hide solution

QUESTION DATAREP-15J. The variable L

Show solution

6, in a read/write data segment

Hide solution

Hide solution

DATAREP-16. Memory and pointers

⚠️ This question may benefit from Unit 4, kernel programming. ⚠️

If multiple processes are sharing data via `mmap`, they may have the file mapped at different virtual addresses. In this case, pointers to the same object will have different values in the different processes. One way to store pointers in mmapped memory so that multiple processes can access them consistently is using relative pointers. Rather than storing a regular pointer, you store the offset from the beginning of the mmapped region and add that to the address of the mapping to obtain a real pointer. An alternative representation is called self-relative pointers. In this case, you store the difference in address between the current location (i.e., the location containing the pointer) and the location to which you want to point. Neither representation addresses pointers between the mmapped region and the rest of the address space; you may assume such pointers do not exist.

QUESTION DATAREP-16A. State one advantage that relative pointers have over self-relative pointers.

Show solution

The key thing to understand is that both of these approaches use relative pointers and both can be used to solve the problem of sharing a mapped region among processes that might have the region mapped at different addresses.

Possible advantages:

Within a region, you can safely use memcpy as moving pointers around inside the region does not change their value. If you copy a self relative pointer to a new location, its value has to change. That is, imagine that you have a self-relative pointer at offset 4 from the region and it points to the object at offset 64 from the region. The value of the self relative pointer is 60. If I copy that pointer to the offset 100 from the region, I have to change it to be -36. If you save the region as a `uintptr_t` or a `char *`, then you can simply add the offset to the region; self-relative-pointers will always be adding/subtracting from the address of the location storing the pointer, which may have a type other than `char *`, so you'd need to cast it before performing the addition/subtraction.

You can use a larger region: if we assume that we have only N bits to store the pointer, then in the base+offset model, offset could be an unsigned value, which will be larger than the maximum offset possible with a signed pointer, which you need for the self-relative case. That is, although the number of values that can be represented by signed and unsigned numbers differs by one, the implementation must allow for a pointer from the beginning of the region to reference an item at the very last location of the region -- thus, your region size is limited by the largest positive number you can represent.

Hide solution

Hide solution

QUESTION DATAREP-16B. State one advantage that self-relative pointers have over relative pointers.

Show solution

You don't have to know the address at which the region is mapped to use them. That is, given a location containing a self-relative pointer, you can find the target of that pointer.

Hide solution

Hide solution

For the following questions, assume the following setup:

```c
char* region; /* Address of the beginning of the region. */

// The following are sample structures you might find in
// a linked list that you are storing in an mmaped region.

struct ll1 {
     unsigned value;
     TYPE1 r_next; /* Relative Pointer. */
};
struct ll2 {
     unsigned value;
     TYPE2 sr_next; /* Self-Relative Pointer. */
};
ll1 node1;
ll2 node2;
```

QUESTION DATAREP-16C. Propose a type for TYPE1 and give 1 sentence why you chose that type.

Show solution

A good choice is `ptrdiff_t`, which represents differences between pointers. Other reasonable choices include `uintptr_t` and `unsigned long`.

Hide solution

Hide solution

QUESTION DATAREP-16D. Write a C expression to generate a (properly typed) pointer to the element referenced by the r_next field of `ll1`.

Show solution

```c
(ll1*) (region + node1.r_next)
```

Hide solution

Hide solution

QUESTION DATAREP-16E. Propose a type for TYPE2 and give 1 sentence why you chose that type.

Show solution

The same choices work; again `ptrdiff_t` is best.

Hide solution

Hide solution

QUESTION DATAREP-16F. Write a C expression to generate a (properly typed) pointer to the element referenced by the sr_next field of `ll2`.

Show solution

`(ll2*) ((char*) &node2.sr_next + node2.sr_next)`

Hide solution

Hide solution

DATAREP-17. Data representation: Allocation sizes

```
union my_union {
    int f1[4];
    long f2[2];
};

int main() {
    void* p = malloc(sizeof(char*));
    my_union u;
    my_union* up = &u;
    ....
}
```

How much *user-accessible* space is allocated on the stack and/or the heap by each of the following statements? Assume x86-64.

QUESTION DATAREP-17A. `union my_union { ... };`

Show solution

0; this declares the *type*, not any object

Hide solution

Hide solution

QUESTION DATAREP-17B. `void* p = malloc(sizeof(char*));`

Show solution

> 16: 8 on the heap plus 8 on the stack
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-17C. `my_union u;`

Show solution

> 16 (on the stack)
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-17D. `my_union* up = &u;`

Show solution

> 8 (on the stack)
>
> Hide solution
>
> Hide solution

DATAREP-18. Data representation: ENIAC

Professor Kohler has been developing Eddie's NIfty Awesome Computer (ENIAC). When he built the C compiler for ENIAC, he assigned the following sizes and alignments to C's fundamental data types. (Assume that every other fundamental type has the same size and alignment as one of these.)

Type	`sizeof`	`alignof`	
`char`	1	1	
`char*`	16	16	Same for any pointer
`short`	4	4	
`int`	8	8	
`long`	16	16	
`long long`	32	32	
`float`	16	16	

`double`	32	32

QUESTION DATAREP-18A. This set of sizes is valid: it obeys all the requirements set by C's abstract machine. Give one *different* size assignment that would make the set as a whole invalid.

Show solution

Some examples: `sizeof(char) = 0`; `sizeof(char) = 2`; `sizeof(short) = 8` (i.e., longer than `int`); `sizeof(int) = 2` (though not discussed in class, turns out that C++ requires ints are at least 2 bytes big); etc.

Hide solution

Hide solution

QUESTION DATAREP-18B. What alignment must the ENIAC malloc guarantee?

Show solution

32

Hide solution

Hide solution

For the following two questions, assume the following struct on the ENIAC:

```
struct s {
    char f1[7];
    char *f2;
    short f3;
    int f4;
};
```

QUESTION DATAREP-18C. What is `sizeof(struct s)`?

Show solution

`f1` is 7 bytes.

`f2` is 16 bytes with 16-byte alignment, so add 9B padding.

`f3` is 4 bytes (and is already aligned).

`f4` is 8 bytes with 8-byte alignment, so add 4B padding.

That adds up to 7 + 9 + 16 + 4 + 4 + 8 = 16 + 16 + 16 = 48 bytes.

That's a multiple of the structure's alignment, which is 16, so no need for any end padding.

Hide solution

Hide solution

QUESTION DATAREP-18D. What is `alignof(struct s)`?

Show solution

16

Hide solution

Hide solution

The remaining questions refer to this structure definition:

```
// This include file defines a struct inner, but you do not know anything
// about that structure, just that it exists.
#include "inner.hh"

struct outer {
    char f1[3];
    inner f2;
    short f3;
    int f4;
};
```

Indicate for each statement whether the statement is **always** true, **possibly** true, or **never** true on the ENIAC.

QUESTION DATAREP-18E: `sizeof(outer) > sizeof(inner)` (Always / Possibly / Never)

Show solution

Always

Hide solution

Hide solution

QUESTION DATAREP-18F: `sizeof(outer)` is a multiple of `sizeof(inner)` (Always / Possibly / Never)

Show solution

Possibly

Hide solution

Hide solution

QUESTION DATAREP-18G: `alignof(outer) > alignof(struct inner)` (Always / Possibly / Never)

Show solution

Possibly

Hide solution

Hide solution

QUESTION DATAREP-18H: `sizeof(outer) - sizeof(inner) < 4` (Always / Possibly / Never)

Show solution

Never

Hide solution

Hide solution

QUESTION DATAREP-18I: `sizeof(outer) - sizeof(inner) > 32` (Always / Possibly / Never)

Show solution

Possibly

Hide solution

Hide solution

QUESTION DATAREP-18J: `alignof(inner) == 2` (Always / Possibly / Never)

Show solution

Never

Hide solution

Hide solution

DATAREP-19. Undefined behavior

Which of the following expressions, instruction sequences, and code behaviors cause undefined behavior? For each question, write Defined or Undefined. (Note that the `INT_MAX` and `UINT_MAX` constants have types `int` and `unsigned`, respectively.)

QUESTION DATAREP-19A. `INT_MAX + 1` (Defined / Undefined)

Show solution

Undefined

Hide solution

Hide solution

QUESTION DATAREP-19B. `UINT_MAX + 1` (Defined / Undefined)

Show solution

Defined

Hide solution

Hide solution

QUESTION DATAREP-19C.

```
movq $0x7FFFFFFFFFFFFFFF, %rax
addl $1, %rax
```

(Defined / Undefined)

Show solution

Defined (only C++ programs can have undefined behavior; the behavior of x86-64 instructions is always defined)

Hide solution

Hide solution

QUESTION DATAREP-19D. Failed memory allocation, i.e., `malloc` returns `nullptr` (Defined / Undefined)

Show solution

Defined

Hide solution

Hide solution

QUESTION DATAREP-19E. Use-after-free (Defined / Undefined)

Show solution

Undefined

Hide solution

Hide solution

QUESTION DATAREP-19F. Here are two functions and a global variable:

```
const char string[128] = ".......";
int read_nth_char(int n) {
    return string[n];
}
int f(int i) {
    if (i & 0x40) {
        return read_nth_char(i * 2);
    } else {
        return i * 2;
    }
}
```

C's undefined behavior rules would allow an aggressive optimizing compiler to simplify the code generated for `f`. Fill in the following function with the simplest C code you can, under the constraint that an aggressive optimizing compiler might generate the same object code for `f` and `f_simplified`.

```
int f_simplified(int i) {



}
```

Show solution

`return i * 2;`

Hide solution

Hide solution

DATAREP-20. Bit manipulation

It's common in systems code to need to switch data between big-endian and little-endian representations. This is because networks represent multi-byte integers using big-endian representation, whereas x86-family processors store multi-byte integers using little-endian representation.

QUESTION DATAREP-20A. Complete this function, which translates an integer from big-endian representation to little-endian representation by swapping bytes. For instance, `big_to_little(0x01020304)` should return `0x04030201`. Your return statement **must** refer to the `u.c` array, and **must not** refer to `x`. This function is compiled on x86-64 Linux (as every function is unless we say otherwise).

```
unsigned big_to_little(unsigned x) {
    union {
        unsigned intval;
        unsigned char c[4];
    } u;
    u.intval = x;



    return ______________________________________;
}
```

Show solution

```
return (u.c[0] << 24) | (u.c[1] << 16) | (u.c[2] << 8) | u.c[3];
```

Hide solution

Hide solution

QUESTION DATAREP-20B. Complete the function again, but this time write a single expression that refers to `x` (you may refer to `x` multiple times, of course).

```
unsigned big_to_little(unsigned x) {



    return ______________________________________;
}
```

Show solution

```
    return ((x & 0xFF) << 24) | ((x & 0xFF00) << 8) | ((x & 0xFF0000) >> 8) | (x
>> 24);
```

Hide solution

Hide solution

QUESTION DATAREP-20C. Now write the function `little_to_big`, which will translate a little-endian integer into big-endian representation. You may introduce helper variables or even call `big_to_little` if that's helpful.

```
unsigned little_to_big(unsigned x) {






}
```

Show solution

```
    return big_to_little(x);
```

Hide solution

Hide solution

DATAREP-21. Computer arithmetic

Bitwise operators and computer arithmetic can represent *vectors* of bits, which in turn are useful for representing *sets*. For example, say we have a function `bit` that maps elements to distinct bits; thus, `bit(X) == (1 << i)` for some `i`. Then a set {X0, X1, X2, …, Xn} can be represented as `bit(X0) | bit(X1) | bit(X2) | … | bit(Xn)`. Element Xi is in the set with integer representation `z` if and only if `(bit(Xi) & z) != 0`.

QUESTION DATAREP-21A. What is the maximum number of set elements that can be represented in a single `unsigned` variable on an x86 machine?

Show solution

32

Hide solution

Hide solution

QUESTION DATAREP-21B. Match each set operation with the C operator(s) that could implement that operation. (Complement is a unary operation.)

intersection	`==`
equality	`~`
complement	`&`
union	`^`

toggle membership (flip whether an element is in the set) |

Show solution

intersection	&
equality	==
complement	~
union	|
toggle membership	^

Hide solution

Hide solution

QUESTION DATAREP-21C. Complete this function, which should return the *set difference* between the sets with representations a and b. This is the set containing exactly those elements of set a that are **not** in set b.

```
unsigned set_difference(unsigned a, unsigned b) {






}
```

Show solution

Any of these work:

```
    return a & ~b;
    return a - (a & b);
    return a & ~(a & b);
```

Hide solution

Hide solution

QUESTION DATAREP-21D. Below we've given a number of C++ expressions, some of their values, and some of their set representations for a set of elements. For example, the first row says that the integer value of expression `0` is just 0, which corresponds to an empty set. Fill in the blanks. This will require figuring out which bits correspond to the set elements `A`, `B`, `C`, and `D`, and the values for the 32-bit `int` variables `a`, `x`, and `s`. No arithmetic operation overflows; `abs(x)` returns the absolute value of `x` (that is, `x < 0 ? -x : x`).

Expression e	Integer value	Represented set
`0`	0	`{}`
`a == a`	____________	`{A}`
`(unsigned) ~a < (unsigned) a`	____________	`{A}`
`a < 0`	____________	____________
`(1 << (s/2)) - 1`	____________	`{A,B,C,D}`
`a * a`	____________	`{C}`
`abs(a)`	____________	____________
`x & (x - 1)`	____________	`{}`
`x - 1`	____________	`{A,D}`
`x`	____________	____________
`s`	____________	____________

Show solution

Expression e	Integer value	Represented set
`0`	0	`{}`
`a == a`	1	`{A}`
`(unsigned) ~a < (unsigned) a`	1	`{A}`
`a < 0`	1	`{A}`
`(1 << (s/2)) - 1`	15	`{A,B,C,D}`
`a * a`	4	`{C}`
`abs(a)`	2	`{D}`
`x & (x - 1)`	0	`{}`
`x - 1`	3	`{A,D}`
`x`	4	`{C}`
`s`	8	`{B}`

Hide solution

Hide solution

DATAREP-22. Bit Tac Toe

Brenda Bitdiddle is implementing tic-tac-toe using bitwise arithmetic. (If you're unfamiliar with tic-tac-toe, see below.) Her implementation starts like this:

```
struct tictactoe {
    unsigned moves[2];
};
#define XS 0
#define OS 1

void tictactoe_init(tictactoe* b) {
    b->moves[XS] = b->moves[OS] = 0;
}

static const unsigned ttt_values[3][3] = {
    { 0x001, 0x002, 0x004 },
    { 0x010, 0x020, 0x040 },
    { 0x100, 0x200, 0x400 }
};

    // Mark a move by player `p` at row `row` and column `col`.
    // Return 0 on success; return -1 if position `row,col` has already been used.
    int tictactoe_move(tictactoe* b, int p, int row, int col) {
1.      assert(row >= 0 && row < 3 && col >= 0 && col < 3);
2.      assert(p == XS || p == OS);
3.      /* TODO: check for position reuse */
4.      b->moves[p] |= ttt_values[row][col];
5.      return 0;
    }
```

Each position on the board is assigned a distinct bit.

> Tic-tac-toe, also known as noughts and crosses, is a simple paper-and-pencil game for two players, X and O. The board is a 3x3 grid. The players take turns writing their symbol (X or O) in an empty square on the grid. The game is won when one player gets their symbol in all three squares in one of the rows, one of the columns, or one of the two diagonals. X goes first; played perfectly, the game always ends in a draw.
>
> You may access the Wikipedia page for tic-tac-toe.

QUESTION DATAREP-22A. Brenda’s current code doesn’t check whether a move reuses a position. Write a snippet of C code that returns –1 if an attempted move is reusing a position. This snippet will replace line 3.

Show solution

Lots of people misinterpreted this to mean the player reused *their own* position and ignored the other player. That mistake was allowed with no points off. The code below checks whether any position was reused by *either* player.

```
if ((b->moves[XS] | b->moves[OS]) & ttt_values[row][col]) {
    return -1;
}

OR
if ((b->moves[XS] | b->moves[OS] | ttt_values[row][col]) == (b->moves[XS] | b->moves[OS])) {
    return -1;
}

OR
if ((b->moves[XS] + b->moves[OS]) & ttt_values[row][col]) {
    return -1;
}

OR
if ((b->moves[p] ^ ttt_values[row][col]) < b->moves[p]) {
    return -1;
}
```

etc.

Hide solution

Hide solution

QUESTION DATAREP-22B. Complete the following function. You may use the following helper function:

- **`int popcount(unsigned n)`**
  Return the number of 1 bits in `n`. (Stands for "population count"; is implemented on recent x86 processors by a single instruction, `popcnt`.)

For full credit, your code should consist of a single "`return`" statement with a simple expression, but for substantial partial credit write any correct solution.

```
// Return the number of moves that have happened so far.
int tictactoe_nmoves(const tictactoe* b) {









}
```

Show solution

```
return popcount(b->moves[XS] | b->moves[OS]);
```

Hide solution

Hide solution

QUESTION DATAREP-22C. Write a simple expression that, if nonzero, indicates that player `XS` has a win on board `b` across the main diagonal (has marks in positions `0,0`, `1,1`, and `2,2`).

Show solution

```
(b->moves[XS] & 0x421) == 0x421
```

Hide solution

Hide solution

Lydia Davis notices Brenda's code and has a brainstorm. "If you use different values," she suggests, "it becomes easy to detect any win." She suggests:

```
static const unsigned ttt_values[3][3] = {
    { 0x01001001, 0x00010002, 0x10100004 },
    { 0x00002010, 0x22020020, 0x00200040 },
    { 0x40004100, 0x00040200, 0x04400400 }
};
```

QUESTION DATAREP-22D. Repeat part A for Lydia's values: Write a snippet of C code that returns –1 if an attempted move is reusing a position. This snippet will replace line 3 in Brenda's code.

Show solution

The same answers as for part A work.

Hide solution

Hide solution

QUESTION DATAREP-22E. Repeat part B for Lydia's values: Use `popcount` to complete `tictactoe_nmoves`.

```
int tictactoe_nmoves(const tictactoe* b) {









}
```

Show solution

Either of:

```
return popcount((b->moves[0] | b->moves[1]) & 0x777);
return popcount((b->moves[0] | b->moves[1]) & 0x777000);
```

Hide solution

Hide solution

QUESTION DATAREP-22F. Complete the following function for Lydia's values. For full credit, your code should consist of a single "`return`" statement containing exactly two constants, but for substantial partial credit write any correct solution.

```
// Return nonzero if player `p` has won, 0 if `p` has not won.
int tictactoe_check_win(const tictactoe* b, int p) {
    assert(p == XS || p == OS);








}
```

Show solution

```
return (b->moves[p] + 0x11111111) & 0x88888888;

// Another amazing possibility (Allen Chen and others):
return b->moves[p] & (b->moves[p] << 1) & (b->moves[p] << 2);
```

Hide solution

Hide solution

DATAREP-23. Memory and Pointers

Two processes are mapping a file into their address space. The mapped file contains an unsorted linked list of integers. As the processes cannot ensure that the file will be mapped at the same virtual address, they use *relative pointers* to link elements in the list. A relative pointer holds not an address, but an *offset* that user code can use to calculate a true address. Our processes define the offset as relative to **the start of the file.**

Thus, each element in the linked list is represented by the following structure:

```
struct ll_node {
    int value;
    size_t offset;
};
```

`offset == (size_t) -1` indicates the end of the list. Other `offset` values represent the position of the next item in the list, calculated relative to the start of the file.

QUESTION DATAREP-23A. Write a function to find an item in the list. The function's prototype is:

```
ll_node* find_element(void* mapped_file, ll_node* list, int value);
```

The `mapped_file` parameter is the address of the mapped file data; the `list` parameter is a pointer to the first node in the list; and the `value` parameter is the value for which we are searching. The function should return a pointer to the linked list element if the value appears in the list or nullptr if the value is not in the list.

Show solution

```
ll_node* find_element(void* mapped_file, ll_node* list, int value) {
    while (1) {
        if (list->value == value)
            return list;
        if (list->offset == (size_t) -1)
            return NULL;
        list = (ll_node*) ((char*) mapped_file + list->offset);
    }
}
```

Hide solution

Hide solution

DATAREP-24. Integer representation

Write the value of the variable or expression in each problem, using signed decimal representation.

For example, if we gave you:

A. `int i = 0xA;`
B. `int j = 0xFFFFFFFF;`

you would write A) 10 B) -1.

QUESTION DATAREP-24A. `int i = 0xFFFF;` (You may write this either in decimal or as an expression using a power of 2)

Show solution

2^{16} - 1 or 65535

Hide solution

Hide solution

QUESTION DATAREP-24B. `short s = 0xFFFF;` (You may write this either in decimal or as an expression using a power of 2)

Show solution

-1

Hide solution

Hide solution

QUESTION DATAREP-24C. `unsigned u = 1 << 10;`

Show solution

1024 or 2^{10}

Hide solution

Hide solution

QUESTION DATAREP-24D. ⚠️ From WeensyOS: `unsigned long l = PTE_P | PTE_U;`

Show solution

5

Hide solution

Hide solution

QUESTION DATAREP-24E. `int j = ~0;`

Show solution

-1

Hide solution

Hide solution

QUESTION DATAREP-24F. ⚠️ From WeensyOS: `sizeof(x86_64_pagetable);`

Show solution

4096 or 2^{12}

Hide solution

Hide solution

QUESTION DATAREP-24G. Given this structure:

```
struct s {
    char c;
    short s;
    long l;
};
s* ps;
```

This expression: `sizeof(ps);`

Show solution

TRICK QUESTION! 8

Hide solution

Hide solution

QUESTION DATAREP-24H. Using the structure above: `sizeof(*ps);`

Show solution

16

Hide solution

Hide solution

QUESTION DATAREP-24I. `unsigned char u = 0xABC;`

Show solution

`0xBC == 11*16 + 12 == 160 + 16 + 12 == 188`

Hide solution

Hide solution

QUESTION DATAREP-24J. `signed char c = 0xABC;`

Show solution

> 0xBC has most-significant bit on, so the value as a signed char is less than zero. We seek `x` so that `0xBC + x == 0x100`. The answer is 0x44: `0xBC + 4 == 0xC0`, and `0xC0 + 0x40 == 0x100`. So `-0x44 == -4*16 - 4 == -68`.
>
> Hide solution
>
> Hide solution

DATAREP-25. Data representation

In gdb, you observe the following values for a set of memory locations.

```
0x100001020:   0xa0   0xb1   0xc2   0xd3    0xe4   0xf5   0x06   0x17
0x100001028:   0x28   0x39   0x4a   0x5b    0x6c   0x7d   0x8e   0x9f
0x100001030:   0x89   0x7a   0x6b   0x5c    0x4d   0x3e   0x2f   0x10
0x100001038:   0x01   0xf2   0xe3   0xd4    0xc5   0xb6   0xa7   0x96
```

For each C expression below, write its value in hexadecimal. For example, if we gave you:

```
char *cp = (char*) 0x100001020; cp[0] =
```

the answer would be `0xa0`.

Assume the following structure and union declarations and variable definitions.

```c
struct _s1 {
	int i;
	long l;
	short s;
};

struct _s2 {
	char c[4];
	int i;
	struct _s1 s;
};

union _u {
	char c[8];
	int i;
	long l;
	short s;
};

char* cp = (char*) 0x100001020;
struct _s1* s1 = (struct _s1*) 0x100001020;
struct _s2* s2 = (struct _s2*) 0x100001020;
union _u* u = (union _u*) 0x100001020;
```

QUESTION DATAREP-25A. `cp[4] =`

Show solution

0xE4 (-28)

Hide solution

Hide solution

QUESTION DATAREP-25B. `cp + 7 =`

Show solution

0x100001027

Hide solution

Hide solution

QUESTION DATAREP-25C. `s1 + 1 =`

Show solution

0x100001038

Hide solution

Hide solution

QUESTION DATAREP-25D. `s1->i =`

Show solution

0xd3c2b1a0 (-742215264)

Hide solution

Hide solution

QUESTION DATAREP-25E. `sizeof(s1) =`

Show solution

8

Hide solution

Hide solution

QUESTION DATAREP-25F. `&s2->s =`

Show solution

0x100001028

Hide solution

Hide solution

QUESTION DATAREP-25G. `&u->s =`

Show solution

0x100001020

Hide solution

Hide solution

QUESTION DATAREP-25H. `s1->l =`

Show solution

0x9f8e7d6c5b4a3928 (-6949479270644565720)

Hide solution

Hide solution

QUESTION DATAREP-25I. `s2->s.s =`

Show solution

0xf201 (-3583)

Hide solution

Hide solution

QUESTION DATAREP-25J. `u->l =`

Show solution

0x1706f5e4d3c2b1a0 (1659283875886707104)

Hide solution

Hide solution

DATAREP-26. Sizes and alignments

Here's a test struct with *n* members. Assume an x86-64 machine, where each `Ti` either is a basic x86-64 type (e.g., `int`, `char`, `double`) or is a type derived from such types (e.g., arrays, structs, pointers, unions, possibly recursively), and assume that $a_i \le 8$ for all *i*.

```
struct test {
    T1 m1;      // sizeof(T1) == s1, alignof(T1) == a1
    T2 m2;      // sizeof(T2) == s2, alignof(T2) == a2
    ...
    Tn mn;      // sizeof(Tn) == sn, alignof(Tn) == an
};
```

In these questions, you will compare this struct with other structs that have the same members, but in other orders.

QUESTION DATAREP-26A. True or false: The size of `struct test` is minimized when its members are sorted by size. In other words, if $s1 \le s2 \le \ldots \le sn$, then `sizeof(struct test)` is less than or equal to the struct size for any other member order.

If true, briefly explain your answer; if false, give a counterexample (i.e., concrete types for `T1`, …, `Tn` that do not minimize `sizeof(struct test)`).

Show solution

> False. T1 = `char`, T2 = `int`, T3 = `char[5]`
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-26B. True or false: The size of `struct test` is minimized when its members are sorted by alignment. In other words, if $a1 \le a2 \le \ldots \le an$, then `sizeof(struct test)` is less than or equal to the struct size for any other member order.

If true, briefly explain your answer; if false, give a counterexample.

Show solution

> True. Padding only occurs between objects with different alignments, and is limited by the second alignment; sorting by alignment therefore minimizes padding.
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-26C. True or false: The **alignment** of `struct test` is minimized when its members are sorted in increasing order by alignment. In other words, if $a1 \le a2 \le \ldots \le an$, then `alignof(struct test)` is less than or equal to the struct alignment for any other member order.

If true, briefly explain your answer; if false, give a counterexample.

Show solution

> True. It's all the same; alignment is max alignment of every component, and is independent of order.
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-26D. What is the maximum number of bytes of padding that `struct test` could contain for a given *n*? The answer will be a pretty simple formula involving *n*. (Remember that $a_i \le 8$ for all *i*.)

Show solution

> Alternating `char` and `long` gives the most padding, which is 7*(n/2) when n is even and 7*(n+1)/2 otherwise.
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-26E. What is the minimum number of bytes of padding that `struct test` could contain for a given *n*?

Show solution

> 0
>
> Hide solution
>
> Hide solution

DATAREP-27. Undefined behavior

QUESTION DATAREP-27A. Sometimes a conforming C compiler can assume that `a + 1 > a`, and sometimes it can't. For each type below, consider this expression:

```
a + (int) 1 > a
```

and say whether the compiler:

- **Must reject** the expression as a type error.
- **May assume** that the expression is true (that `a + (int) 1 > a` for all `a`).
- **Must not assume** that the expression is true.

1. `int a`

2. `unsigned a`
3. `char* a`
4. `unsigned char a`
5. `struct {int m;} a`

Show solution

> 1—May assume; 2—Must not assume; 3—May assume; 4—May assume (in fact due to integer promotion, this statement really is always true, even in mathematical terms); 5—Must reject.
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-27B. The following code checks its arguments for sanity, but not well: each check can cause undefined behavior.

```
void sanity_check(int* array, size_t array_size, int* ptr_into_array) {
    if (array + array_size < array) {
        fprintf(stderr, "`array` is so big that it wraps around!\n");
        abort();
    }
    if (ptr_into_array < array || ptr_into_array > array + array_size) {
        fprintf(stderr, "`ptr_into_array` doesn't point into the array!\n");
        abort();
    }
    ...
```

Rewrite these checks to avoid all undefined behavior. You will likely add one or more casts to `uintptr_t`. For full credit, write each check as a *single* comparison (no `&&` or `||`, even though the current `ptr_into_array` check uses `||`).

`array_size` check:

`ptr_into_array` check:

Show solution

`array_size` check: `(uintptr_t) array + 4 * array_size < (uintptr_t) array`

`ptr_into_array` check: `(uintptr_t) ptr_into_array - (uintptr_t) array > 4 * array_size`

Hide solution

Hide solution

QUESTION DATAREP-27C. In lecture, we discussed several ways to tell if a signed integer `x` is negative. One of them was the following:

```
int isnegative = (x & (1UL << (sizeof(x) * CHAR_BIT))) != 0;
```

But this is incorrect: it has undefined behavior. Correct it by adding two characters.

Show solution

`(x & (1UL << (sizeof(x) * CHAR_BIT - 1))) != 0`

Hide solution

Hide solution

DATAREP-28. Memory errors and garbage collection

⚠️ We didn't discuss garbage collectors in class this year. ⚠️

Recall that a *conservative garbage collector* is a program that can automatically free dynamically-allocated memory by detecting when that memory is no longer referenced. Such a GC works by scanning memory for currently-referenced pointers, starting from stack and global memory, and recursing over each referenced object until all referenced memory has been scanned. We built a conservative garbage collector in lecture datarep6.

QUESTION DATAREP-28A. An application program that uses conservative GC, and does not call `free` directly, will avoid certain errors and undefined behaviors. Which of the following errors are avoided? List all that apply.

1. Use-after-free
2. Double free
3. Signed integer overflow
4. Boundary write error
5. Unaligned access

Show solution

1, 2

Hide solution

Hide solution

QUESTION DATAREP-28B. Write a C program that leaks unbounded memory without GC, but does not do so with GC. You should need less than 5 lines. (Leaking "unbounded" memory means the program will exhaust the memory capacity of any machine on which it runs.)

Show solution

```
while (1) {
    (void) malloc(1);
}
```

Hide solution

Hide solution

QUESTION DATAREP-28C. Not every valid C program works with a conservative GC, because the C abstract machine allows a program to manipulate pointers in strange ways. Which of the following pointer manipulations might cause the conservative GC from class to inappropriately free a memory allocation? List all that apply.

1. Storing the pointer in a `uintptr_t` variable
2. Writing the pointer to a disk file and reading it back later
3. Using the least-significant bit of the pointer to store a flag:

```
int* set_ptrflag(int* x, int flagval) {
    return (int*) ((uintptr_t) x | (flagval ? 1 : 0));
}
int get_ptrflag(int* x) {
    return (uintptr_t) x & 1;
}
int deref_ptrflag(int* x) {
    return *((int*) ((uintptr_t) x & ~1UL));
}
```

4. Storing the pointer in textual form:

```c
void save_ptr(char buf[100], void* p) {
    sprintf(buf, "%p", p);
}
void* restore_ptr(const char buf[100]) {
    void* p;
    sscanf(buf, "%p", &p);
    return p;
}
```

5. Splitting the pointer into two parts and storing the parts in an array:

```c
typedef union {
    unsigned long ival;
    unsigned arr[2];
} value;
value save_ptr(void* p) {
    value v;
    v.arr[0] = (uintptr_t) p & 0xFFFFFFFFUL;
    v.arr[1] = ((uintptr_t) p / 4294967296UL) & 0xFFFFFFFFUL;
    return v;
}
```

Show solution

2, 4

Hide solution

Hide solution

DATAREP-29. Bitwise

QUESTION DATAREP-29A. Consider this C fragment:

```c
uintptr_t x = ...;
uintptr_t r = 0;
if (a < b) {
    r = x;
}
```

Or, shorter:

```c
uintptr_t r = a < b ? x : 0;
```

Write a single expression that evaluates to the same value, but that **does not** use the conditional ?: operator. You will use the fact that `a < b` always equals 0 or 1. For full credit, do not use expensive operators (multiply, divide, modulus).

Show solution

> Examples: `(a < b) * x`, `(-(uintptr_t) (a < b)) & x`
>
> Hide solution
>
> Hide solution

QUESTION DATAREP-29B. This function returns one more than the index of the least-significant 1 bit in its argument, or 0 if its argument is zero.

```
int ffs(unsigned long x) {
    for (int i = 0; i < sizeof(x) * CHAR_BIT; ++i) {
        if (x & (1UL << i)) {
            return i + 1;
        }
    }
    return 0;
}
```

This function runs in $O(B)$ time, where B is the number of bits in an `unsigned long`. Write a version of `ffs` that runs instead in $O(\log B)$ time.

Show solution

```c
int ffs(unsigned long x) {
    if (!x) {
        return 0;
    }
    int ans = 1;
    if (!(x & 0x00000000FFFFFFFFUL)) {
        ans += 32; x >>= 32;
    }
    if (!(x & 0x0000FFFF)) {
        ans += 16; x >>= 16;
    }
    if (!(x & 0x00FF)) {
        ans += 8; x >>= 8;
    }
    if (!(x & 0x0F)) {
        ans += 4; x >>= 4;
    }
    if (!(x & 0x3)) {
        ans += 2; x >>= 2;
    }
    return ans + (x & 0x1 ? 0 : 1);
}
```

Hide solution

Hide solution

DATAREP-30. Data representation

QUESTION DATAREP-30A. Write a type whose size is 19,404,329 times larger than its alignment.

Show solution

`char[19404329]`

Hide solution

Hide solution

QUESTION DATAREP-30B. Consider a structure type `T` with *N* members, all of which have nonzero size. Assume that `sizeof(T) == alignof(T)`. What is *N*?

Show solution

1

Hide solution

Hide solution

QUESTION DATAREP-30C. What is a C type that obeys `(T) -1 == (T) 255` on x86-64?

Show solution

`char` (or `unsigned char` or `signed char`)

Hide solution

Hide solution

Parts D–G use this code. The architecture *might or might not* be x86-64.

```
unsigned char a[] = {
    0x7A, 0xEC, 0x0D, 0xBE, 0x99, 0x0A, 0xD8, 0x0E
};
unsigned* s1 = (unsigned*) &a[0];
unsigned* s2 = s1 + 1;
```

Assume that `(uintptr_t) s2 - (uintptr_t) s1 == 4` and `*s1 > *s2`.

QUESTION DATAREP-30D. What is `sizeof(a)`?

Show solution

8

Hide solution

Hide solution

QUESTION DATAREP-30E. What is `sizeof(unsigned)` on this architecture?

Show solution

4

Hide solution

Hide solution

QUESTION DATAREP-30F. Is this architecture big-endian or little-endian?

Show solution

Little-endian

Hide solution

Hide solution

QUESTION DATAREP-30G. Might the architecture be x86-64?

Show solution

Yes

Hide solution

Hide solution

DATAREP-31. Memory errors

Mavis Gallant is starting on her debugging memory allocator. She's written code that aims to detect invalid frees, where a pointer passed to `m61_free` was not returned by an earlier `m61_malloc`.

```
D1.    typedef struct m61_metadata {
D2.        size_t magic;
D3.        size_t padding;
D4.    } m61_metadata;

M1.    void* m61_malloc(size_t sz) {
M2.        m61_metadata* meta = base_malloc(sz + sizeof(m61_metadata));
M3.        meta->magic = 0x84157893401;
M4.        return (void*) (meta + 1);
M5.    }

F1.    void m61_free(void* ptr) {
F2.        m61_metadata* meta = (m61_metadata*) ptr - 1;
F3.        if (meta->magic != 0x84157893401) {
F4.            fprintf(stderr, "invalid free of %p\n", ptr);
F5.            abort();
F6.        }
F7.        base_free(ptr);
F8.    }

C1.    void* m61_calloc(size_t count, size_t sz) {
C2.        void* p = m61_malloc(sz * count);
C3.        memset(p, 0, sz * count);
C4.        return p;
C5.    }`
```

Help her track down bugs.

QUESTION DATAREP-31A. What is `sizeof(struct m61_metadata)`?

Show solution

16

Hide solution

Hide solution

QUESTION DATAREP-31B. Give an `m61_` function call (function name and arguments) that would cause both unsigned integer overflow and invalid memory accesses.

Show solution

`m61_malloc((size_t) -15)`. This turns into `malloc(1)` and the dereference of `meta->magic` becomes invalid.

Hide solution

Hide solution

QUESTION DATAREP-31C. Give an `m61_` function call (function name and arguments) that would cause integer overflow, but no invalid memory access *within the `m61_` functions*. (The application might or might not make an invalid memory access later.)

Show solution

`m61_malloc((size_t) -1)`

Hide solution

Hide solution

QUESTION DATAREP-31D. These functions have some potential null pointer dereferences. Fix one such problem, including the line number(s) where your code should go.

Show solution

C3: `if (p) { memset(p, 0, sz * count); }`

Hide solution

Hide solution

QUESTION DATAREP-31E. Put **a single line of C code** in the blank. The resulting program should (1) be well-defined with no memory leaks when using default `malloc`/`free`/`calloc`, but (2) always cause undefined behavior when using Mavis's debugging `malloc`/`free`/`calloc`.

```
... #includes ...
int main(void) {



    ________________________
}
```

Show solution

```
free(nullptr);
```

Hide solution

Hide solution

QUESTION DATAREP-31F. A double free should print a different message than an invalid free. Write code so Mavis's implementation does this; include the line numbers where the code should go.

Show solution

F4: `fprintf(stderr, meta->magic == 0xB0B0B0B0 ? "double free of %p" : "invalid free of %p", ptr)`

after F6: `meta->magic = 0xB0B0B0B0;`

Hide solution

Hide solution

Assembly exercises

Many exercises that seem less appropriate this year, or which cover topics that we haven't covered in class, are marked with ⚠️. However, we may have missed some.

ASM-1. Disassemble

Here's some assembly produced by compiling a C program.

```
        .globl  f
        .align  16, 0x90
        .type   f,@function
f:
        movl    $1, %r8d
        jmp     .LBB0_1
.LBB0_6:
        incl    %r8d
.LBB0_1:
        movl    %r8d, %ecx
        imull   %ecx, %ecx
        movl    $1, %edx
.LBB0_2:
        movl    %edx, %edi
        imull   %edi, %edi
        movl    $1, %esi
        .align  16, 0x90
.LBB0_3:
        movl    %esi, %eax
        imull   %eax, %eax
        addl    %edi, %eax
        cmpl    %ecx, %eax
        je      .LBB0_7
        cmpl    %edx, %esi
        leal    1(%rsi), %eax
        movl    %eax, %esi
        jl      .LBB0_3
        cmpl    %r8d, %edx
        leal    1(%rdx), %eax
        movl    %eax, %edx
        jl      .LBB0_2
        jmp     .LBB0_6
.LBB0_7:
        pushq   %rax
.Ltmp0:
        movl    $.L.str, %edi
        xorl    %eax, %eax
        callq   printf
        movl    $1, %eax
        popq    %rcx
        retq

        .type   .L.str,@object
.L.str:
        .asciz  "%d %d\n"
        .size   .L.str, 7
```

QUESTION ASM-1A. How many arguments might this function have? Circle all that apply.

1. 0
2. 1
3. 2
4. 3 or more

Show solution

> All (1–4). The function has no arguments *that it uses*, but it might have arguments it doesn't use.
>
> Hide solution
>
> Hide solution

QUESTION ASM-1B. What might this function return? Circle all that apply.

1. 0
2. 1
3. −1
4. Its first argument, whatever that argument is
5. A square number other than 0 or 1
6. None of the above

Show solution

> It can only return 1.
>
> Hide solution
>
> Hide solution

QUESTION ASM-1C. Which callee-saved registers does this function save and restore? Circle all that apply.

1. %rax
2. %rbx
3. %rcx
4. %rdx
5. %rbp
6. %rsi
7. %rdi
8. None of the above

Show solution

The callee-saved registers are %rbx, %rbp, %rsp, and %r12-%r15. The code does not modify any of these registers, so it doesn't "save and restore" them either.

Hide solution

Hide solution

QUESTION ASM-1D. This function handles signed integers. If we changed the C source to use *unsigned* integers instead, which instructions would change? Circle all that apply.

1. `movl`
2. `imull`
3. `addl`
4. `cmpl`
5. `je`
6. `jl`
7. `popq`
8. None of the above

Show solution

`jl`

We accepted circled `imull` or not! Although x86 `imull` is signed, as used in C it behaves equivalently to the nominally-unsigned `mull`, and some compilers use `imull` for both kinds of integer. From the Intel manuals:

> "[These] forms [of `imul`] may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero."

Hide solution

Hide solution

QUESTION ASM-1E. What might this function print? Circle all that apply.

1. `0 0`
2. `1 1`
3. `3 4`
4. `4 5`
5. `6 8`
6. None of the above

Show solution

Choice #3 (`3 4`) only. The function searches for a solution to $x^2 + y^2 == z^2$, under the constraint that `x ≤ y`. When it finds one, it prints `x` and `y` and then returns. It always starts from `1 1` and increments `x` and `y` one at a time, so it can only print `3 4`.

Hide solution

Hide solution

ASM-2. Assembly

Here is some x86 assembly code.

```
f:
        movl a, %eax
        movl b, %edx
        andl $255, %edx
        subl %edx, %eax
        movl %eax, a
        retq
```

QUESTION ASM-2A. Write valid C code that could have compiled into this assembly (i.e., write a C definition of function `f`), given the global variable declarations "`extern unsigned a, b;`." Your C code should compile without warnings. **REMINDER:** You are not permitted to run a C compiler, except for the C compiler that is your brain.

Show solution

```
void f() {
    a -= b & 255;
}
```

Or see below for more possibilities.

Hide solution

Hide solution

QUESTION ASM-2B. Write *different* valid, warning-free C code that could have compiled into that assembly. This version should contain different operators than your first version. (For extra credit, use *only one operator*.)

Show solution

```
void f() {
    a += -(b % 256);
}
```

Hide solution

Hide solution

QUESTION ASM-2C. Again, write *different* valid, warning-free C code that could have compiled into that assembly. In this version, `f` should have a different type than in your first version.

Show solution

```
unsigned f() {
    a = a - b % 0x100;
    return a;
}
unsigned f() {
    a -= (unsigned char) b;
    return a;
}
char* f(int x, int y, int z[1000]) {
    a -= (unsigned char) b;
    return (char*) a;
}
```

Hide solution

Hide solution

ASM-3. Assembly and Data Structures

For each code sample below, indicate the most likely type of the data being accessed. (If multiple types are equally likely, just pick one.)

QUESTION ASM-3A. `movzbl %al, %eax`

Show solution

`unsigned char`

Hide solution

Hide solution

QUESTION ASM-3B. `movl -28(%rbp), %edx`

Show solution

> `int` or `unsigned`
>
> Hide solution
>
> Hide solution

QUESTION ASM-3C. `movsbl -32(%rbp), %eax`

Show solution

> `[signed] char`
>
> Hide solution
>
> Hide solution

QUESTION ASM-3D. `movzwl -30(%rbp), %eax`

Show solution

> `unsigned short`
>
> Hide solution
>
> Hide solution

For each code sample below, indicate the most likely data structure being accessed (assume that `g_var` is a global variable). Be as specific as possible.

QUESTION ASM-3E. `movzwl 6(%rdx,%rax,8), %eax`

Show solution

> `unsigned short` in an array of 8-byte structures
>
> Hide solution
>
> Hide solution

QUESTION ASM-3F. `movl (%rdx,%rax,4), %eax`

Show solution

> Array of `int`s or `unsigned int`s
>
> Hide solution
>
> Hide solution

QUESTION ASM-3G.

```
movzbl 4(%rax), %eax
movsbl %al, %eax
```

Show solution

> `char` field from a structure; *or* the 4th character in a string
>
> Hide solution
>
> Hide solution

For the remaining questions, indicate for what values of the register contents will the jump be taken.

QUESTION ASM-3H.

```
xorl %eax, %eax
jge LABEL
```

Show solution

> Always
>
> Hide solution
>
> Hide solution

QUESTION ASM-3I.

```
testb $1, %eax
jne LABEL
```

Show solution

Any odd value (the fact that we're only looking at the lowest byte is pretty irrelevant)

Hide solution

Hide solution

QUESTION ASM-3J.

```
cmpl %edx, %eax
jl LABEL
```

Show solution

When %eax is less than %edx, considered as signed integers

Hide solution

Hide solution

ASM-4. Assembly language

The next four questions pertain to the following four code samples.

f1

```
f1:
        subq    $8, %rsp
        call    callfunc
        movl    %eax, %edx
        leal    1(%rax,%rax,2), %eax
        testb   $1, %dl
        jne     .L3
        movl    %edx, %eax
        shrl    $31, %eax
        addl    %edx, %eax
        sarl    %eax
.L3:
        addq    $8, %rsp
        ret
```

f2

```
f2:
        pushq   %rbx
        xorl    %ebx, %ebx
.L3:
        movl    %ebx, %edi
        addl    $1, %ebx
        call    callfunc
        cmpl    $10, %ebx
        jne     .L3
        popq    %rbx
        ret
```

f3

```
f3:
        subq    $8, %rsp
        call    callfunc
        subl    $97, %eax
        cmpb    $4, %al
        ja      .L2
        movzbl  %al, %eax
        jmp     *.L4(,%rax,8)
.L4:
        .quad   .L3
        .quad   .L9
        .quad   .L6
        .quad   .L7
        .quad   .L8
.L3:
        movl    $42, %edx
        jmp     .L5
.L6:
        movl    $4096, %edx
        jmp     .L5
.L7:
        movl    $52, %edx
        jmp     .L5
.L8:
        movl    $6440, %edx
        jmp     .L5
.L2:
        movl    $0, %edx
        jmp     .L5
.L9:
        movl    $61, %edx
.L5:
        movl    $.LC0, %esi
        movl    $1, %edi
        movl    $0, %eax
        call    __printf_chk
        addq    $8, %rsp
        ret
.LC0:
        .string "Sum = %d\n"
```

f4

```
f4:
        subq    $40, %rsp
        movl    $1, (%rsp)
        movl    $0, 16(%rsp)
.L2:
        leaq    16(%rsp), %rsi
        movq    %rsp, %rdi
        call    callfunc
        movl    16(%rsp), %eax
        cmpl    %eax, (%rsp)
        jne     .L2
        addq    $40, %rsp
        ret
```

Now answer the following questions. Pick the most likely sample; you will use each sample exactly once.

QUESTION ASM-4A. Which sample contains a for loop?

Show solution

`f2`

Hide solution

Hide solution

QUESTION ASM-4B. Which sample contains a switch statement?

Show solution

`f3`

Hide solution

Hide solution

QUESTION ASM-4C. Which sample contains only an if/else construct?

Show solution

`f1`

Hide solution

Hide solution

QUESTION ASM-4D. Which sample contains a while loop?

Show solution

`f4`

Hide solution

Hide solution

ASM-5. Calling conventions: 6186

University Professor Helen Vendler is designing a poetic new processor, the 6186. Can you reverse-engineer some aspects of the 6186's calling convention from its assembly?

Here's a function:

```
int f(int* a, unsigned b) {
    extern int g(int x);
    int index = g(a[2*b + 1]);
    return a[index + b];
}
```

And here's that function compiled into 6186 instructions.

```
f:
    sub $24, %rsp
    movq %ra, (%rsp)
    mov %rb, %rx
    shl $1, %rx
    add $1, %rx
    movl (%ra, %rx, 4), %ra
    call g
    add %rb, %rr
    movq (%rsp), %ra
    movl (%ra, %rr, 4), %ra
    mov %ra, %rr
    add $24, %rsp
    ret
```

6186 assembly syntax is based on x86-64 assembly, and like the x86-64, 6186 registers are 64 bits wide. However, the 6186 has a different set of registers. There are just five general-purpose registers, `%ra`, `%rb`, `%rr`, `%rx`, and `%ry`. (“[W]hen she tries to be deadly serious she is speaking under…constraint”.) The example also features the stack pointer register, `%rsp`.

Give *brief* explanations if unsure.

QUESTION ASM-5A. Which register holds function return values?

Show solution

> `%rr`
>
> Hide solution
>
> Hide solution

QUESTION ASM-5B. What is `sizeof(int)` on the 6186?

Show solution

> 4
>
> Hide solution
>
> Hide solution

QUESTION ASM-5C. Which general-purpose register(s) must be callee-saved?

Show solution

> `%rb`
>
> Hide solution
>
> Hide solution

QUESTION ASM-5D. Which general-purpose register(s) must be caller-saved?

Show solution

`%rr`, `%ra`, `%rx`

Hide solution

Hide solution

QUESTION ASM-5E. Which general-purpose register(s) might be callee-saved or caller-saved (you can't tell which)?

Show solution

`%ry`

Hide solution

Hide solution

QUESTION ASM-5F. Assuming the compiler makes function stack frames as small as possible given the calling convention, what is the alignment of stack frames?

Show solution

32

Hide solution

Hide solution

QUESTION ASM-5G. Assuming that the 6186 supports the same addressing modes as the x86-64, write a *single instruction* that has the same effect on `%ra` as these three instructions:

```
shl $1, %rx
add $1, %rx
movl (%ra,%rx,4), %ra
```

Show solution

```
movl 4(%ra,%rx,8), %ra
```

Hide solution

Hide solution

ASM-6. Data structure assembly

Here are four assembly functions, `f1` through `f4`.

```
f1:
        pushq   %rbp
        movq    %rsp, %rbp
        testl   %esi, %esi
        jle     LBB0_3
        incl    %esi
LBB0_2:
        movq    8(%rdi), %rdi
        decl    %esi
        cmpl    $1, %esi
        jg      LBB0_2
LBB0_3:
        movl    (%rdi), %eax
        popq    %rbp
        retq
```

```
f2:
        pushq   %rbp
        movq    %rsp, %rbp
        movslq  %esi, %rax
        movq    (%rdi,%rax,8), %rcx
        movl    (%rcx,%rax,4), %eax
        popq    %rbp
        retq
```

```
f3:
        testl   %esi, %esi
        jle     LBB2_3
        incl    %esi
LBB2_2:
        movl    %edx, %eax
        andl    $1, %eax
        movq    8(%rdi,%rax,8), %rdi
        sarl    %edx
        decl    %esi
        cmpl    $1, %esi
        jg      LBB2_2
LBB2_3:
        movl    (%rdi), %eax
        retq
```

```
f4:
        movslq  %esi, %rax
        movl    (%rdi,%rax,4), %eax
        retq
```

QUESTION ASM-6A. Each function returns a value loaded from some data structure. Which function uses which data structure?

1. Array
2. Array of pointers to arrays
3. Linked list
4. Binary tree

Show solution

> Array—`f4`; Array of pointers to arrays—`f2`; Linked list—`f1`; Binary tree—`f3`
>
> Hide solution
>
> Hide solution

QUESTION ASM-6B. The array data structure is an array of type T. Considering the code for the function that manipulates the array, which of the following types are likely possibilities for T? Circle all that apply.

1. `char`
2. `int`
3. `unsigned long`
4. `unsigned long long`
5. `char*`
6. None of the above

Show solution

> `int`
>
> Hide solution
>
> Hide solution

ASM-7. Where's Waldo?

In the following questions, we give you C code and a portion of the assembly generated by some compiler for that code. (Sometimes we blank out a part of the assembly.) The C code contains a variable, constant, or function called `waldo`, and a point in the assembly is marked with asterisks `***`. Your job is to find Waldo: write an **assembly expression or constant** that holds the value of `waldo` at the marked point. We've done the first one for you.

NON-QUESTION: Where's Waldo?

```
int identity(int waldo) {
    return waldo;
}
```

```
00000000004007f6 <identity>:
  4007f6:       55                      push   %rbp
  4007f7:       48 89 e5                mov    %rsp,%rbp
  4007fa:       89 7d fc                mov    %edi,-0x4(%rbp)
  4007fd:       8b 45 fc                mov    -0x4(%rbp),%eax
           ***
  400800:       5d                      pop    %rbp
  400801:       c3                      retq
```

ANSWER: `%edi`, `-0x4(%rbp)`, `%eax`, and `%rax` all hold the value of `waldo` at the marked point, so any of them is a valid answer. If the asterisks came before the *first* instruction, only `%edi` would work.

QUESTION ASM-7A: Where's Waldo?

```
int f1(int a, int b, int waldo, int d) {
    if (a > b) {
        return waldo;
    } else {
        return d;
    }
}
```

```
0000000000400802 <f1>:
          ***
  400802:       55                      push   %rbp
  400803:       48 89 e5                mov    %rsp,%rbp
  400806:       89 7d fc                mov    %edi,-0x4(%rbp)
  400809:       89 75 f8                mov    %esi,-0x8(%rbp)
  40080c:       89 55 f4                mov    %edx,-0xc(%rbp)
  40080f:       89 4d f0                mov    %ecx,-0x10(%rbp)
  400812:       8b 45 fc                mov    -0x4(%rbp),%eax
  400815:       3b 45 f8                cmp    -0x8(%rbp),%eax
  400818:       7e 05                   jle    40081f <f1+0x1d>
  40081a:       8b 45 f4                mov    -0xc(%rbp),%eax
  40081d:       eb 03                   jmp    400822 <f1+0x20>
  40081f:       8b 45 f0                mov    -0x10(%rbp),%eax
  400822:       5d                      pop    %rbp
  400823:       c3                      retq
```

Show solution

`%edx`

Hide solution

Hide solution

QUESTION ASM-7B: Where's Waldo?

```
int int_array_get(int* a, int waldo) {
    int x = a[waldo];
    return x;
}
```

```
00000000004007d9 <int_array_get>:
INSTRUCTIONS OMITTED
          ***
 4007dc:       8b 04 b7                mov    (%rdi,%rsi,4),%eax
 4007df:       c3                      retq
```

Show solution

`%rsi`

Hide solution

Hide solution

QUESTION ASM-7C: Where's Waldo?

```
int matrix_get(int** matrix, int row, int col) {
    int* waldo = matrix[row];
    return waldo[col];
}
```

```
00000000004007e0 <matrix_get>:
 4007e0:       48 63 f6                movslq %esi,%rsi
 4007e3:       48 63 d2                movslq %edx,%rdx
          ***
 4007e6:       ?? ?? ?? ??             mov    ??,%rax
 4007ea:       8b 04 90                mov    (%rax,%rdx,4),%eax
 4007ed:       c3                      retq
```

Show solution

```
(%rdi,%rsi,8)
```

Hide solution

Hide solution

QUESTION ASM-7D: Where's Waldo?

```
int f5(int x) {
    extern int waldo(int);
    return waldo(x * 45);
}
```

```
0000000000400be0 <f5>:
          ***
 400be0:       6b ff 2d                imul   $0x2d,%edi,%edi
 400be3:       eb eb                   jmp    400bd0
```

Show solution

```
0x400bd0
```

Hide solution

Hide solution

QUESTION ASM-7E: Where's Waldo?

```c
int factorial(int waldo) {
    if (waldo < 2) {
        return 1;
    } else {
        return waldo * factorial(waldo - 1);
    }
}
```

```
0000000000400910 <factorial>:
     400910:       83 ff 01                  cmp    $0x1,%edi
     400913:       b8 01 00 00 00            mov    $0x1,%eax
     400918:       7e 13                     jle    .L2 <factorial+0x1b>
     40091a:       [6 bytes of padding (a no-op instruction)]
.L1:         ***
     400920:       0f af c7                  imul   %edi,%eax
     400923:       83 ef 01                  sub    $0x1,%edi
     400926:       83 ff 01                  cmp    $0x1,%edi
     400929:       75 f5                     jne    .L1 <factorial+0x10>
.L2: 40092b:       f3 c3                     repz retq
```

Show solution

`%edi`

Hide solution

Hide solution

QUESTION ASM-7F: Where's Waldo?

⚠️ This question currently uses 32-bit assembly.

```c
int binary_search(const char* needle, const char** haystack, unsigned sz) {
    unsigned waldo = 0, r = sz;
    while (waldo < r) {
        unsigned m = waldo + ((r - waldo) >> 1);
        if (strcmp(needle, haystack[m]) < 0) {
            r = m;
        } else if (strcmp(needle, haystack[m]) == 0) {
            waldo = r = m;
        } else {
            waldo = m + 1;
        }
    }
    return waldo;
}
```

```
80484ab <binary_search>:
     INSTRUCTIONS OMITTED
.L1: 80484c3:       89 fe                     mov    %edi,%esi
     80484c5:       29 de                     sub    %ebx,%esi
     80484c7:       d1 ee                     shr    %esi
     80484c9:       01 de                     add    %ebx,%esi
     80484cb:       8b 44 b5 00               mov    0x0(%ebp,%esi,4),%eax
     80484cf:       89 44 24 04               mov    %eax,0x4(%esp)
     80484d3:       8b 44 24 30               mov    0x30(%esp),%eax
     80484d7:       89 04 24                  mov    %eax,(%esp)
     80484da:       e8 11 fe ff ff            call   80482f0 <strcmp@plt>
     80484df:       85 c0                     test   %eax,%eax
     80484e1:       78 09                     js     .L2 <binary_search+0x41>
     80484e3:       85 c0                     test   %eax,%eax
     80484e5:       74 13                     je     80484fa <binary_search+0x4f>
               ***
     80484e7:       8d 5e 01                  lea    0x1(%esi),%ebx
     80484ea:       eb 02                     jmp    .L3 <binary_search+0x43>
.L2: 80484ec:       89 f7                     mov    %esi,%edi
.L3: 80484ee:       39 df                     cmp    %ebx,%edi
     80484f0:       77 d1                     ja     .L1 <binary_search+0x18>
     INSTRUCTIONS OMITTED
```

Show solution

`%ebx`

Hide solution

Hide solution

In the remaining questions, you are given assembly compiled from one of the above functions by a different compiler, or at a different optimization level. Your goal is to figure out what C code corresponds to the given assembly.

QUESTION ASM-7G:

⚠️ This question currently uses 32-bit assembly.

```
804851d <waldo>:
804851d:       55                      push   %ebp
804851e:       89 e5                   mov    %esp,%ebp
8048520:       83 ec 18                sub    $0x18,%esp
8048523:       83 7d 08 01             cmpl   $0x1,0x8(%ebp)
8048527:       7f 07                   jg     8048530
8048529:       b8 01 00 00 00          mov    $0x1,%eax
804852e:       eb 10                   jmp    8048540
8048530:       8b 45 08                mov    0x8(%ebp),%eax
8048533:       48                      dec    %eax
8048534:       89 04 24                mov    %eax,(%esp)
8048537:       e8 e1 ff ff ff          call   804851d
804853c:       0f af 45 08             imul   0x8(%ebp),%eax
8048540:       c9                      leave
8048541:       c3                      ret
```

What's Waldo? Circle one.

1. `f1`
2. `f5`
3. `matrix_get`
4. `permutation_compare`
5. `factorial`
6. `binary_search`

Show solution

> 5, `factorial`
>
> Hide solution
>
> Hide solution

QUESTION ASM-7H:

> ⚠️ This question currently uses 32-bit assembly.

```
8048425 <waldo>:
8048425:       55                      push   %ebp
8048426:       89 e5                   mov    %esp,%ebp
8048428:       8b 45 08                mov    0x8(%ebp),%eax
804842b:       3b 45 0c                cmp    0xc(%ebp),%eax
804842e:       7e 05                   jle    8048435 <waldo+0x10>
8048430:       8b 45 10                mov    0x10(%ebp),%eax
8048433:       eb 03                   jmp    8048438 <waldo+0x13>
8048435:       8b 45 14                mov    0x14(%ebp),%eax
8048438:       5d                      pop    %ebp
8048439:       c3                      ret
```

What's Waldo? Circle one.

1. `f1`
2. `f5`
3. `matrix_get`
4. `permutation_compare`
5. `factorial`
6. `binary_search`

Show solution

> 1, `f1`
>
> Hide solution
>
> Hide solution

QUESTION ASM-7I:

```
00000000004008b4 <waldo>:
 4008b4:       55                      push   %rbp
 4008b5:       48 89 e5                mov    %rsp,%rbp
 4008b8:       48 83 ec 10             sub    $0x10,%rsp
 4008bc:       89 7d fc                mov    %edi,-0x4(%rbp)
 4008bf:       8b 45 fc                mov    -0x4(%rbp),%eax
 4008c2:       6b c0 2d                imul   $0x2d,%eax,%eax
 4008c5:       89 c7                   mov    %eax,%edi
 4008c7:       e8 9e 05 00 00          callq  400e6a
 4008cc:       c9                      leaveq
 4008cd:       c3                      retq
```

What's Waldo? Circle one.

1. `f1`

2. `f5`
3. `matrix_get`
4. `permutation_compare`
5. `factorial`
6. `binary_search`

Show solution

2, `f5`

Hide solution

Hide solution

ASM-8. (removed because redundant)

ASM-9. Disassembly II

The questions in this section concern a function called `ensmallen`, which has the following assembly.

```
ensmallen:
 1.          movzbl  (%rsi),  %edx
 2.          testb   %dl, %dl
 3.          movb    %dl, (%rdi)
 4.          jne     .L22
 5.          jmp     .L23
 6.   .L18:
 7.          addq    $1, %rsi
 8.   .L22:
 9.          movzbl  (%rsi), %eax
10.          cmpb    %dl, %al
11.          je      .L18
12.          addq    $1, %rdi
13.          testb   %al, %al
14.          movb    %al, (%rdi)
15.          je      .L23
16.          movl    %eax, %edx
17.          jmp     .L22
18.   .L23:
19.          retq
```

QUESTION ASM-9A. How many arguments is this function likely to take? Give line numbers that helped you determine an answer.

Show solution

> 2, because of lines 1 & 3
>
> Hide solution
>
> Hide solution

QUESTION ASM-9B. Are the argument(s) pointers? Give line numbers that helped you determine an answer.

Show solution

> Yes, because of lines 1, 3, 9, 14
>
> Hide solution
>
> Hide solution

QUESTION ASM-9C. What type(s) are the argument(s) likely to have? Give line numbers that helped you determine an answer.

Show solution

> `unsigned char*`. Lines 1, 3, 9, and 14 are *byte*-moving instructions. The `z` in `movzbl` (Lines 1 and 9) indicates zero-extension, i.e., `unsigned char`. But `char*` is possible too; the characters are only compared for equality with each other (Line 10) or zero (Lines 2/4 and 13/15), so we can't really distinguish signed from unsigned.
>
> Hide solution
>
> Hide solution

QUESTION ASM-9D. Write a likely signature for the function. Use return type `void`.

Show solution

```
void ensmallen(unsigned char* a, unsigned char* b);
```

> Hide solution
>
> Hide solution

QUESTION ASM-9E. Write an alternate likely signature for the function, different from your last answer. Again, use return type `void`.

Show solution

```
void ensmallen(unsigned char* a, const unsigned char* b);
void ensmallen(char* a, char* b);
void ensmallen(void* dst, const void* src);
```

etc., etc.

Hide solution

Hide solution

QUESTION ASM-9F. Which callee-saved registers does this function use? Give line numbers that helped you determine an answer.

Show solution

None except possibly %rsp (no callee-saved registers are referenced in the code).

Hide solution

Hide solution

QUESTION ASM-9G. The function has an “input” and an “output”. Give an “input” that would cause the CPU to jump from line 5 to label `.L23`, and describe what is placed in the “output” for that “input”.

Show solution

The input is an empty string (`""`), and the function puts an empty string in the output.

You might think the function’s output was the value of its %eax register what it returned. But remember that functions without return values can also use %eax, and we told you above that this function’s return type is `void`! `ensmallen`’s “output” is most likely the string pointed to by its first parameter. In that sense `ensmallen` is sort of like `strcpy` or `memcpy`.

Hide solution

Hide solution

QUESTION ASM-9H. Give an “input” for which the corresponding “output” is **not** a copy of the “input”. Your answer must differ from the previous answer.

Show solution

"aaaa" (output is "a"); any string that has adjacent characters that are the same

Hide solution

Hide solution

QUESTION ASM-9I. Write C code corresponding to this function. Make it as compact as you can.

Show solution

```
void ensmallen(char* dst, const char* src) {
    while ((*dst = *src)) {
        while (*dst == *src)
            ++src;
        ++dst;
    }
}
```

Or, a little less compactly:

```
void ensmallen(char* dst, const char* src) {
    while (*src) {
        *dst = *src;
        while (*src == *dst)
            ++src;
        ++dst;
    }
    *dst = 0;
}
```

Hide solution

Hide solution

ASM-10. Machine programming

Intel really messed up this time. They've released a processor, the Fartium Core Trio, where every instruction is broken *except* the ones on this list.

1. `cmpq %rdi, %rsi`
2. `decq %rsi`
3. `incq %rax`
4. `je L1`
5. `jl L2`

6. `jmp L3`
7. `movl (%rdi,%rax,4), %edi`
8. `retq`
9. `xchgq %rax, %rcx`
10. `xorq %rax, %rax`

(In case you forgot, `xchgq` swaps two values—here, the values in two registers—without modifying condition codes.)

"So what if it's buggy," says Andy Grove; "it can still run programs." For instance, he argues convincingly that this function:

```
void do_nothing() {
}
```

is implemented correctly by this Fartium instruction sequence:

```
retq
```

Your job is to implement more complex functions using **only** Fartium instructions. Your implementations must have the same semantics as the C functions, but may perform much worse than one might expect. You may leave off arguments and write instruction numbers (#1–10) or instruction names. Indicate where labels `L1`–`L3` point (if you need them). Assume that the Fartium Core Trio uses the normal x86-64 calling convention.

QUESTION ASM-10A.

```
int return_zero() {
    return 0;
}
```

Show solution

`xorq %rax, %rax; retq`.

`%rax` has unknown value when a function begins, so we need to clear it.

Hide solution

Hide solution

QUESTION ASM-10B.

```
int identity(int a) {
    return a;
}
```

Show solution

`xchgq %rdi, %rax; retq`.

Hide solution

Hide solution

QUESTION ASM-10C.

```
void infinite_loop() {
    while (1) {
        /* do nothing */
    }
}
```

Show solution

`L3: jmp L3`.

Hide solution

Hide solution

QUESTION ASM-10D.

```
struct point {
    int x;
    int y;
    int z;
};

int extract_z(point* p) {
    return p->z;
}
```

Show solution

```
xorq %rax, %rax
incq %rax
incq %rax
movl (%rdi,%rax,4), %edi
xchgl %rax, %rdi
ret
```

Hide solution

Hide solution

So much for the easy ones. Now complete *one* out of the following parts, or more than one for extra credit.

QUESTION ASM-10E.

```
long add(long a, long b) {
    return a + b;
}
```

Show solution

```
    xorq %rax, %rax        # %rax := 0
    xchgq %rax, %rdi       # now %rax == a and %rdi == 0
L3: cmpq %rdi, %rsi        # compare %rsi and %rdi (which is 0)
    je L1                  # "if %rsi == 0 goto L1"
    incl %rax              # ++%rax
    decl %rsi              # --%rsi
    jmp L3
L1: retq
```

The loop at L3 executes b times, incrementing %eax each time. Here's morally equivalent C++:

```
long add(long a, long b) {
    while (b != 0) {
        ++a;
        --b;
    }
    return a;
}
```

Hide solution

Hide solution

QUESTION ASM-10F.

```
int array_dereference(int* a, long i) {
    return a[i];
}
```

Show solution

```
    xorq %rax, %rax          # %rax := 0
L3: xchgq %rax, %rdi
    cmpl %rdi, %rsi
    xchgq %rax, %rdi
    je L1                    # "if %rax == i goto L1"
    incq %rax                # ++%rax
    jmp L3
L1: movl (%rdi,%rax,4), %edi  # %edi := a[i]
    xchgq %rax, %rdi
    ret
```

Hide solution

Hide solution

ASM-11. Program Layout

For the following questions, select the part(s) of memory from the list below that best describes where you will find the object.

1. heap
2. stack
3. between the heap and the stack
4. in a read-only data segment
5. in a text segment starting at address 0x08048000
6. in a read/write data segment
7. in a register

Assume the following code, compiled without optimization.

```
#include <errno.h>
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

// The following is copied from stdio.h for your reference
#define EOF (-1)
```

```
1.    unsigned long fib(unsigned long n) {
2.        if (n < 2) {
3.            return n;
4.        }
5.        return fib(n - 1) + fib(n - 2);
6.    }
7.
8.    int main(int argc, char *argv[]) {
9.        extern int optind;
10.       char ch;
11.       unsigned long f, n;
12.
13.       // Command line processing
14.       while ((ch = getopt(argc, argv, "h")) != EOF) {
15.           switch (ch) {
16.           case 'h':
17.           case '?':
18.           default:
19.               return (usage());
20.           }
21.       }
22.
23.       argc -= optind;
24.       argv += optind;
25.
26.       if (argc != 1) {
27.           return usage();
28.       }
29.
30.       n = strtoul(strdup(argv[0]), nullptr, 10);
31.       if (n == 0 && errno == EINVAL) {
32.           return usage();
33.       }
34.
35.       /* Now call one of the fib routines. */
36.       f = fib(n);
37.       printf("fib(%lu) = %lu\n", n, f);
38.
39.       return 0;
40.   }
```

QUESTION ASM-11A. The string `"fib(%lu) = %lu\n"` (line 37).

Show solution

Read-only data segment, aka text segment

Hide solution

Hide solution

QUESTION ASM-11B. `optind` (line 23).

Show solution

Read/write data segment

Hide solution

Hide solution

QUESTION ASM-11C. When executing at line 5, where you will find the address to which `fib` returns.

Show solution

Stack

Hide solution

Hide solution

QUESTION ASM-11D. Where will you find the value of EOF that is compared to the return value of `getopt` in line 14.

Show solution

Register—although this register is likely to be hidden inside the processor, not one of the ones that have programmable names. Alternately, text segment, since the −1 will be encoded into some instruction.

Hide solution

Hide solution

QUESTION ASM-11E. `getopt` (line 14)

Show solution

Text segment; alternately: Between the heap and the stack (because that’s where shared libraries tend to be loaded)

Hide solution

Hide solution

QUESTION ASM-11F. `fib` (lines 1-6)

Show solution

Text segment

Hide solution

Hide solution

QUESTION ASM-11G. the variable `f` (line 36)

Show solution

Register or stack

Hide solution

Hide solution

QUESTION ASM-11H. the string being passed to `strtoul` (line 30)

Show solution

Heap

Hide solution

Hide solution

QUESTION ASM-11I. `strdup` (line 30)

Show solution

Text segment or between heap & stack (same as `getopt`)

Hide solution

Hide solution

QUESTION ASM-11J. The value of the `fib` function when we return from `fib` (line 5).

Show solution

Register (%rax)

Hide solution

Hide solution

ASM-12. Assembly and Data Structures

Consider the following assembly function.

```
func:
        xorl    %eax, %eax
        cmpb    $0, (%rdi)
        je      .L27
.L26:
        addq    $1, %rdi
        addl    $1, %eax
        cmpb    $0, (%rdi)
        jne     .L26
.L27:
        retq
```

QUESTION ASM-12A. How many parameters does this function appear to have?

Show solution

1

Hide solution

Hide solution

QUESTION ASM-12B. What do you suppose the type of that parameter is?

Show solution

`const char*` (or `const unsigned char*`, `char*`, etc.)

Hide solution

Hide solution

QUESTION ASM-12C. Write C++ code that corresponds to it.

Show solution

It's `strlen`!

```
int strlen(const char* x) {
    int n = 0;
    for (; *x; ++x)
        ++n;
    return n;
}
```

Hide solution

Hide solution

ASM-13. Assembly language

Consider the following four assembly functions.

```
# Code Sample 1
       movq    %rdi, %rax
       testq   %rdx, %rdx
       je      .L2
       addq    %rdi, %rdx
       movq    %rdi, %rcx
.L3:
       addq    $1, %rcx
       movb    %sil, -1(%rcx)
       cmpq    %rdx, %rcx
       jne     .L3
.L2:
       rep ret
```

```
# Code Sample 2
        movq    %rdi, %rax
        testq   %rdx, %rdx
        je      .L2
        addq    %rdi, %rdx
        movq    %rdi, %rcx
.L3:
        addq    $1, %rcx
        addq    $1, %rsi
        movzbl  -1(%rsi), %r8d
        movb    %r8b, -1(%rcx)
        cmpq    %rdx, %rcx
        jne     .L3
.L2:
        rep ret
```

```
# Code Sample 3
       movb    (%rsi), %al
       testb   %al, %al
       je      .L3
       incq    %rsi
.L2:
       movb    %al, (%rdi)
       incq    %rdi
       movb    (%rsi), %al
       incq    %rsi
       testb   %al, %al
       jne     .L2
.L3:
       movq    %rdi, %rax
       ret
```

```
# Code Sample 4
        testq   %rdx, %rdx
        je      .L3
        movq    %rdi, %rax
.L2:
        movb    %sil, (%rax)
        incq    %rax
        decq    %rdx
        jne     .L2
.L3:
        movq    %rdi, %rax
        ret
```

(*Note:* The `%sil` register is the lowest-order byte of register `%rsi`, just as `%al` is the lowest-order byte of `%rax` and `%r8b` is the lowest-order byte of `%r8`.)

QUESTION ASM-13A. Which two of the assembly functions perform the exact same task?

Show solution

1 and 4

Hide solution

Hide solution

QUESTION ASM-13B. What is that task? You can describe it briefly, or give the name of the corresponding C library function.

Show solution

`memset`

Hide solution

Hide solution

QUESTION ASM-13C. Explain how the other two functions differ from each other.

Show solution

One is `memcpy` and the other is `strcpy`, so the difference is that #2 terminates after copying a number of bytes indicated by the parameter while the other terminates when it encounters a NUL value in the source string.

Hide solution

Hide solution

ASM-14. Golden Baron

A very rich person has designed a new x86-64-based computer, the Golden Baron Supercomputer 9000, and is paying you handsomely to write a C compiler for it. There's just one problem. This person, like many very rich people, is dumb, and on their computer, *odd-numbered memory addresses don't work for data*. When data is loaded into a general-purpose register from an odd-numbered address, the value read is zero. For example, consider the following instructions:

```
movl $0x01020304, a(%rip)
movl a(%rip), %eax
```

(where the address of `a` is even). Executed on true x86-64, `%rax` will hold the value 0x01020304; on Golden Baron, `%rax` will hold 0x00020004.

But it is still possible to write a correct C compiler for this ungodly hardware—you just have to work around the bad memory with code. You plan to use two bytes of Golden Baron memory for every one byte of normal x86-64 memory. For instance, an array `int a[2] = {1, 0x0a0b0c0d};` would be stored in 16 bytes of memory, like so:

```
01 00 00 00 00 00 00 00 0d 00 0c 00 0b 00 0a 00
```

Pointer arithmetic, and moving multi-byte values to and from registers, must account for the zero bytes that alternate with meaningful bytes. So to read the correct value for `a[2]`, the compiler must arrange to read the bytes at addresses `A+8`, `A+10`, `A+12`, and `A+14`, where `A` is the address of the first byte in `a`.

QUESTION ASM-14A. What should `printf("%zu\n", sizeof(char))` print on Golden Baron?

Show solution

1. This is required by the C++ abstract machine: `sizeof(char) == 1`.

Hide solution

Hide solution

QUESTION ASM-14B. This function

```
int f(signed char* c, size_t i) {
    return c[i];
}
```

can compile to two instructions on x86-64, including `retq`. It can also compile to two instructions on Golden Baron. (We're assuming that memory used for Golden Baron instructions works normally.) What are those instructions?

Show solution

```
movsbl (%rdi,%rsi,2), %eax
retq
```

Hide solution

Hide solution

QUESTION ASM-14C. This function

```
int g(int* a, size_t i) {
    return a[i];
}
```

can compile to two instructions on x86-64, but Golden Baron requires more work. Write the Golden Baron translation of this function in x86-64 assembly language. For partial credit, write C code that, executed on x86-64, would return the correct value from a Golden Baron-formatted array.

Show solution

```
movzbl (%rdi,%rsi,8), %eax
movzbl 2(%rdi,%rsi,8), %ecx
shll $8, %ecx
orl %ecx, %eax
movzbl 4(%rdi,%rsi,8), %ecx
shlq $16, %ecx
orl %ecx, %eax
movzbl 8(%rdi,%rsi,8), %ecx
shlq $24, %ecx
orl %ecx, %eax
retq
```

Or:

```
movq (%rdi,%rsi,8), %rcx
movq %rcx, %rax
andq $255, %rax

shrq $8, %rcx
movq %rcx, %rdx
andq $0xff00, %rdx
orl %edx, %eax

shrq $16, %rcx
movq %rcx, %rdx
andq $0xff0000, %rdx
orl %edx, %eax

shrq $32, %rcx
movq %rcx, %rdx
andq $0xff000000, %rdx
orl %edx, %eax

retq
```

Hide solution

Hide solution

QUESTION ASM-14D. The Golden Baron's x86-64 processor actually supports a secret instruction, `swizzleq SRC, REG`, which rearranges the nybbles (the hexadecimal digits—the aligned 4-bit slices) of the destination register `REG` based on the source argument `SRC`. Here's some examples. Assuming that `%rax` holds the value `0x0123456789ABCDEF`, the following `swizzleq` instructions leave the indicated results in `%rax`:

- `swizzleq $0, %rax`: %rax gets `0xFFFF'FFFF'FFFF'FFFF`.

The contents of nybble 0 [bits 0-3, inclusive], are repeated into every nybble.

- `swizzleq $0xFEDCBA9876543210, %rax`: %rax gets`0x0123'4567'89AB'CDEF`.

  Each nybble is mapped to its current value: nybble 0 [bits 0-3] is placed in nybble 0 [bits 0-3], nybble 1 in nybble 1, and so forth.

- `swizzleq $0x0123456701234567, %rax`: %rax gets `0xFEDC'BA98'FEDC'BA98`.

  Nybble 0 [bits 0-3] is placed in nybbles 7 and 15 [bits 28-31 and 60-63]; nybble 1 [bits 4-7] is placed in nybbles 6 and 14 [bits 24-27 and 56-59]; etc.

- `swizzleq $0xEFCDAB8967452301, %rax`: %rax gets `0x1032'5476'98BA'DCFE`.

  The nybbles of each byte are exchanged.

Use `swizzleq` to shorten your answer for Part C.

Show solution

```
movq (%rdi,%rsi,8), %rax
swizzleq $0x2222'2222'dc98'5410, %rax
retq
```

Hide solution

Hide solution

ASM-15. Instruction behavior

QUESTION ASM-15A. Name three different x86-64 instructions that *always* modify the stack pointer, no matter their arguments (instruction names only; suffixes don't count, so `movl` and `movq` are the same instruction name).

Show solution

`push`, `pop`, `call`, `ret`

Hide solution

Hide solution

QUESTION ASM-15B. Name three different x86-64 instructions that *sometimes* modify the stack pointer, depending on their arguments.

Show solution

`mov`, `add`, `sub`, `or`, `and`, …

Hide solution

Hide solution

QUESTION ASM-15C. Name three different x86-64 instructions that *never* modify the stack pointer, no matter their arguments.

Show solution

`jmp`, `jne`, `je`, `jWHATEVER`, `cmp`, `test`, `nop`, many others

Hide solution

Hide solution

QUESTION ASM-15D. List three different instructions, *including arguments,* that if placed immediately before a `retq` instruction that ends a function, will *never* change the function's behavior. The instructions should have different names. No funny business: assume the function was compiled from valid C, that relative jumps are fixed up, and that, for example, it doesn't access its own code.

Show solution

Many examples:

- `retq` :)
- `jmp [next instruction]`
- `test (any register), (any register)`
- `cmp (any register), (any register)`
- `nop`
- `mov`s or arithmetic instructions that involve caller-saved registers other than `%rax`

Hide solution

Hide solution

ASM-16. Calling convention

The following questions concern valid C++ functions compiled using the normal x86-64 calling convention. True or false?

QUESTION ASM-16A. If the function's instructions do not save and restore any registers, then the C++ function did not call any other function.

Show solution

> False for two reasons: (1) If this function doesn’t use any callee-saved registers, it doesn't need to explicitly save & restore anything. (2) Tail call elimination.
>
> Hide solution
>
> Hide solution

QUESTION ASM-16B. If the function’s instructions do not change the stack pointer, then the function’s instructions do not contain a `call` instruction.

Show solution

> True because of stack alignment.
>
> Hide solution
>
> Hide solution

QUESTION ASM-16C. If the function’s instructions do not change the stack pointer, then the C++ function did not call any other function. **Explain your answer briefly.**

Show solution

> False because of tail call elimination.
>
> Hide solution
>
> Hide solution

QUESTION ASM-16D. If the function’s instructions do not modify the `%rax` register, then the C++ function must return `void`.

Show solution

> False; the function could return the result of calling another function.
>
> Hide solution
>
> Hide solution

QUESTION ASM-16E. If the function’s instructions store a local variable on the stack, then that variable’s address will be less than the function’s initial stack pointer.

Show solution

True

Hide solution

Hide solution

ASM-17. Assembly

Here are three x86-64 assembly functions that were compiled from C.

```
f1:
    xorl    %eax, %eax
L2:
    movsbq  (%rdi), %rdx
    subq    $48, %rdx
    cmpq    $9, %rdx
    ja      L5
    imulq   $10, %rax, %rax
    incq    %rdi
    addq    %rdx, %rax
    jmp     L2
L5:
    ret
```

```
f2:
    movq    %rdi, %rax
L7:
    cmpb    $0, (%rax)
    je      L9
    incq    %rax
    jmp     L7
L9:
    cmpq    %rax, %rdi
    jnb     L11
    decq    %rax
    movb    (%rdi), %cl
    incq    %rdi
    movb    (%rax), %dl
    movb    %cl, (%rax)
    movb    %dl, -1(%rdi)
    jmp     L9
L11:
    ret
```

```
f3:
    xorl    %eax, %eax
L13:
    cmpq    %rax, %rdx
    je      L15
    movb    (%rdi,%rax), %cl
    movb    (%rsi,%rax), %r8b
    movb    %r8b, (%rdi,%rax)
    movb    %cl, (%rsi,%rax)
    incq    %rax
    jmp     L13
L15:
    ret
```

(*Note:* `imulq $10, %rax, %rax` means `%rax *= 10`.)

QUESTION ASM-17A. How many arguments does each function most likely take?

Show solution

`f1`—1, `f2`—1, `f3`—3

Hide solution

Hide solution

QUESTION ASM-17B. Which functions modify at least one caller-saved register? List all that apply or write "none".

Show solution

All of them

Hide solution

Hide solution

QUESTION ASM-17C. Which functions **never** modify memory? List all that apply or write "none".

Show solution

`f1`

Hide solution

Hide solution

QUESTION ASM-17D. Write a signature for each function, giving a likely type for each argument and a likely return type. (You may give a `void` return type if you think the function doesn't return a useful value.)

```
________ f1(______________________________________)

________ f2(______________________________________)

________ f3(______________________________________)
```

Show solution

```
long f1(const char*);
void f2(char*);
void f3(char*, char*, size_t);
```

Hide solution

Hide solution

QUESTION ASM-17E. One of these functions swaps the contents of two memory regions. Which one?

Show solution

`f3`

Hide solution

Hide solution

QUESTION ASM-17F. What is the value of `%rax` in `f2` the first time `L9` is reached? Write a C expression in terms of `f2`'s argument or arguments; you may use standard library functions.

Show solution

`%rdi + strlen(%rdi)`

Hide solution

Hide solution

QUESTION ASM-17G. Give arguments for each function that would result in the function returning without writing to memory or causing a fault.

```
f1(________________________)

f2(________________________)

f3(________________________)
```

Show solution

`f1("")`, `f2("")`, `f3("", "", 0)`

Hide solution

Hide solution

QUESTION ASM-17H. Complete this function so that it returns the number

6161. For full credit, **use only calls to `f1`, `f2`, and `f3`**. For partial credit, do something simpler.

```c
int magic() {
    char s1[] = "Shaka kaSenzangakhona became King of the Zulu Kingdom in 1816";
    char s2[] = "Dingane kaSenzangakhona succeeded Shaka in 1828";
    char s3[] = "1661 in the Gregorian calendar is 3994 in the Korean calendar";





}
```

Show solution

```c
int magic() { …
    f2(s1);
    f3(s1, s3, 2);
    return f1(s3);
}
```

Hide solution

Hide solution

Storage and caching exercises

Many exercises that seem less appropriate this year, or which cover topics that we haven't covered in class, are marked with ⚠️. However, we may have missed some.

IO-1. I/O caching

Mary Ruefle, a poet who lives in Vermont, is working on her caching I/O library for CS 61. She wants to implement a cache with *N* slots. Since searching those slots might slow down her library, she writes a function that maps addresses to slots. Here's some of her code.

```c
#define SLOTSIZ 4096
struct io61_slot {
    char buf[SLOTSIZ];
    off_t pos; // = (off_t) -1 for empty slots
    ssize_t sz;
};

#define NSLOTS 64
struct io61_file {
    int fd;
    off_t pos; // current file position
    io61_slot slots[NSLOTS];
};

static inline int find_slot(off_t off) {
    return off % NSLOTS;
}

int io61_readc(io61_file* f) {
    int slotindex = find_slot(f->pos);
    io61_slot* s = &f->slots[slotindex];

    if (f->pos < s->pos || f->pos >= s->pos + s->sz) {
        // slot contains wrong data, need to refill it
        off_t new_pos = lseek(f->fd, f->pos, SEEK_SET);
        assert(new_pos != (off_t) -1); // only handle seekable files for now
        ssize_t r = read(f->fd, s->buf, SLOTSIZ);
        if (r == -1 || r == 0) {
            return EOF;
        }
        s->pos = f->pos;
        s->sz = r;
    }

    int ch = (unsigned char) s->buf[f->pos - s->pos];
    ++f->pos;
    return ch;
}
```

Before she can run and debug this code, Mary is led “to an emergency of feeling that … results in a poem.” She’ll return to CS61 and fix her implementation soon, but in the meantime, let’s answer some questions about it.

QUESTION IO-1A. True or false: Mary’s cache is a direct-mapped cache.

Show solution

True

Hide solution

Hide solution

QUESTION IO-1B. What changes to Mary's code could change your answer to Part A? Circle all that apply.

1. New code for `find_slot` (keeping `io61_readc` the same)
2. New code in `io61_readc` (keeping `find_slot` the same)
3. New code in `io61_readc` *and* new code for `find_slot`
4. None of the above

Show solution

#2 and #3

Hide solution

Hide solution

QUESTION IO-1C. Which problems would occur when Mary's code was used to sequentially read a seekable file of size 2MiB (2×2^{20} = 2097152 bytes) one character at a time? Circle all that apply.

1. Excessive CPU usage (>10x stdio)
2. Many system calls to read data (>10x stdio)
3. Incorrect data (byte *x* read at a position where the file has byte $y \neq x$)
4. Read too much data (more bytes read than file contains)
5. Read too little data (fewer bytes read than file contains)
6. Crash/undefined behavior
7. None of the above

Show solution

#2 only

Hide solution

Hide solution

QUESTION IO-1D. Which of these new implementations for `find_slot` would fix at least one of these problems with reading sequential files? Circle all that apply.

1. `return (off * 2654435761) % NSLOTS; /* integer hash function from Stack Overflow */`

2. `return (off / SLOTSIZ) % NSLOTS;`
3. `return off & (NSLOTS - 1);`
4. `return 0;`
5. `return (off >> 12) & 0x3F;`
6. None of the above

Show solution

#2, #4, #5

Hide solution

Hide solution

IO-2. Caches and reference strings

QUESTION IO-2A. True or false: A direct-mapped cache with *N* or more slots can handle any reference string containing ≤*N* distinct addresses with no misses except for cold misses.

Show solution

False: direct-mapped caches can have conflict misses

Hide solution

Hide solution

QUESTION IO-2B. True or false: A fully-associative cache with *N* or more slots can handle any reference string containing ≤*N* distinct addresses with no misses except for cold misses.

Show solution

True

Hide solution

Hide solution

Consider the following 5 reference strings.

Name	String
α	1
β	1, 2

γ	1, 2, 3, 4, 5
δ	2, 4
ε	5, 2, 4, 2

QUESTION IO-2C. Which of the strings might indicate a sequential access pattern? Circle all that apply.

α β γ δ ε None of these

Show solution

(α), β, γ

Hide solution

Hide solution

QUESTION IO-2D. Which of the strings might indicate a strided access pattern with stride >1? Circle all that apply.

α β γ δ ε None of these

Show solution

(α), δ

One very clever person pointed out that β and γ could also represent large strides: for example, consider a file with 10 bytes accessed with stride 11!

Hide solution

Hide solution

The remaining questions concern concatenated permutations of these five strings. For example, the permutation αβγδε refers to this reference string:

1, 1, 2, 1, 2, 3, 4, 5, 2, 4, 5, 2, 4, 2.

We pass such permutations through an initially-empty, fully-associative cache with 3 slots, and observe the numbers of hits.

QUESTION IO-2E. How many cold misses might a permutation observe? Circle all that apply.

0 1 2 3 4 5 Some other number

Show solution

> 5. The first time a reference string address is encountered, it must cause a cold miss.
>
> Hide solution
>
> Hide solution

Under LRU eviction, the permutation αβεγδ observes 5 hits as follows. (We annotate each access with "h" for hit or "m" for miss.)

1m; 1h, 2m; 5m, 2h, 4m, 2h; 1m, 2h, 3m, 4m, 5m; 2m, 4h.

QUESTION IO-2F. How many hits does this permutation observe under FIFO eviction?

Show solution

> 4 hits
>
> Hide solution
>
> Hide solution

QUESTION IO-2G. Give a permutation that will observe 8 hits under LRU eviction, which is the maximum for any permutation. There are several possible answers. (Write your answer as a permutation of αβγδε. For partial credit, find a permutation that has 7 hits, etc.)

Show solution

> The following four permutations observe 8 hits under LRU: αβγδε, αβγεδ, βαγδε, βαγεδ. 28 permutations observe 7 hits; 25 observe 6 hits; and 38 observe 5 hits.
>
> Hide solution
>
> Hide solution

QUESTION IO-2H. Give a permutation that will observe 2 hits under LRU eviction, which is the *minimum* for any permutation. There is one unique answer. (Write your answer as a permutation of αβγδε. For partial credit, find a permutation that has 3 hits, etc.)

Show solution

δαεγβ. 4 permutations observe 3 hits and 20 observe 4 hits.

Hide solution

Hide solution

IO-3. Processor cache

The git version control system is based on *commit hashes*, which are 160-bit (20-byte) hash values used to identify commits. In this problem you'll consider the processor cache behavior of several versions of a "grading server" that maps commits to grades. Here's the first version:

```
struct commit_info {
    char hash[20];
    int grade[11];
};

commit_info* commits;
size_t N;

int get_grade1(const char* hash, int pset) {
    for (size_t i = 0; i != N; ++i) {
        if (memcmp(commits[i].hash, hash, 20) == 0) {
            return commits[i].grade[pset];
        }
    }
    return -1;
}
```

We will ask questions about the average number of cache lines accessed by variants of `get_grade(hash, pset)`. You should make the following assumptions:

- The `hash` argument is uniformly drawn from the set of known commits. That is, the probability that `hash` equals the *i*th commit's hash is 1/`N`.
- Only count cache lines accessible via `commits`. Don't worry about cache lines used for local variables, for parameters, for global variables, or for instructions. For instance, do not count the `hash` argument or the global-data cache line that stores the `commits` variable itself.
- Every object is 64-byte aligned, and no two objects share the same cache line.
- Commit hashes are mathematically indistinguishable from random numbers. Thus, the probability that two different hashes have the same initial k bits equals $1/2^k$.
- Fully correct answers would involve ceiling functions, but you don't need to include them.

QUESTION IO-3A. What is the expected number of cache lines accessed by `get_grade1`, in terms of `N`?

Show solution

Each commit_info object is on its own cache line, and we will examine 1/2 of the objects on average, so the answer is $\lceil N/2 \rceil$.

Hide solution

Hide solution

The second version:

```
struct commit_info {
   char hash[20];
   int grade[11];
};

commit_info** commits;
size_t N;

int get_grade2(const char hash[20], int pset) {
    for (size_t i = 0; i != N; ++i) {
        if (memcmp(commits[i]->hash, hash, 20) == 0) {
            return commits[i]->grade[pset];
        }
    }
    return -1;
}
```

QUESTION IO-3B. What is the expected number of cache lines accessed by `get_grade2`, in terms of `N`?

Show solution

This still examines N/2 commit_info objects. But in addition it examines cache lines to evaluate the POINTERS to those objects. There are 8 such pointers per cache line ($8 \times 8 = 64$), and we examine N/2 pointers, for $N/8/2 = N/16$ additional cache lines. Thus $\lceil N/2 \rceil + \lceil N/16 \rceil \cong 9N/16$.

Hide solution

Hide solution

The third version:

```
struct commit_info {
    char hash[20];
    int grade[11];
};

struct commit_hint {
    char hint[8];
    commit_info* commit;
};

commit_hint* commits;
size_t N;

int get_grade3(const char* hash, int pset) {
    for (size_t i = 0; i != N; ++i) {
        if (memcmp(commits[i].hint, hash, 8) == 0
            && memcmp(commits[i].commit->hash, hash, 20) == 0) {
            return commits[i].commit->grade[pset];
        }
    }
    return -1;
}
```

QUESTION IO-3C. What is the expected number of cache lines accessed by `get_grade3`, in terms of N? (You may assume that N≤2000.)

Show solution

The assumption that N≤2000 means we're exceedingly unlikely to encounter a hint collision (i.e. a commit with the same hint, but different commit value). That means that we will examine N/2 commit_hint objects but ONLY ONE commit_info object. commit_hint objects are 16B big, so 4 hints/cache line: we examine N/4/2 = N/8 cache lines for hint objects, plus one for the info. $\lceil N/8 \rceil + 1$.

Hide solution

Hide solution

The fourth version is a hash table.

```
struct commit_info {
    char hash[20];
    int grade[11];
};

commit_info** commits;
size_t commits_hashsize;

int get_grade4(const char* hash, int pset) {
    // choose initial bucket
    size_t bucket;
    memcpy(&bucket, hash, sizeof(bucket));
    bucket = bucket % commits_hashsize;
    // search for the commit starting from that bucket
    while (commits[bucket] != nullptr) {
        if (memcmp(commits[bucket]->hash, hash, 20) == 0) {
            return commits[bucket]->grade[pset];
        }
        bucket = (bucket + 1) % commits_hashsize;
    }
    return -1;
}
```

QUESTION IO-3D. Assume that a call to `get_grade4` encounters `C` expected hash collisions (i.e., examines `C` buckets before finding the bucket that actually contains `hash`). What is the expected number of cache lines accessed by `get_grade4`, in terms of `N` and `C`?

Show solution

For commit_info objects, the lookup will access C cache lines, for the collisions, plus 1, for the successful lookup. But we must also consider the commits[bucket] pointers. We will examine at least 1 cache line for the successful bucket. The C collisions that happen before that will access C buckets. But those buckets might be divided among multiple cache lines; for instance, if C=1, 2 buckets are accessed, but if the first bucket=15, those buckets will be divided among 2 cache lines. The correct formula for buckets, including the final lookup, is 1 + C/8. Thus the total lookup will examine 2 + C + C/8 cache lines on average.

Hide solution

Hide solution

IO-4. IO caching and strace

Elif Batuman is investigating several program executables left behind by her ex-roommate Fyodor. She runs each executable under `strace` in the following way:

```
strace -o strace.txt ./EXECUTABLE files/text1meg.txt > files/out.txt
```

Help her figure out properties of these programs based on their system call traces.

QUESTION IO-4A. Program `./mysterya`:

```
open("files/text1meg.txt", O_RDONLY)    = 3
brk(0)                                  = 0x8193000
brk(0x81b5000)                          = 0x81b5000
read(3, "A", 1)                         = 1
write(1, "A", 1)                        = 1
read(3, "\n", 1)                        = 1
write(1, "\n", 1)                       = 1
read(3, "A", 1)                         = 1
write(1, "A", 1)                        = 1
read(3, "'", 1)                         = 1
write(1, "'", 1)                        = 1
read(3, "s", 1)                         = 1
write(1, "s", 1)                        = 1
...
```

Circle at least one option in each column.

1. Sequential IO	a. No read cache	i. No write cache	A. Cache size 4096
2. Reverse sequential IO	b. Unaligned read cache	ii. Write cache	B. Cache size 2048
3. Strided IO	c. Aligned read cache		C. Cache size 1024
			D. Other

Show solution

1, Sequential IO; a, No read cache; i, No write cache; D, Other

Hide solution

Hide solution

QUESTION IO-4B. Program `./mysteryb`:

```
open("files/text1meg.txt", O_RDONLY)    = 3
brk(0)                                  = 0x96c5000
brk(0x96e6000)                          = 0x96e6000
read(3, "A\nA's\nAA's\nAB's\nABM's\nAC's\nACTH'"..., 2048) = 2048
write(1, "A\nA's\nAA's\nAB's\nABM's\nAC's\nACTH'"..., 2048) = 2048
read(3, "kad\nAkron\nAkron's\nAl\nAl's\nAla\nAl"..., 2048) = 2048
write(1, "kad\nAkron\nAkron's\nAl\nAl's\nAla\nAl"..., 2048) = 2048
...
```

Circle at least one option in each column.

1. Sequential IO
2. Reverse sequential IO
3. Strided IO

a. No read cache
b. Unaligned read cache
c. Aligned read cache

i. No write cache
ii. Write cache

A. Cache size 4096
B. Cache size 2048
C. Cache size 1024
D. Other

Show solution

1, Sequential IO; b/c, (Un)aligned read cache; ii, Write cache; B, Cache size 2048

Hide solution

Hide solution

QUESTION IO-4C. Program `./mysteryc`:

```
open("files/text1meg.txt", O_RDONLY)    = 3
brk(0)                                  = 0x9064000
brk(0x9085000)                          = 0x9085000
fstat64(3, {st_mode=S_IFREG|0664, st_size=1048576, ...}) = 0
lseek(3, 1046528, SEEK_SET)             = 1046528
read(3, "ingau\nRheingau's\nRhenish\nRhianno"..., 2048) = 2048
write(1, "oR\ntlevesooR\ns'yenooR\nyenooR\ns't"..., 2048) = 2048
lseek(3, 1044480, SEEK_SET)             = 1044480
read(3, "Quinton\nQuinton's\nQuirinal\nQuisl"..., 2048) = 2048
write(1, "ehR\neehR\naehR\ns'hR\nhR\nsdlonyeR\ns"..., 2048) = 2048
lseek(3, 1042432, SEEK_SET)             = 1042432
read(3, "emyslid's\nPrensa\nPrensa's\nPrenti"..., 2048) = 2048
write(1, "\ns'nailitniuQ\nnailitniuQ\nnniuQ\ns"..., 2048) = 2048
lseek(3, 1040384, SEEK_SET)             = 1040384
read(3, "Pindar's\nPinkerton\nPinocchio\nPin"..., 2048) = 2048
write(1, "rP\ndilsymerP\ns'regnimerP\nregnime"..., 2048) = 2048
...
```

Circle at least one option in each column.

1. Sequential IO
2. Reverse sequential IO
3. Strided IO

a. No read cache
b. Unaligned read cache
c. Aligned read cache

i. No write cache
ii. Write cache

A. Cache size 4096
B. Cache size 2048
C. Cache size 1024
D. Other

Show solution

2, Reverse sequential IO; c, Aligned read cache; ii, Write cache; B, Cache size 2048

Hide solution

Hide solution

QUESTION IO-4D. Program `./mysteryd`:

```
open("files/text1meg.txt", O_RDONLY)     = 3
brk(0)                                   = 0x9a0e000
brk(0x9a2f000)                           = 0x9a2f000
fstat64(3, {st_mode=S_IFREG|0664, st_size=1048576, ...}) = 0
lseek(3, 1048575, SEEK_SET)              = 1048575
read(3, "o", 2048)                       = 1
lseek(3, 1048574, SEEK_SET)              = 1048574
read(3, "Ro", 2048)                      = 2
lseek(3, 1048573, SEEK_SET)              = 1048573
read(3, "\nRo", 2048)                    = 3
...
lseek(3, 1046528, SEEK_SET)              = 1046528
read(3, "ingau\nRheingau's\nRhenish\nRhianno"..., 2048) = 2048
write(1, "oR\ntlevesooR\ns'yenooR\nyenooR\ns't"..., 2048) = 2048
lseek(3, 1046527, SEEK_SET)              = 1046527
read(3, "eingau\nRheingau's\nRhenish\nRhiann"..., 2048) = 2048
lseek(3, 1046526, SEEK_SET)              = 1046526
read(3, "heingau\nRheingau's\nRhenish\nRhian"..., 2048) = 2048
...
```

Circle at least one option in each column.

1. Sequential IO
2. Reverse sequential IO
3. Strided IO

a. No read cache
b. Unaligned read cache
c. Aligned read cache

i. No write cache
ii. Write cache

A. Cache size 4096
B. Cache size 2048
C. Cache size 1024
D. Other

Show solution

2, Reverse sequential IO; b, Unaligned read cache; ii, Write cache; B, Cache size 2048

Hide solution

Hide solution

QUESTION IO-4E. Program `./mysterye`:

```
open("files/text1meg.txt", O_RDONLY)    = 3
brk(0)                                  = 0x93e5000
brk(0x9407000)                          = 0x9407000
read(3, "A", 1)                         = 1
read(3, "\n", 1)                        = 1
read(3, "A", 1)                         = 1
...
read(3, "A", 1)                         = 1
read(3, "l", 1)                         = 1
write(1, "A\nA's\nAA's\nAB's\nABM's\nAC's\nACTH'"..., 1024) = 1024
read(3, "t", 1)                         = 1
read(3, "o", 1)                         = 1
read(3, "n", 1)                         = 1
...
```

Circle at least one option in each column.

1. Sequential IO
2. Reverse sequential IO
3. Strided IO

a. No read cache
b. Unaligned read cache
c. Aligned read cache

i. No write cache
ii. Write cache

A. Cache size 4096
B. Cache size 2048
C. Cache size 1024
D. Other

Show solution

1, Sequential IO; a, No read cache; ii, Write cache; C, (write) cache size 1024

Hide solution

Hide solution

QUESTION IO-4F. Program `./mysteryf`:

```
open("files/text1meg.txt", O_RDONLY)    = 3
brk(0)                                  = 0x9281000
brk(0x92a3000)                          = 0x92a3000
read(3, "A\nA's\nAA's\nAB's\nABM's\nAC's\nACTH'"..., 4096) = 4096
write(1, "A", 1)                        = 1
write(1, "\n", 1)                       = 1
write(1, "A", 1)                        = 1
...
write(1, "A", 1)                        = 1
write(1, "l", 1)                        = 1
read(3, "ton's\nAludra\nAludra's\nAlva\nAlvar"..., 4096) = 4096
write(1, "t", 1)                        = 1
write(1, "o", 1)                        = 1
write(1, "n", 1)                        = 1
...
```

Circle at least one option in each column.

1. Sequential IO	a. No read cache	i. No write cache	A. Cache size 4096
2. Reverse sequential IO	b. Unaligned read cache	ii. Write cache	B. Cache size 2048
3. Strided IO	c. Aligned read cache		C. Cache size 1024
			D. Other

Show solution

1, Sequential IO; b/c, (un)aligned read cache; i, No write cache; A, Cache size 4096

Hide solution

Hide solution

IO-5. Processor cache

The following questions use the following C definition for an `N`x`M` matrix (the matrix has `N` rows and `M` columns).

```
struct matrix {
    unsigned N;
    unsigned M;
    double elt[0];
};

matrix* matrix_create(unsigned N, unsigned M) {
    matrix* m = (matrix*) malloc(sizeof(matrix) + N * M * sizeof(double));
    m->N = N;
    m->M = M;
    for (size_t i = 0; i < N * M; ++i) {
        m->elt[i] = 0.0;
    }
    return m;
}
```

Typically, matrix data is stored in row-major order: element m_{ij} (at row i and column j) is stored in `m->elt[i*m->M + j]`. We might write this in C using an inline function:

```
inline double* melt1(matrix* m, unsigned i, unsigned j) {
    return &m->elt[i * m->M + j];
}
```

But that's not the only possible method to store matrix data. Here are several more.

```
inline double* melt2(matrix* m, unsigned i, unsigned j) {
    return &m->elt[i + j * m->N];
}

inline double* melt3(matrix* m, unsigned i, unsigned j) {
    return &m->elt[i + ((m->N - i + j) % m->M) * m->N];
}

inline double* melt4(matrix* m, unsigned i, unsigned j) {
    return &m->elt[i + ((i + j) % m->M) * m->N];
}

inline double* melt5(matrix* m, unsigned i, unsigned j) {
    assert(m->M % 8 == 0);
    unsigned k = (i/8) * (m->M/8) + (j/8);
    return &m->elt[k*64 + (i % 8) * 8 + j % 8];
}
```

QUESTION IO-5A. Which method (of `melt1`–`melt5`) will have the best processor cache behavior if most matrix accesses use loops like this?

```
for (unsigned j = 0; j < 100; ++j) {
    for (unsigned i = 0; i < 100; ++i) {
        f(*melt(m, i, j));
    }
}
```

Show solution

> `melt2`
>
> Hide solution
>
> Hide solution

QUESTION IO-5B. Which method will have the best processor cache behavior if most matrix accesses use loops like this?

```
for (unsigned i = 0; i < 100; ++i) {
    f(*melt(m, i, i));
}
```

Show solution

> `melt3`
>
> Hide solution
>
> Hide solution

QUESTION IO-5C. Which method will have the best processor cache behavior if most matrix accesses use loops like this?

```
for (unsigned i = 0; i < 100; ++i) {
    for (unsigned j = 0; j < 100; ++j) {
        f(*melt(m, i, j));
    }
}
```

Show solution

> `melt1` (but `melt5` is almost as good)
>
> Hide solution
>
> Hide solution

QUESTION IO-5D. Which method will have the best processor cache behavior if most matrix accesses use loops like this?

```
for (int di = -3; di <= 3; ++di) {
    for (int dj = -3; dj <= 3; ++dj) {
        f(*melt(m, I + di, J + dj));
    }
}
```

Show solution

`melt5`

Hide solution

Hide solution

QUESTION IO-5E. Here is a matrix-multiply function in *ikj* order.

```
matrix* matrix_multiply(matrix* a, matrix* b) {
    assert(a->M == b->N);
    matrix* c = matrix_create(a->N, b->M);
    for (unsigned i = 0; i != a->N; ++i) {
        for (unsigned k = 0; k != a->M; ++k) {
            for (unsigned j = 0; j != b->M; ++j) {
                *melt(c, i, j) += *melt(a, i, k) * *melt(b, k, j);
            }
        }
    }
}
```

This loop order is cache-optimal when data is stored in `melt1` order. What loop order is cache-optimal for `melt2`?

Show solution

jki is best; kji is a close second.

Hide solution

Hide solution

QUESTION IO-5F. You notice that accessing a matrix element using `melt1` is very slow. After some debugging, it seems like the processor on which you are running code has a very slow multiply instruction. Briefly describe a change to `struct matrix` that would let you write a version of `melt1` with no multiply instruction. You may add members, change sizes, or anything you like.

Show solution

Example answers:

1. Add a `double** rows` member that points to each row so you don't need to multiply
2. Round M up to a power of 2 and use shifts

Hide solution

Hide solution

IO-6. Caching

Assume that we have a cache that holds four slots. Assume that each letter below indicates an access to a block. Answer the following questions as they pertain to the following sequence of accesses.

```
E D C B A E D A A A B C D E
```

QUESTION IO-6A. What is the hit rate assuming an LRU replacement policy?

Show solution

	E	D	C	B	A	E	D	A	A	A	B	C	D	E
1	Ⓔ				Ⓐ			A	A	A				Ⓔ
2		Ⓓ				Ⓔ						Ⓒ		
3			Ⓒ				Ⓓ						D	
4				Ⓑ							B			

The hit rate is 5/14.

Hide solution

Hide solution

QUESTION IO-6B. What pages will you have in the cache at the end of the run?

Show solution

B C D E

Hide solution

Hide solution

QUESTION IO-6C. What is the best possible hit rate attainable if you could see into the future?

Show solution

With Bélády's algorithm we get:

	E	D	C	B	A	E	D	A	A	A	B	C	D	E
1	Ⓔ					E								E
2		Ⓓ					D						D	
3			Ⓒ		Ⓐ			A	A	A		Ⓒ		
4				Ⓑ							B			

So 8/14 (4/7).

Hide solution

Hide solution

IO-7. Caching

Intel and CrossPoint have announced a new persistent memory technology with performance approaching that of DRAM. Your job is to calculate some performance metrics to help system architectects decide how to best incorporate this new technology into their platform.

Let's say that it takes 64ns to access one (32-bit) word of main memory (DRAM) and 256ns to access one (32-bit) word of this new persistent memory, which we'll call NVM (non-volatile memory). The block size of the NVM is 256 bytes. The NVM designers are quite smart and although it takes a long time to access the first byte, when you are accessing NVM sequentially, the devices perform read ahead and stream data efficiently -- at 32 GB/second, which is identical to the bandwidth of DRAM.

QUESTION IO-7A. Let's say that we are performing random accesses of 32 bits (on a 32-bit processor). What fraction of the accesses must be to main memory (as opposed to NVM) to achieve performance within 10% of DRAM?

Show solution

Let X be the fraction of accesses to DRAM: access time = 64X + 256(1-X). We want that to be <= 1.1*64 (within 10% of DRAM). So, 1.1*64 = 70.4. So, let's solve for: 64X + 256(1-X) = 70.4.

```
64X + 256 - 256X = 70.4.
(256X - 64X) = 256 - 70.4
192X = 186
X = 186/192
about .97
```

So, we need a hit rate in main memory of 97%

Hide solution

Hide solution

QUESTION IO-7B. Let's say that they write every byte of a 256 block in units of 32 bits. How much faster will write-back cache perform relative to a write-through cache? (An approximate order of magnitude will be sufficient; showing work can earn partial credit.)

Show solution

Write-through is going to cost 256ns/4 byte write = 256 * 64 = $2^{8*2}6$ = 2^14 = 16 K ns which is roughly 16 microseconds. If we assume a write-back, then it will take us 64 * 64ns to write into the DRAM, but then we get to stream the data from DRAM into the NVM at a rate of 32 GB/sec. So, 64*64 ns = 2^12 ns = 4 microseconds to write into DRAM. Let's convert 32 GB/second ito KB -- that's about 32 KB/microsecond. We need 1/4 of 1 KB which is 1/128 of a microsecond, which is about 8 ns. So, it's really really really fast to stream the data -- once you know that, then you also realize that the real difference is just the relative cost of writing to DRAM versus the cost of writing to NVM. So, the writeback cache is almost 4 times faster than the write through cache. You can get full credit by saying something like: the time to stream the data out of the DRAM into the NVM at the sequential speed is tiny relative to the time to write even a single word to DRAM, so the ultimate difference is the difference in writing to DRAM relative to NVM which is a ratio of 4:1. So, the writeback cache is about 4 times faster (because it is running at almost the full DRAM speed).

Hide solution

Hide solution

QUESTION IO-7C. Why might you not want to use a write-back cache?

Show solution

A write-through cache will have very different persistence guarantees. If you need every 4- byte write to be persistent, then you have no choice but to implement a write-through cache.

Hide solution

Hide solution

IO-8. Reference strings

The following questions concern the FIFO (First In First Out), LRU (Least Recently Used), and LFU (Least Frequently Used) cache eviction policies.

Your answers should refer to **seven-item** reference strings made up of digits in the range 0–9. An example answer might be "1231231". In each case, the reference string is processed by a 3-slot cache that's initially empty.

QUESTION IO-8A. Give a reference string that has a 1/7 hit rate in all three policies.

Show solution

1123456

Hide solution

Hide solution

QUESTION IO-8B. Give a reference string that has a 6/7 hit rate in all three policies.

Show solution

1111111

Hide solution

Hide solution

QUESTION IO-8C. Give a reference string that has *different* hit rates under LRU and LFU policies, and compute the hit rates.

String:

LRU hit rate:

LFU hit rate:

Show solution

String: 1234111

LRU hit rate: 2/7

LFU hit rate: 3/7

Hide solution

Hide solution

QUESTION IO-8D. Give a reference string that has *different* hit rates under FIFO and LRU policies, and compute the hit rates.

String:

FIFO hit rate:

LRU hit rate:

Show solution

String: 1231411

FIFO hit rate: 2/7

LRU hit rate: 3/7

Hide solution

Hide solution

QUESTION IO-8E. Now let's assume that you know a reference string in advance. Given a 3-slot cache and the following reference string, what caching algorithm discussed in class and/or exercises would produce the best hit rate, and would would that hit rate be?

“12341425321521”

Show solution

Bélády’s optimal algorithm (ACCENTS REQUIRED FOR FULL CREDIT!)(!*#^‡°

1m 2m 3m 4m [124] 1h 4h 2h 5m [125] 3m [123] 2h 1h 5m [125] 2h 1h

7/14 = 1/2

Hide solution

Hide solution

IO-9. Caching: Access times and hit rates

Recall that x86-64 instructions can access memory in units of 1, 2, 4, or 8 bytes at a time. Assume we are running on an x86-64-like machine with 1024-byte cache lines. Our machine takes 32ns to access a unit if the cache hits, regardless of unit size. If the cache misses, an additional 8160ns are required to load the cache, for a total of 8192ns.

QUESTION IO-9A. What is the average access time per access to access all the data in a cache line as an array of 256 integers, starting from an empty cache?

Show solution

(8192ns * 1 + 32ns * 255)/256 (= 63.875)

Hide solution

Hide solution

QUESTION IO-9B. What unit size (1, 2, 4, or 8) minimizes the access time to access all data in a cache line, starting from an empty cache?

Show solution

8

Hide solution

Hide solution

QUESTION IO-9C. What unit size (1, 2, 4, or 8) *maximizes* the *hit rate* to access all data in a cache line, starting from an empty cache?

Show solution

1

Hide solution

Hide solution

IO-10. Single-slot cache code

Donald Duck is working on a single-slot cache for reading. He's using the `pos_tag`/`end_tag` representation, which is:

```
struct io61_file {
   int fd;
   unsigned char cbuf[BUFSIZ];
   off_t tag;      // file offset of first character in cache (same as before)
   off_t end_tag;  // file offset one past last valid char in cache; end_tag - tag ==
old `csz`
   off_t pos_tag;  // file offset of next char to read in cache; pos_tag - tag == old
`cpos`
};
```

Here's our solution code; in case you want to scribble, the code is copied in the appendix.

```
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    size_t pos = 0;
    while (pos != sz) {
        if (f->pos_tag < f->end_tag) {
            ssize_t n = sz - pos;
            if (n > f->end_tag - f->pos_tag)
                n = f->end_tag - f->pos_tag;
            memcpy(&buf[pos], &f->cbuf[f->pos_tag - f->tag], n);
            f->pos_tag += n;
            pos += n;
        } else {
            f->tag = f->end_tag;
            ssize_t n = read(f->fd, f->cbuf, BUFSIZ);
            if (n > 0)
                f->end_tag += n;
            else
                return pos ? pos : n;
        }
    }
    return pos;
}
```

Donald has ideas for "simplifying" this code. Specifically, he wants to try each of the following independently:

A. Replacing line 4 with "`if (f->pos_tag <= f->end_tag) {`".
B. Removing lines 6–7.
C. Removing line 9.
D. Removing lines 16–17.

QUESTION IO-10A. Which simplifications could lead to undefined behavior? List all that apply or say "none."

Show solution

B (removing 6–7): you read beyond the cache buffer.

Hide solution

Hide solution

QUESTION IO-10B. Which simplifications could cause `io61_read` to loop forever without causing undefined behavior? List all that apply or say "none."

Show solution

A (replacing 4): you spin forever after exhausting the cache

D (removing 16–17): you spin forever if the file runs out of data or has a persistent error

Hide solution

Hide solution

QUESTION IO-10C. Which simplifications could lead to `io61_read` returning incorrect data in `buf`, meaning that the data read by a series of `io61_read` calls won't equal the data in the file? List all that apply or say "none."

Show solution

B (removing 6–7): you read garbage beyond the cache buffer

C (removing 9): you read the same data over & over again.

Hide solution

Hide solution

QUESTION IO-10D. Chastened, Donald decides to optimize the code for a specific situation, namely when `io61_read` is called with a `sz` that is larger than `BUFSIZ`. He wants to add code after line 11, like so, so that fewer `read` system calls will happen for large `sz`:

```
        } else if (sz - pos > BUFSIZ) {
            // DONALD'S CODE HERE




        } else {
            f->tag = f->end_tag;
            ....
```

Finish Donald's code. Your code should maintain the relevant invariants between `tag`, `pos_tag`, `end_tag`, and the file position, but you need not keep `tag` aligned.

Show solution

```
ssize_t n = read(f->fd, &buf[pos], sz - pos);
if (n > 0) {
    f->tag = f->pos_tag = f->end_tag = f->end_tag + n;
    pos += n;
} else {
    return pos ? pos : n;
}
```

Hide solution

Hide solution

IO-11. Caching

QUESTION IO-11A. If it takes 200ns to access main memory, which of the following two caches will produce a lower average access time?

- A cache with a 10ns access time that produces a 90% hit rate
- A cache with a 20ns access time that produces a 98% hit rate

Show solution

Let's compute average access time for each case:

```
.9 * 10 + .1 * 200 = 9 + 20 = 29
.98 * 20 + .02 * 200 = 19.6 + 4 = 23.6
```

The 20ns cache produces a lower average access time.

Hide solution

Hide solution

QUESTION IO-11B. Let’s say that you have a direct-mapped cache with four slots. A page with page number `N` must reside in the slot numbered `N % 4`. What is the best hit rate this could achieve given the following sequence of page accesses?

```
3 6 7 5 3 2 1 1 1 8
```

Show solution

Since it's direct mapped, each item can go in only one slot, so if we list the slots for each access, we get:

```
3 2 3 1 3 2 1 1 1 0
```

The only hits are the 2 1's, so your hit rate is 2/10 or 20% or .2.

Hide solution

Hide solution

QUESTION IO-11C. What is the best hit rate a *fully-associative* four-slot cache could achieve for that sequence of page accesses? (A fully-associative cache may put any page in any slot. You may assume you know the full reference stream in advance.)

Show solution

Now we can get hits for 3, 1, and 1, so our hit rate goes to 3/10 or 30%.

Hide solution

Hide solution

QUESTION IO-11D. What hit rate would the fully-associative four-slot cache achieve if it used the LRU eviction policy?

Show solution

> Still 30% (3/10, .3)
>
> Hide solution
>
> Hide solution

IO-12. I/O traces

QUESTION IO-12A. Which of the following programs *cannot* be distinguished by the output of the `strace` utility, not considering `open` calls? List all that apply; if multiple indistinguishable groups exist (e.g., A, B, & C can't be distinguished, and D & E can't be distinguished, but the groups *can* be distinguished from each other), list them all.

1. Sequential byte writes using stdio
2. Sequential byte writes using system calls
3. Sequential byte writes using system calls and `O_SYNC`
4. Sequential block writes using stdio and block size 2
5. Sequential block writes using system calls and block size 2
6. Sequential block writes using system calls and `O_SYNC` and block size 2
7. Sequential block writes using stdio and block size 4096
8. Sequential block writes using system calls and block size 4096
9. Sequential block writes using system calls and `O_SYNC` and block size 4096

Show solution

> 1, 4, 7, 8 can't be distinguished.
>
> If you consider `open`, then the O_SYNC cases and the others become distinguishable. Assuming that, 2&3 can't be distinguished, 4&5 can't be distinguished, and 1,4,7,8,9 can't be distinguished.
>
> Hide solution
>
> Hide solution

QUESTION IO-12B. Which of the programs in Part A cannot be distinguished using `blktrace` output? List all that apply.

Show solution

1, 2, 4, 5, 7, 8, and possibly 9 can't be distinguished.

Hide solution

Hide solution

QUESTION IO-12C. The buffer cache is coherent. Which of the following operating system changes could make the buffer cache **incoherent?** List all that apply.

1. Application programs can obtain direct read access to the buffer cache
2. Application programs can obtain direct write access to the disk, bypassing the buffer cache
3. Other computers can communicate with the disk independently
4. The computer has a uninterruptible power supply (UPS), ensuring that the operating system can write the contents of the buffer cache to disk if main power is lost

Show solution

#2, #3

Hide solution

Hide solution

QUESTION IO-12D. The stdio cache is **incoherent.** Which of the operating system changes from Part C could make the stdio cache **coherent?** List all that apply.

Show solution

#1

Hide solution

Hide solution

IO-13. Reference strings and eviction

QUESTION IO-13A. When demonstrating cache eviction in class, we modeled a completely *reactive* cache, meaning that the cache performed at most one load from slow storage per access. Name a class of reference string that will have a 0% hit rate on any cold reactive cache. For partial credit, give several examples of such reference strings.

Show solution

Sequential access

Hide solution

Hide solution

QUESTION IO-13B. What cache optimization can be used to improve the hit rate for the class of reference string in Part A? One word is enough; put the best choice.

Show solution

Prefetching. Batching is an OK answer, but mostly because it involves prefetching when done for reads.

Hide solution

Hide solution

QUESTION IO-13C. Give a single reference string with the following properties:

- There exists a cache size and eviction policy that gives a 70% hit rate for the string.
- There exists a cache size and eviction policy that gives a 0% hit rate for the string.

Show solution

Example: 1231231231. 70% on any 3-slot cache; 0% on a 1-slot cache.

Hide solution

Hide solution

QUESTION IO-13D. Put the following eviction algorithms in order of how much space they require for per-slot metadata, starting with the least space and ending with the most space. (Assume the slot order is fixed, so once a block is loaded into slot *i*, it stays in slot *i* until it is evicted.) For partial credit say what you think the metadata would be.

1. FIFO
2. LRU
3. Random

Show solution

Random, then FIFO, then LRU. Random needs no additional metadata; FIFO can deal with a single integer for the whole cache, pointing to the next index to use; LRU nees a least-recently-used time.

Hide solution

Hide solution

IO-14. Cache code

Several famous musicians have just started working on CS61 Problem Set

3. They share the following code for their read-only, sequential, single-slot cache:

```
struct io61_file {
    int fd;
    unsigned char buf[4096];
    size_t pos;    // position of next character to read in `buf`
    size_t sz;     // number of valid characters in `buf`
};

int io61_readc(io61_file* f) {
    if (f->pos >= f->sz) {
        f->pos = f->sz = 0;
        ssize_t nr = read(f->fd, f->buf, sizeof(f->buf));
        if (nr <= 0) {
            f->sz = 0;
            return -1;
        } else {
            f->sz = nr;
        }
    }
    int ch = f->buf[f->pos];
    ++f->pos;
    return ch;
}
```

But they have different `io61_read` implementations. Donald (Lambert)'s is:

```
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    return read(f->fd, buf, sz);
}
```

Solange (Knowles)'s is:

```c
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    for (size_t pos = 0; pos < sz; ++pos, ++buf) {
        *buf = io61_readc(f);
    }
    return sz;
}
```

Caroline (Shaw)'s is:

```c
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    if (f->pos >= f->sz) {
        return read(f->fd, buf, sz);
    } else {
        int ch = io61_readc(f);
        if (ch < 0) {
            return 0;
        }
        *buf = ch;
        return io61_read(f, buf + 1, sz - 1) + 1;
    }
}
```

You are testing each of these musicians' codes by executing a sequence of `io61_readc` and/or `io61_read` calls on an input file and printing the resulting characters to standard output. There are no seeks, and your test programs print until end of file, so your tests' output should equal the input file's contents.

You should assume for these questions that **no `read` system call ever returns -1**.

QUESTION IO-14A. Describe an access pattern—that is, a sequence of `io61_readc` and/or `io61_read` calls (with lengths)—for which Donald's code can return incorrect data.

Show solution

`io61_readc`, `io61_read(1)`, …: Any alternation between readc and read is a disaster.

Hide solution

Hide solution

QUESTION IO-14B. Which of these musicians' codes can generate an output file with incorrect length?

For the remaining parts, assume the problem in Part B has been corrected, so that all musicians' codes generate output files with correct lengths.

Show solution

Solange's code never returns end-of-file; she mistakes EOF for a valid return value. If a program using Donald's code calls `io61_readc` once and then switches to `io61_read`, then they will read too few bytes (bytes in the readc buffer won't be returned).

Hide solution

Hide solution

QUESTION IO-14C. Give an access pattern for which Solange's code will return correct data and outperform Donald's, or vice versa, and say whose code will win.

Show solution

Solange's code will outperform Donald's when `io61_read` is called with small sizes. Donald's code will outperform Solange's when `io61_read` is called with large sizes.

Hide solution

Hide solution

QUESTION IO-14D. Suggest a small change (≤10 characters) to Caroline's code that would, most likely, make it perform at least as well as *both* Solange's and Donald's codes on all access patterns. Explain briefly.

Show solution

I would change Caroline's test from `if (f->pos >= f->sz)` to `if (f->pos >= f->sz && sz >= 1024)` (or 4096, or something similar). This uses the cache when read is called with small sizes, but avoids the extra copy when read is called with large sizes.

Hide solution

Hide solution

IO-15. Caches

Parts A–C concern different implementations of Pset 3's stdio cache. Assume a program that reads a 32768-byte file a character at a time, like this:

```
while (io61_readc(inf) != EOF) {
}
```

This program will call `io61_readc` 32769 times. ($32769 = 2^{15} + 1 = 8\times2^{12} + 1$; the +1 accounts for the EOF return.) But the cache implementation might make many fewer system calls.

QUESTION IO-15A. How many `read` system calls are required assuming a single-slot, 4096-byte io61 cache?

Show solution

> 9
>
> Hide solution
>
> Hide solution

QUESTION IO-15B. How many `read` system calls are required assuming an eight-slot, 4096-byte io61 cache?

Show solution

> 9
>
> Hide solution
>
> Hide solution

QUESTION IO-15C. How many `mmap` system calls are required assuming an `mmap`-based io61 cache?

Show solution

> 1
>
> Hide solution
>
> Hide solution

Parts D–F concern cache implementations and styles. We discussed many caches in class, including:

A. The buffer cache
B. The processor cache
C. Single-slot aligned stdio caches
D. Single-slot unaligned stdio caches
E. Circular bounded buffers

QUESTION IO-15D. Which of those caches are implemented entirely in hardware? List all that apply.

Show solution

B

Hide solution

Hide solution

QUESTION IO-15E. Which of those *software* caches could help speed up reverse sequential access to a disk file? List all that apply.

Show solution

A, C

Hide solution

Hide solution

QUESTION IO-15F. Which of those *software* caches could help speed up access to a pipe or network socket? List all that apply.

Show solution

C, D, E

Hide solution

Hide solution

Kernel exercises

Many exercises that seem less appropriate this year, or which cover topics that we haven't covered in class, are marked with ⚠️. However, we may have missed some.

KERN-1. Virtual memory

QUESTION KERN-1A. What is the x86-64 page size? Circle all that apply.

1. 4096 bytes
2. 64 cache lines
3. 256 words
4. `0x1000` bytes
5. 2^{16} bits
6. None of the above

Show solution

#1, #2, #4. The most common x86-64 cache line size is 64 = 2^6 bytes, and $2^6 \times 2^6 = 2^{12}$, but there may have been some x86-64 processors with 128-byte cache lines. The word size is 8; 256×8 = 2048, not 4096. There are 8 bits per byte; $2^{16}/8 = 2^{13}$, not 2^{12}.

Hide solution

Hide solution

The following questions concern the sizes of page tables. Answer the questions in units of pages. For instance, the page tables in WeensyOS each contained one level-4 page table page (the highest level, corresponding to address bits 39-47); one level-3 page table page; one level-2 page table page; and two level-1 page table pages, for a total size of 5 pages per page table.

QUESTION KERN-1B. What is the maximum size (in pages) of an x86-64 page table (page tables only, not destination pages)? You may write an expression rather than a number.

Show solution

```
                       1    level-4 page table page
+                    512    level-3 page table pages
+              512 * 512    level-2 page table pages
+        512 * 512 * 512    level-1 page table pages
-----------------------
  2^27 + 2^18 + 2^9 + 1
= 0x8040201 = 134480385 page table pages
```

Hide solution

Hide solution

QUESTION KERN-1C. What is the minimum size (in pages) of an x86-64 page table that would allow a process to access 2^{21} distinct physical addresses?

Show solution

4 is a good answer—x86-64 page tables have four levels—but the best answer is one.

Whaaat?! Consider a level-4 page table whose first entry refers *to the level-4 page table page itself*, and the other entries referred to different pages. Like this:

Physical address	Index	Contents
`0x1000`	0	`0x1007`
`0x1008`	1	`0x2007`
`0x1010`	2	`0x3007`
`0x1018`	3	`0x4007`
…	…	…
`0x1ff8`	511	`0x200007`

With this page table in force, the 2^{21} virtual addresses `0x0` through `0x1FFFFF` access the 2^{21} distinct physical addresses `0x1000` through `0x200FFF`.

Hide solution

Hide solution

The 64-bit x86-64 architecture is an extension of the 32-bit x86 architecture, which used 32-bit virtual addresses and 32-bit physical addresses. But before 64 bits came along, Intel extended 32-bit x86 in a more limited way called Physical Address Extension (PAE). Here's how they differ.

- PAE allows 32-bit machines to access up to 2^{52} bytes of physical memory (which is about 4000000 GB).

That is, virtual addresses are 32 bits, and physical addresses are 52 bits.

- The x86-64 architecture evolves the x86 architecture to a 64-bit word size. x86-64 pointers are 64 bits wide instead of 32. However, only 48 of those bits are meaningful: the upper 16 bits of each virtual address are ignored. Thus, virtual addresses are 48 bits. As with PAE, physical addresses are 52 bits.

QUESTION KERN-1D. Which of these two machines would support a higher number of concurrent processes?

1. x86-32 with PAE with 100 GB of physical memory.
2. x86-64 with 20 GB of physical memory.

Show solution

> #1 x86-32 with PAE. Each concurrent process occupies some space in physical memory, and #1 has more physical memory.
>
> (Real operating systems swap, so *either* machine could support more processes than fit in virtual memory, but this would cause thrashing. #1 supports more processes before it starts thrashing.)
>
> Hide solution
>
> Hide solution

QUESTION KERN-1E. Which of these two machines would support a higher maximum number of threads per process?

1. x86-32 with PAE with 100 GB of physical memory.
2. x86-64 with 20 GB of physical memory.

Show solution

> #2 x86-64. Each thread in a process needs some address space for its stack, and an x86-64 process address space is much bigger than an x86-32's.
>
> Hide solution
>
> Hide solution

KERN-2. Virtual memory and kernel programming

The WeensyOS kernel occupies virtual addresses 0 through 0xFFFFF; the process address space starts at `PROC_START_ADDR` == 0x100000 and goes up to (but not including) `MEMSIZE_VIRTUAL` == 0x300000.

QUESTION KERN-2A. True or false: On x86-64 Linux, like on WeensyOS, the kernel occupies low virtual addresses.

Show solution

> False
>
> Hide solution
>
> Hide solution

QUESTION KERN-2B. On WeensyOS, which region of a process's address space is closest to the kernel's address space? Choose from code, data, stack, and heap.

Show solution

> Code
>
> Hide solution
>
> Hide solution

QUESTION KERN-2C. On Linux on an x86-64 machine, which region of a process's address space is closest to the kernel's address space? Choose from code, data, stack, and heap.

Show solution

> Stack
>
> Hide solution
>
> Hide solution

The next problems consider implementations of virtual memory features in a WeensyOS-like operating system. Recall that the WeensyOS `sys_page_alloc(addr)` system call allocates a new physical page at the given virtual address. Here's an example kernel implementation of `sys_page_alloc`, taken from the WeensyOS `syscall` function:

```
case SYSCALL_PAGE_ALLOC: {
    uintptr_t addr = current->regs.reg_rdi;

    /* [A] */

    void* pg = kalloc(PAGESIZE);
    if (!pg) { // no free physical pages
        console_printf(CPOS(24, 0), 0x0C00, "Out of physical memory!\n");
        return -1;
    }

    /* [B] */

    // and map it into the user's address space
    vmiter(current->pagetable, addr).map((uintptr_t) pg, PTE_P | PTE_W | PTE_U);

    /* [C] */

    return 0;
}
```

(Assume that `kalloc` and `kfree` are correctly implemented.)

QUESTION KERN-2D. Thanks to insufficient checking, this implementation allows a WeensyOS process to crash the operating system or even take it over. This kernel is not isolated. What the kernel *should* do is return −1 when the calling process supplies bad arguments. Write code that, if executed at slot **[A]**, would preserve kernel isolation and handle bad arguments correctly.

Show solution

```
if (addr % PAGESIZE != 0 || addr < PROC_START_ADDR || addr >= MEMSIZE_VIRTUAL) {
    return -1;
}
```

Hide solution

Hide solution

QUESTION KERN-2E. This implementation has another problem, which the following process would trigger:

```
void process_main() {
    heap_top = /* ... first address in heap region ... */;
    while (1) {
        sys_page_alloc(heap_top);
        sys_yield();
    }
}
```

This process code repeatedly allocates a page at the *same* address. What *should* happen is that the kernel should repeatedly deallocate the old page and replace it with a newly-allocated zeroed-out page. But that's not what *will* happen given the example implementation.

What will happen instead? And what is the name of this kind of problem?

Show solution

> Eventually the OS will run out of physical memory. At least it will print "`Out of physical memory!`" (that was in the code we provided). This is a memory leak.
>
> Hide solution
>
> Hide solution

QUESTION KERN-2F. Write code that would fix the problem, and name the slot in the `SYSCALL_PAGE_ALLOC` implementation where your code should go. (You may assume that this version of WeensyOS never shares process pages among processes.)

Show solution

```
if (vmiter(current, addr).user()) {
    kfree((void*) vmiter(current, addr).pa());
}
```

> This goes in slot A or slot B. Slot C is too late; it would free the newly mapped page.
>
> Hide solution
>
> Hide solution

KERN-3. Kernel programming

WeensyOS processes are quite isolated: the only way they can communicate is by using the console. Let's design some system calls that will allow processes to explicitly share pages of memory. Then the processes can communicate by writing and reading the shared memory region. Here are two new WeensyOS system calls that allow minimal page sharing; they return 0 on success and –1 on error.

- **`int share(pid_t p, void* addr)`**
  Allow process `p` to access the page at address `addr`.
- **`int attach(pid_t p, void* remote_addr, void* local_addr)`**
  Access the page in process `p`'s address space at address `remote_addr`. That physical page is added to the calling process's address space at address `local_addr`, replacing any page that was previously mapped there. It is an error if `p` has not shared the page at `remote_addr` with the calling process.

Here's an initial implementation of these system calls, written as clauses in the WeensyOS kernel's `syscall` function.

```
case SYSCALL_SHARE: {
    pid_t p = current->regs.reg_rdi;
    uintptr_t addr = current->regs.reg_rsi;

    /* [A] */

    int shindex = current->nshared;
    if (shindex >= MAX_NSHARED) {
        return -1;
    }

    /* [B] */

    ++current->nshared;
    current->shared[shindex].sh_addr = addr;
    current->shared[shindex].sh_partner = p;
    return 0;
}

case SYSCALL_ATTACH: {
    pid_t p = current->regs.reg_rdi;
    uintptr_t remote_addr = current->regs.reg_rsi;
    uintptr_t local_addr = current->regs.reg_rdx;

    /* [C] */

    int shindex = -1;
    for (int i = 0; i < processes[p].nshared; ++i) {
        if (processes[p].shared[i].sh_addr == remote_addr
            && processes[p].shared[i].sh_partner == current->p_pid) {
            shindex = i;
        }
    }
    if (shindex == -1) {
        return -1;
    }
```

```
    /* [D] */

    vmiter it(processes[p].pagetable, remote_addr);

    /* [E] */

    vmiter(current->pagetable, local_addr).map(it.pa(), PTE_P | PTE_W | PTE_U);

    /* [F] */

    return 0;
}
```

Some notes:

- The implementation stores sharing records in an array. A process may call `share` successfully at most `MAX_NSHARED` times. After that, its future `share` calls will return an error.
- `processes[p].nshared` is initialized to 0 for all processes.
- Assume that WeensyOS has been implemented as in Problem Set 4 up through **step 6** (shared read-only memory).

QUESTION KERN-3A. True or false: Given this implementation, a single WeensyOS process can cause the kernel to crash simply by calling `share` one or more times (with no process ever calling `attach`). If true, give an example of a call or calls that would likely crash the kernel.

Show solution

> False
>
> Hide solution
>
> Hide solution

QUESTION KERN-3B. True or false: Given this implementation, a single WeensyOS process can cause the kernel to crash simply by calling `attach` one or more times (with no process ever calling `share`). If true, give an example of a call or calls that would likely crash the kernel.

Show solution

True. If the user supplies an out-of-range process ID argument, the kernel will try to read out of bounds of the `processes` array. Example call: `attach(0x1000000, 0, 0)`.

Hide solution

Hide solution

QUESTION KERN-3C. True or false: Given this implementation, WeensyOS processes 2 and 3 could work together to obtain write access to the kernel code located at address `KERNEL_START_ADDR`. If true, give an example of calls that would obtain this access.

Show solution

True, since the `attach` and `share` code don't check whether the user process is allowed to access its memory. An example:

```
#2: share(3, KERNEL_START_ADDR)
#3: attach(2, KERNEL_START_ADDR, 0x110000)
```

Hide solution

Hide solution

QUESTION KERN-3D. True or false: Given this implementation, WeensyOS processes 2 and 3 could work together to obtain write access to *any* memory, *without* crashing or modifying kernel code or data. If true, give an example of calls that would obtain access to a page mapped at address 0x110000 in process 5.

Show solution

The best answer here is false. Processes are able to gain access to any page mapped in one of their page tables. But it's not clear whether 5's 0x110000 is mapped in either of the current process's page tables. Now, 2 and 3 could first read the `processes` array (via share/attach) to find the physical address of 5's page table; then, if 2 and 3 are in luck and the page table itself is mapped in their page table, they could read that page table to find the physical address of 0x110000; and then, if 2 and 3 are in luck again, map that page using the VA accessible in one of *their* page tables (which would differ from 0x110000). But that might not work.

Hide solution

Hide solution

QUESTION KERN-3E. True or false: Given this implementation, WeensyOS child processes 2 and 3 could work together to modify the code run by a their shared parent, process 1, *without* crashing or modifying kernel code or data. If true, give an example of calls that would obtain write access to process 1's code, which is mapped at address PROC_START_ADDR.

Show solution

True; since process code is shared after step 6, the children can map *their own* code read/write, and this is the same code as the parent's.

```
#2: share(3, PROC_START_ADDR)
#3: attach(2, PROC_START_ADDR, PROC_START_ADDR)
```

Hide solution

Hide solution

QUESTION KERN-3F. Every "true" answer to the preceding questions is a bug in WeensyOS's process isolation. Fix these bugs. Write code snippets that address these problems, and say where they go in the WeensyOS code (for instance, you could refer to bracketed letters to place your snippets); or for partial credit describe what your code should do.

Show solution

Here's one possibility.

Prevent `share` from sharing an invalid address in `[A]`:

```
if ((addr & 0xFFF) || addr < PROC_START_ADDR) {
    return -1;
}
```

NB don't need to check `addr < MEMSIZE_VIRTUAL` as long as we check the permissions from `vmiter` below (but that doesn't hurt).

Prevent `attach` from accessing an invalid process or mapping at an invalid address in `[B]`:

```
if (p >= NPROC
    || (local_addr & 0xFFF)
    || local_addr < PROC_START_ADDR
    || local_addr >= MEMSIZE_VIRTUAL) {
    return -1;
}
```

We do need to check `MEMSIZE_VIRTUAL` here.

Check the mapping at `remote_addr` before installing it in `[E]`:

```
if (!it.user()) {
    return -1;
}
```

Also, in the `….map` call, use `it.perm()` instead of `PTE_P|PTE_W|PTE_U`.

For greatest justice we would also fix a potential memory leak caused by `attach`ing over an address that already had a page, but this isn't necessary.

Hide solution

Hide solution

KERN-4. Teensy OS VM System

The folks at Teensy Computers, Inc, need your help with their VM system. The hardware team that developed the VM system abruptly left and the folks remaining aren't quite sure how VM works. I volunteered you to help them.

The Teensy machine has a 16-bit virtual address space with 4 KB pages. The Teensy hardware specifies a single-level page table. Each entry in the page table is 16-bits. Eight of those bits are reserved for the physical page number and 8 of the bits are reserved for flag values. Sadly, the hardware designers did not document what the bits do!

QUESTION KERN-4A. How many pages are in the Teensy virtual address space?

Show solution

16 (2^4)

Hide solution

Hide solution

QUESTION KERN-4B. How many bits comprise a physical address?

Show solution

20 (8 bits of physical page number + 12 bits of page offset)

Hide solution

Hide solution

QUESTION KERN-4C. Is the physical address space larger or smaller than the virtual address space?

Show solution

Larger!

Hide solution

Hide solution

QUESTION KERN-4D. Write, in hex, a `PAGE_OFFSET_MASK` (the value that when anded with an address returns the offset of the address on a page).

Show solution

0xFFF

Hide solution

Hide solution

QUESTION KERN-4E. Write a C expression that takes a virtual address, in the variable `vaddr`, and returns the virtual page number.

Show solution

`(vaddr >> 12)` OR `((vaddr) & 0xF000 >> 12)`

Hide solution

Hide solution

You are now going to work with the Teensy engineers to figure out what those other bits in the page table entries mean! Fortunately, they have some engineering notes from the hardware team—they need your help in making sense of them. Each letter below has the contents of a note, state what you can conclude from that note about the lower 8 bits of the page table entries.

QUESTION KERN-4F. “Robin, I ran 8 tests using a kernel that did nothing other than loop infinitely -- for each test I set a different bit in all the PTEs of the page table. All of them ended up in the exception handler except for the one where I set bit 4. Any idea what this means?”

Show solution

Bit 4 is the “present/valid bit”, the equivalent of x86 PTE_P.

Hide solution

Hide solution

QUESTION KERN-4G. “Lynn, I'm writing a memory test that iterates over all of memory making sure that I can read back the same pattern I write into memory. If I don't set bit 7 of the page table entries to 1, I get permission faults. Do you know what might be happening?”

Show solution

Bit 1 is the “writable bit”, the equivalent of x86 PTE_W.

Hide solution

Hide solution

QUESTION KERN-4H. “Pat, I almost have user level processes running! It seems that the user processes take permission faults unless I have both bit 4 and bit 3 set. Do you know why?”

Show solution

Bit 3 is the "user/unprivileged bit", the equivalent of x86 PTE_U.

Hide solution

Hide solution

KERN-5. Teensy OS Page Tables

The Teensy engineers are well on their way now, but they do have a few bugs and they need your help debugging the VM system. They hand you the following page table, using x86-64 notation for permissions, and need your help specifying correct behavior for the operations that follow.

	Entry contents:	
Index	Page number of physical page	Permissions
0	0x00	PTE_U
1	0x01	PTE_P
2	0x02	PTE_P|PTE_W
3	0x03	PTE_P|PTE_W|PTE_U
4	0xFF	PTE_W|PTE_U
5	0xFE	PTE_U
6	0x80	PTE_W
7	0x92	PTE_P|PTE_W|PTE_U
8	0xAB	PTE_P|PTE_W|PTE_U
9	0x09	PTE_P|PTE_U
10	0xFE	PTE_P|PTE_U
11	0x00	PTE_W
12	0x11	PTE_U
All others	(Invalid)	0

For each problem below, write either the physical address of the given virtual address or identify what fault would be produced. The fault types should be one of:

1. Missing page (there is no mapping for the requested page)

2. Privilege violation (user level process trying to access a supervisor page)
3. Permission violation (attempt to write a read-only page)

QUESTION KERN-5A. The kernel dereferences a null pointer

Show solution

> Missing page
>
> Hide solution
>
> Hide solution

QUESTION KERN-5B. A user process dereferences a null pointer

Show solution

> Missing page
>
> Hide solution
>
> Hide solution

QUESTION KERN-5C. The kernel writes to the address 0x8432

Show solution

> 0xAB432
>
> Hide solution
>
> Hide solution

QUESTION KERN-5D. A user process writes to the address 0xB123

Show solution

> Missing page (when both PTE_P and PTE_U are missing, it's PTE_P that counts)
>
> Hide solution
>
> Hide solution

QUESTION KERN-5E. The kernel reads from the address 0x9876

Show solution

> 0x09876
>
> Hide solution
>
> Hide solution

QUESTION KERN-5F. A user process reads from the address 0x7654

Show solution

> 0x92654
>
> Hide solution
>
> Hide solution

QUESTION KERN-5G. A user process writes to the address 0xABCD

Show solution

> Permission violation
>
> Hide solution
>
> Hide solution

QUESTION KERN-5H. A user process writes to the address 0x2321

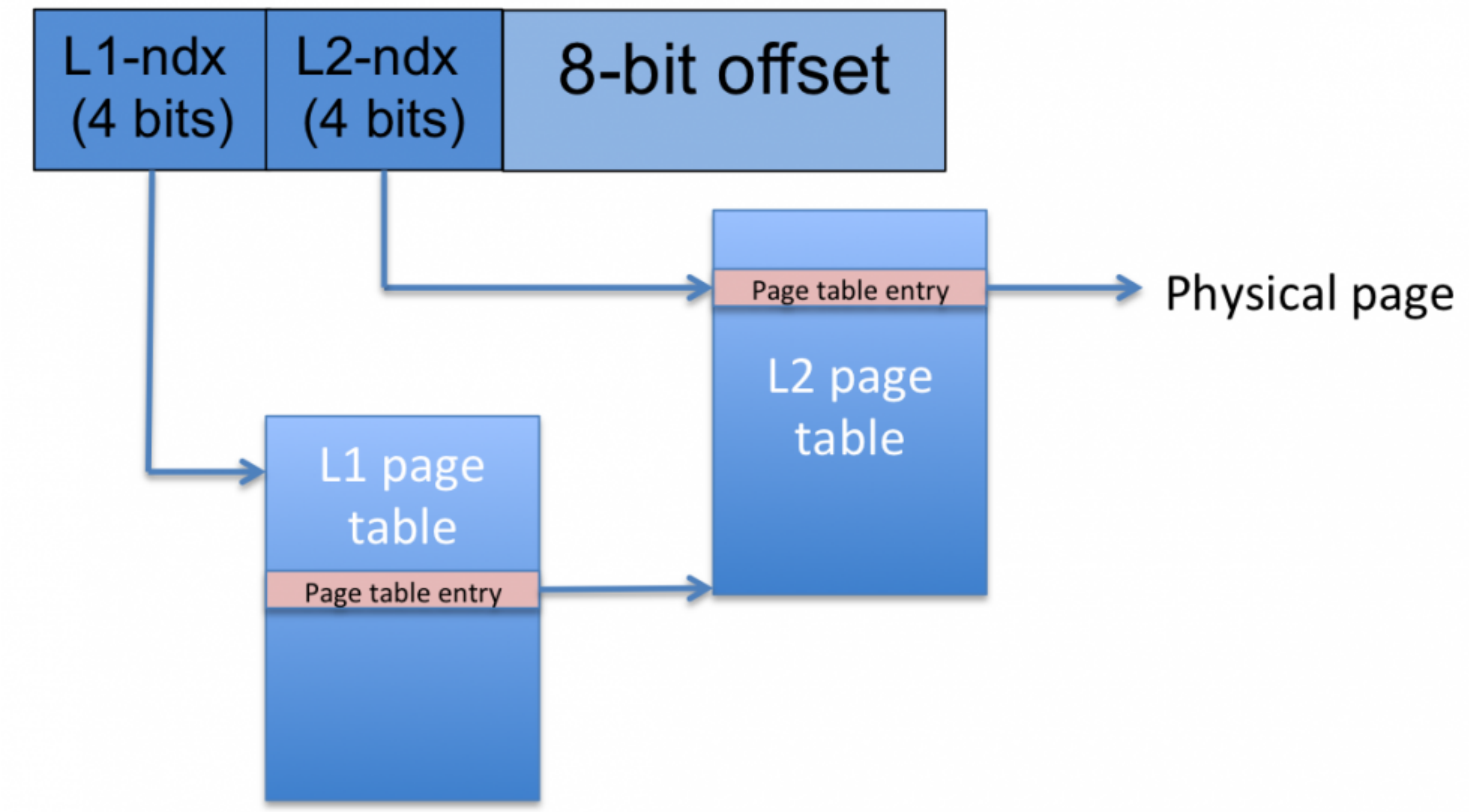
Show solution

> Privilege violation
>
> Hide solution
>
> Hide solution

KERN-6. Virtual Memory

You may recall that Professor Seltzer loves inventing strange and wonderful virtual memory systems—she's at it again! The Tom and Ginny (TAG) processor has 16-bit virtual addresses and 256-byte pages. Virtual memory translation is provided via two-level page tables as shown in the figure below.

⚠️ *Please note that this question uses the opposite convention for page table level numbering than we use for x86-64.*



QUESTION KERN-6A. How many entries are in an L1 page table?

Show solution

16

Hide solution

Hide solution

QUESTION KERN-6B. How many entries are in an L2 page table?

Show solution

16

Hide solution

Hide solution

QUESTION KERN-6C. If each page table entry occupies 2 bytes of memory, how large (in bytes) is a single page table?

Show solution

One good answer is 32 bytes (= 2 * 16). 256 also makes sense—page tables start on page boundaries on most architectures, and TAG pages are 256 pages long—but most of that space would be unused for page table data.

Hide solution

Hide solution

QUESTION KERN-6D. What is the maximum number of L1 page tables that a process can have?

Show solution

1

Hide solution

Hide solution

QUESTION KERN-6E. What is the maximum number of L2 page tables that a process can have?

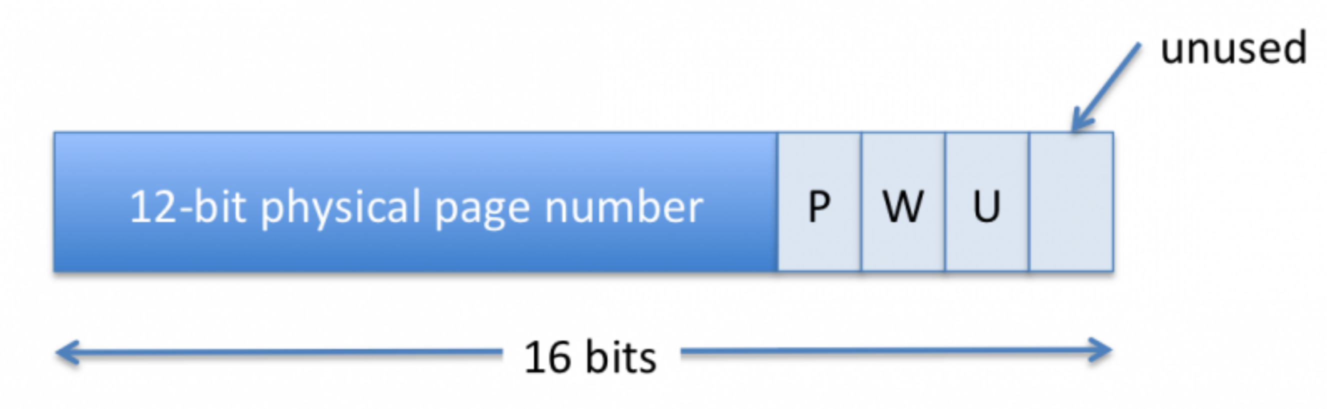
Show solution

16

Hide solution

Hide solution

QUESTION KERN-6F. The Figure below shows how the PTEs are organized.



Given the number of bits allocated to the physical page number in the PTE, how much physical memory can the TAG processor support?

Show solution

12 bits of page number plus 8 bits of page offset is 20 bits, which is 1 MB.

Hide solution

Hide solution

QUESTION KERN-6G. Finally, you'll actually perform virtual address translation in software. We will define a TAG page table entry as follows:

```
typedef unsigned short tag_pageentry;
```

Write a function `unsigned virtual_to_physical(tag_pageentry* pagetable, unsigned vaddr)` that takes as arguments:

- `pagetable`: a TAG page table (that is, a pointer to the first entry in the L1 page table)
- `vaddr`: a TAG virtual address

and returns a physical address if a valid mapping exists and an invalid physical address if no valid mapping exists. Comment your code to explain each step that you want the function to take. You may assume that this function runs with an identity-mapped page table (i.e., each virtual address maps to the physical address with the same numeric value), and that all page tables are accessible.

Show solution

```c
#define PTE_SHIFT       4
#define PT_NDX_MASK     0xF
#define PAGE_SHIFT      8
#define PAGE_MASK       0xFF
#define L1_SHIFT        12
#define PTE_U   2
#define PTE_W   4
#define PTE_P   8

typedef unsigned short* tag_pagetable;
typedef unsigned short pte_t;

unsigned virtual_to_physical(tag_pagetable pagetable, unsigned vaddr) {
    // grab the PTE for the L2 page table
    pte_t pte = pagetable[(vaddr >> L1_SHIFT) & PT_NDX_MASK];
    if ((pte & PTE_P) == 0) {
        return INVALID_PHYSADDR;
    }

    // Calculate L2 page table
    unsigned l2_pt = (unsigned) (pte >> PTE_SHIFT);  // lose permission bits
    l2_pt <<= PAGE_SHIFT;                         // Put page offset in
    tag_pagetable l2_pagetable = (tag_pagetable) l2_pt;

    // Now grab pte from L2 page table
    pte = l2_pagetable[(vaddr >> PAGE_SHIFT) & PT_NDX_MASK];
    if ((pte & PTE_P) == 0) {
        return INVALID_PHYSADDR;
    }

    unsigned physaddr = (unsigned) (pte >> PTE_SHIFT); // lose permissions
    physaddr <<= PAGE_SHIFT;
    physaddr |= vaddr & PAGE_MASK;

    return physaddr;
}
```

Hide solution

Hide solution

KERN-7. Cost expressions

In the following questions, you will reason about the abstract costs of various operations, using the following tables of constants.

Table of Basic Costs

S	System call overhead (i.e., entering and exiting the kernel)
F	Page fault cost (i.e., entering and exiting the kernel)
P	Cost of allocating a new physical page
M	Cost of installing a new page mapping
B	Cost of copying a byte

Table of Sizes

n_k		Number of memory pages allocated to the kernel
n_p		Number of memory pages allocated to process p
r_p		Number of read-only memory pages allocated to process p
w_p	$= n_p - r_p$	Number of writable memory pages allocated to process p
m_p		Number of memory pages actually modified by process p after its previous `fork()`

QUESTION KERN-7A. Our tiny operating systems' processes start out with a single stack page each. A recursive function can cause the stack pointer to move beyond this page, and the program to crash.

This problem can be solved in the process itself. The process can examine its stack pointer before calling a recursive function and call `sys_page_alloc` to map a new stack page when necessary.

Write an expression for the cost of this `sys_page_alloc()` system call in terms of the constants above.

Show solution

S + P + M

Hide solution

Hide solution

QUESTION KERN-7B. Another solution to the stack overflow issue uses the operating system's page fault handler. When a fault occurs in a process's stack region, the operating system allocates a new page to cover the corresponding address. Write an expression for the cost of such a fault in terms of the constants above.

Show solution

F + P + M

Hide solution

Hide solution

QUESTION KERN-7C. Design a revised version of `sys_page_alloc` that supports batching. Give its signature and describe its behavior.

Show solution

> Example: `sys_page_alloc(void *addr, int npages)`
>
> Hide solution
>
> Hide solution

QUESTION KERN-7D. Write an expression for the cost of a call to your batching allocation API.

Show solution

> Can vary; for this example, S + npages*(P + M)
>
> Hide solution
>
> Hide solution

In the remaining questions, a process *p* calls `fork()`, which creates a child process, *c*.

Assume that the *base* cost of performing a `fork()` system call is Φ. This cost includes the `fork()` system call overhead (*S*), the overhead of allocating a new process, the overhead of allocating a new page directory with kernel mappings, and the overhead of copying registers. But it *does not* include overhead from allocating, copying, or mapping other memory.

QUESTION KERN-7E. Consider the following implementations of `fork()`:

A. **Naive fork**: Copy all process memory.
B. **Eager fork**: Copy all *writable* process memory; share *read-only* process memory, such as code.
C. **Copy-on-write fork**: Initially share *all* memory as read-only. Create writable copies later, on demand, in response to write faults.

Which expression best represents the total cost of the `fork()` system call in process *p*, for each of these fork implementations? *Only consider the system call itself,* not later copy-on-write faults.

(*Note*: Per-process variables, such as n, are defined for each process. So, for example, n_p is the number of pages allocated to the parent process *p*, and n_c is the number of pages allocated to the child process *c*.)

1. Φ
2. $\Phi + n_p \times M$

3. $\Phi + (n_p + w_p) \times M$
4. $\Phi + n_p \times 2^{12} \times (B + F)$
5. $\Phi + n_p \times (2^{12}B + P + M)$
6. $\Phi + n_p \times (P + M)$
7. $\Phi + w_p \times (2^{12}B + P + M)$
8. $\Phi + n_p \times (2^{12}B + P + M) - r_p \times (2^{12}B + P)$
9. $\Phi + n_p \times M + m_c \times (P + F)$
10. $\Phi + n_p \times M + m_c \times (2^{12}B + F + P)$
11. $\Phi + n_p \times M + (m_p{+}m_c) \times (P$
    - $F)$
12. $\Phi + n_p \times M + (m_p{+}m_c) \times (2^{12}B + F + P)$

Show solution

A: #5, B: #8 (good partial credit for #7), C: #2

Hide solution

Hide solution

QUESTION KERN-7F. When would copy-on-write fork be more efficient than eager fork (meaning that the sum of all fork-related overheads, including faults for pages that were copied on write, would be less for copy-on-write fork than eager fork)? Circle the best answer.

1. When $n_p < n_k$.
2. When $w_p \times F < w_p \times (M + P)$.
3. When $m_c \times (F + M + P) < w_p \times (M + P)$.
4. When $(m_p{+}m_c) \times (F + M + P + 2^{12}B) < w_p \times (P + 2^{12}B)$.
5. When $(m_p{+}m_c) \times (F + P + 2^{12}B) < w_p \times (P + M + 2^{12}B)$.
6. When $m_p < m_c$.
7. None of the above.

Show solution

#4

Hide solution

Hide solution

KERN-8. Virtual memory

QUESTION KERN-8A. What kind of address is stored in x86-64 register `%cr3`, virtual or physical?

Show solution

physical

Hide solution

Hide solution

QUESTION KERN-8B. What kind of address is stored in x86-64 register `%rip`, virtual or physical?

Show solution

virtual

Hide solution

Hide solution

QUESTION KERN-8C. What kind of address is stored in an x86-64 page table entry, virtual or physical?

Show solution

physical

Hide solution

Hide solution

QUESTION KERN-8D. What is the x86-64 word size in bits?

Show solution

64

Hide solution

Hide solution

Many paged-virtual-memory architectures can be characterized in terms of the *PLX* constants:

- *P* = the length of the page offset, in bits.
- *L* = the number of different page indexes (equivalently, the number of page table levels).
- *X* = the length of each page index, in bits.

QUESTION KERN-8E. What are the numeric values for *P*, *L*, and *X* for x86-64?

Show solution

P=12, L=4, X=9

Hide solution

Hide solution

Assume for the remaining parts that, as in x86-64, each page table page fits within a single page, and each page table entry holds an address and some flags, including a Present flag.

QUESTION KERN-8F. Write a *PLX* formula for the number of bytes per page, using both mathematical and C notation.

Mathematical notation: ____________________

C notation: ____________________________

Show solution

2^P, `1 << P`

Hide solution

Hide solution

QUESTION KERN-8G. Write a *PLX* formula for the number of meaningful bits in a virtual address.

Show solution

P + L*X

Hide solution

Hide solution

QUESTION KERN-8H. Write a *PLX* formula that is an upper bound on the number of bits in a physical address. (Your upper bound should be relatively tight; $P^{X^{100L}}$ is a bad answer.)

Show solution

8 * `sizeof(entry)` = $8 * 2^P/2^X = 8 * 2^{P-X}$

Hide solution

Hide solution

QUESTION KERN-8I. Write a *PLX* formula for the *minimum* number of pages it would take to store a page table that allows access to 2^X distinct destination physical pages.

Show solution

L (for a well-formed page table with distinct pages for each level), or 1 (for a “trick” page table that reuses the top-level page for all subsequent levels; see question KERN-1C).

Hide solution

Hide solution

KERN-9. Weensy signals

WeensyOS lacks signals. Let’s add them.

⚠️ *Signals are covered in the Shell unit.*

Here’s `sys_kill`, a system call that should deliver a signal to a process.

```
// sys_kill(pid, sig)
//    Send signal `sig` to process `pid`.
inline int sys_kill(pid_t pid, int sig) {
    register uintptr_t rax asm("rax") = SYSCALL_KILL;
    asm volatile ("syscall"
                  : "+a" (rax), "+D" (pid), "+S" (sig)
                  :
                  : "cc", "rcx", "rdx", …, "memory");
    return rax;
}
```

QUESTION KERN-9A. Implement the WeensyOS kernel `syscall` case for `SYSCALL_KILL`. Your implementation should simply change the receiving process’s state to `P_BROKEN`. Check arguments as necessary to avoid kernel isolation violations; return 0 on success and -1 if the receiving process does not exist or is not running. A process may kill itself.

```
uintptr_t syscall(...) { ...
case SYSCALL_KILL:
    // your code here
```

Show solution

```
    uintptr_t pid = current->regs.reg_rdi;
    if (pid == 0 || pid >= NPROC || proc[pid].state != P_RUNNING) {
        return -1;
    } else {
        proc[pid].state = P_BROKEN;
        return 0;
    }
```

Hide solution

Hide solution

The WeensyOS signal handling mechanism is based on that of Unix. When a signal is delivered to a WeensyOS process:

1. The kernel saves the receiving process's registers and switches the process to signal-handling mode.
2. The kernel causes the receiving process to execute its signal handler.
    - It creates a stack frame for the signal handler by subtracting **at least 128 bytes** from the process's stack pointer. This ensures that the signal handler's local variables (if any) do not overwrite those of the interrupted function.
    - It sets the signal handler's arguments. A WeensyOS signal handler takes one argument, the signal number.
    - It sets the process's instruction pointer to the signal handler address and resumes the process.
3. The signal handler can tell the kernel to resume normal processing by calling the `sigreturn` system call. This will restore the registers to their saved values and resume the process.

Implement this. Begin from the following system call definitions and changes to the WeensyOS kernel's `struct proc`.

```
inline int sys_kill(pid_t pid, int sig) { ... } // as above

// sys_signal(sighandler)
//    Set the current process's signal handler to `sighandler`.
inline int sys_signal(void (*sighandler)(int)) {
    register uintptr_t rax asm("rax") = SYSCALL_SIGNAL;
    asm volatile ("syscall"
                  : "+a" (rax), "+D" (sighandler)
                  : ...);
    return rax;
}

// sys_sigreturn()
//    Returns from a signal handler to normal mode. Does nothing if in normal mode
already.
inline int sys_sigreturn() {
    register uintptr_t rax asm("rax") = SYSCALL_SIGRETURN;
    asm volatile ("syscall"
                  : "+a" (rax)
                  : ...);
    return rax;
}

struct proc {
    x86_64_pagetable* pagetable;
    pid_t pid;
    regstate regs;
    procstate_t state;

    // Signal support:
    uintptr_t sighandler;         // signal handler (0 means default)
    bool sigmode;                 // true iff in signal-handling mode
    regstate saved_regs;          // saved registers, if in signal-handling mode
};
```

QUESTION KERN-9B. Implement the WeensyOS kernel `syscall` case for `SYSCALL_SIGNAL`.

```
uintptr_t syscall(...) { ...
case SYSCALL_SIGNAL:
    // your code here (< 5 lines)
```

Show solution

```
    current->sighandler = current->regs.reg_rdi;
    return 0;
```

Hide solution

Hide solution

QUESTION KERN-9C. Implement the WeensyOS kernel `syscall` case for `SYSCALL_SIGRETURN`. If the current process is in signal-handling mode (`current->p_sigmode != 0`), restore the saved registers and leave signal-handling mode; otherwise simply return 0.

```
void syscall(...) { ...
case SYSCALL_SIGRETURN:
    // your code here (< 10 lines)
```

Show solution

```
    if (current->sigmode) {
        current->regs = current->saved_regs;
        current->sigmode = false;
        run(current);
        // not as correct as `run(current)`, but OK:
        // return current->regs.reg_rax;
    } else {
        return 0;
    }
```

Note that it is important to call `run(current)` in the `sigmode` case, since the normal `syscall` return path will not restore all registers.

Hide solution

Hide solution

QUESTION KERN-9D. Implement the WeensyOS kernel `syscall` case for `SYSCALL_KILL`. If the receiving process's `sighandler` is 0, behave as in part A. Otherwise, if the receiving process is in signal-handling mode, return -1 to the calling process rather than delivering the signal. Otherwise, save the receiving process's registers and cause it to call its signal handler in signal-handling mode, as described above.

```
uintptr_t syscall(...) { ...
case SYSCALL_KILL:
    // your code here (< 25 lines)
```

Show solution

```
    uintptr_t pid = current->regs.reg_rdi;
    int sig = current->regs.reg_rsi;
    if (pid == 0 || pid >= NPROC || proc[pid].state != P_RUNNING
        || proc[pid].sigmode) {
        return -1;
    } else if (proc[pid].sighandler == 0) {    // default signal handler: kill
process
        proc[pid].state = P_BROKEN;
        return 0;
    } else {
        proc[pid].saved_regs = proc[pid].regs;
        proc[pid].regs.reg_rsp -= 128; // ignore alignment
        proc[pid].regs.reg_rdi = sig;
        proc[pid].regs.reg_rip = proc[pid].p_sighandler;
        if (pid == current->pid) {
            // special case for killing self
            proc[pid].saved_regs.reg_rax = 0;   // after signal handler runs,
`kill` returns 0
            run(current);
        } else {
            return 0;
        }
    }
    break;
```

Hide solution

Hide solution

QUESTION KERN-9E. Unix has some signals that cannot be caught or handled, especially `SIGKILL` (signal 9), which unconditionally exits a process. Which kernel and/or process code would change to support equivalent functionality in WeensyOS? List all that apply.

Show solution

Just the `SYSCALL_KILL` case in `syscall`.

Hide solution

Hide solution

QUESTION KERN-9F. Is it necessary to verify the signal handler address to avoid kernel-isolation violations? Explain briefly why or why not.

Show solution

No, because that address is only accessed in unprivileged mode.

Hide solution

Hide solution

QUESTION KERN-9G. BONUS QUESTION. A WeensyOS signal handler function must end with a call to `sys_sigreturn()`. Describe how the WeensyOS kernel could set it up so that `sys_sigreturn()` is called *automatically* when a signal-handler function returns.

Show solution

Write instructions to the stack, after the 128-byte region, that implement `sys_sigreturn()`. Then push the address of those instructions—that will be the return address.

Hide solution

Hide solution

KERN-10. Weensy threads

⚠️ *Threads are covered in the Synchronization unit.*

Betsy Ross is changing her WeensyOS to support threads. There are many ways to implement threads, but Betsy wants to implement threads using the `ptable` array. "After all," she says, "a thread is just like a process, except it shares memory with some other process!"

Betsy has defined a new system call, `sys_create_thread`, that starts a new thread running a given thread function, with a given argument, and a given stack pointer:

```
typedef void* (*thread_function)(void*);
pid_t sys_create_thread(thread_function f, void* arg, void* stack_top);
```

The system call's return value is the ID of the new thread.

Betsy's kernel contains the following code for her `sys_fork` implementation.

```
// in syscall()
case SYSCALL_FORK:
    return handle_fork(current);
...

uint64_t handle_fork(proc* p) {
    proc* new_p = find_unused_process();
    if (!new_p)
        return -1;

    new_p->pagetable = copy_pagetable(p->pagetable);
    if (!new_p->pagetable)
        return -1;

    new_p->regs = p->regs;
    new_p->regs.reg_rax = 0;
    new_p->state = P_RUNNABLE;
    return 0;
}
```

And here's the start of her `sys_create_thread` implementation.

```
// in syscall()
case SYSCALL_CREATE_THREAD:
    return handle_create_thread(current);
...

uint64_t handle_create_thread(proc* p) {
    // Whoops! Got a revolution to run, back later
    return -1;
}
```

QUESTION KERN-10A. Complete her `handle_create_thread` implementation. Assume for now that the thread function never exits. You may use these helper functions if you need them (you may not):

- `proc* find_unused_process()`
  Return a usable `proc*` that has state `P_FREE`, or `nullptr` if no unused process exists.
- `x86_64_pagetable* copy_pagetable(x86_64_pagetable* pgtbl)`
  Return a copy of pagetable `pgtbl`, with all unprivileged writable pages copied. Returns `nullptr` if any allocation fails.
- `void* kalloc(size_t)`
  Allocates a new physical page, zeros it, and returns its physical address.

Recall that system call arguments are passed according to the x86-64 calling convention: first argument in `%rdi`, second in `%rsi`, third in `%rdx`, etc.

Show solution

```
uint64_t handle_create_thread(proc* p) {
    proc* np = find_unused_process();
    if (!np) {
        return (uint64_t) -1;
    }
    np->regs = p->regs;
    np->regs.reg_rip = p->regs.reg_rdi;
    np->regs.reg_rdi = p->regs.reg_rsi;
    np->regs.reg_rsp = p->regs.reg_rdx;
    np->state = P_RUNNABLE;
    np->pagetable = p->pagetable;
    return np;
}
```

Hide solution

Hide solution

QUESTION KERN-10B. Betsy's friend Prince Dimitri Galitzin thinks Betsy should give processes even more flexibility. He suggests that `sys_create_thread` take a full set of registers, rather than just a new instruction pointer and a new stack pointer. That way, the creating thread can supply *all* registers to the new thread, rather than just a single argument.

```
pid_t sys_create_thread(x86_64_registers* new_registers);
```

The kernel will simply copy `*new_registers` into the `proc` structure for the new thread. Easy!

Which of the following properties of `x86_64_registers` would allow Dimitri's plan to violate kernel isolation? List all that apply.

1. `reg_rax` contains the thread's `%rax` register.
2. `reg_rip` contains the thread's instruction pointer.
3. `reg_cs` contains the thread's privilege level, which is 3 for unprivileged.
4. `reg_intno` contains the number of the last interrupt the thread caused.
5. `reg_rflags` contains the `EFLAGS_IF` flag, which indicates that the thread runs with interrupts enabled.
6. `reg_rsp` contains the thread's stack pointer.

Show solution

#3, #5 only, though it is OK to list #4.

Hide solution

Hide solution

Now Betsy wants to handle thread exit. She introduces two new system calls, `sys_exit_thread` and `sys_join_thread`:

```
void sys_exit_thread(void* exit_value);
void* sys_join_thread(pid_t thread);
```

`sys_exit_thread` causes the thread to exit with the given exit value; it does not return. `sys_join_thread` behaves like `pthread_join` or `waitpid`. If `thread` corresponds is a thread of the same process, and `thread` has exited, `sys_join_thread` cleans up the thread and returns its exit value; otherwise, `sys_join_thread` returns `(void*) -1`.

QUESTION KERN-10C. Is the `sys_join_thread` specification blocking or polling?

Show solution

Polling

Hide solution

Hide solution

QUESTION KERN-10D. Betsy makes the following changes to WeensyOS internal structures to support thread exit.

1. She adds a `void* exit_value` member to `struct proc`.
2. She adds a new process state, `P_EXITED`, that corresponds to exited threads.

Complete the case for `SYSCALL_EXIT_THREAD` in `syscall()`. Don't worry about the case where the last thread in a process calls `sys_exit_thread` instead of `sys_exit`.

```
case SYSCALL_EXIT_THREAD:
```

Show solution

```
    current->state = P_EXITED;
    current->exit_value = p->regs.reg_rdi;
    schedule();
```

Note that we don't even need `exit_value`, since we could use `regs.reg_rdi` to look up the exit value elsewhere!

If you wanted to clean up the last thread in a process, you might do something like this:

```
    int nlive = 0;
    for (int i = 1; i < NPROC && !nlive; ++i) {
        if (is_thread_in(i, current) && processes[i].state != P_EXITED) {
            ++nlive;
        }
    }
    if (!nlive) {
        for (int i = 1; i < NPROC; ++i) {
            if (is_thread_in(i, current)) {
                do_sys_exit(&processes[i]);
            }
        }
    }
```

Hide solution

Hide solution

QUESTION KERN-10E. Complete the following helper function.

```
// Test whether `test_pid` is the PID of a thread in the same process as `p`.
// Return 1 if it is; return 0 if `test_pid` is an illegal PID, it corresponds to
// a freed process, or it corresponds to a thread in a different process.
int is_thread_in(pid_t test_pid, proc* p) {
```

Show solution

```
    return test_pid >= 0 && test_pid < NPROC && processes[test_pid].state !=
P_FREE
        && processes[test_pid].pagetable == p->pagetable;
```

Hide solution

Hide solution

QUESTION KERN-10F. Complete the case for `SYSCALL_JOIN_THREAD` in `syscall()`. Remember that a thread may be successfully joined at most once: after it is joined, its PID is made available for reallocation.

```
case SYSCALL_JOIN_THREAD:
```

Show solution

```
    int pid = current->regs.reg_rdi;
    if (pid != current->regs.reg_rdi
        || !is_thread_in(pid, current)
        || processes[pid].state != P_EXITED) {
        return -1;
    } else {
        processes[pid].state = P_FREE;
        return processes[pid].exit_value;
    }
```

Note that we can't distinguish a –1 return value from error from a –1 return value from `sys_exit_thread(-1)`.

Hide solution

Hide solution

QUESTION KERN-10G. In pthreads, a thread can exit by returning from its thread function; the return value is used as an exit value. So far, that's not true in Weensy threads: a thread returning from its thread function will execute random code, depending on what random garbage was stored in its initial stack in the return address position. But Betsy thinks she can implement pthread-style behavior entirely at user level, with two changes:

1. She'll write a *two-instruction function* called `thread_exit_vector`.
2. Her `create_thread` library function will write a *single 8-byte value* to the thread's new stack before calling `sys_create_thread`.

Explain how this will work. What instructions will `thread_exit_vector` contain? What 8-byte value will `create_thread` write to the thread's new stack? And where will that value be written relative to `sys_create_thread`'s `stack_top` argument?

Show solution

```
thread_exit_vector:
    movq %rax, %rdi
    jmp sys_exit_thread
```

The 8-byte value will equal the address of `thread_exit_vector`, and it will be placed in the return address slot of the thread's new stack. So it will be written starting at address `stack_top`.

Hide solution

Hide solution

Shell exercises

Exercises that seem less appropriate this year, or which cover topics that we haven't covered in class, should be marked with ⚠️. However, we may have missed some.

See also question KERN-9.

SH-1. Rendezvous and pipes

This question builds versions of the existing system calls based on new abstractions. Here are three system calls that define a new abstraction called a rendezvous.

int newrendezvous()
Returns a rendezvous ID that hasn't been used yet.

int rendezvous(int *rid*, int *data*)
Blocks the calling process P1 until some other process P2 calls rendezvous() with the same *rid* (rendezvous ID). Then, both of the system calls return, but P1's system call returns P2's *data* and vice versa. Thus, the two processes swap their *data*. Rendezvous acts pairwise; if three processes call rendezvous, then two of them will swap values and the third will block, waiting for a fourth.

void freezerendezvous(int *rid*, int *freezedata*)
Freezes the rendezvous *rid*. All future calls to rendezvous(*rid*, *data*) will immediately return *freezedata*.

Here's an example. The two columns represent two processes. Assume they are the only processes using rendezvous ID 0.

`int result = rendezvous(0, 5);`	`printf("About to rendezvous\n");`
	`int result = rendezvous(0, 600);`
/* The processes swap data;	both become runnable */
`printf("Process A got %d\n", result);`	`printf("Process B got %d\n", result);`

This code will print

```
About to rendezvous
Process B got 5
Process A got 600
```

(the last 2 lines might appear in either order).

QUESTION SH-1A. How might you implement pipes in terms of rendezvous? Try to figure out analogues for the pipe(), close(), read(), and write() system calls (perhaps with different signatures), but *only worry about reading and writing 1 character at a time.*

Show solution

Here's one mapping.

- `pipe()`: `newrendezvous()`. We use a rendezvous ID as the equivalent of a pipe file descriptor.
- `close(p)`: To close the "pipe" `p`, call `freezerendezvous(p, -1)`. Now all future `read` and `write` calls will return -1.
- `read(p, &ch, 1)`: To read a single character `ch` from the "pipe" `p`, call `ch = rendezvous(p, -1)`.
- `write(p, &ch, 1)`: To write a single character `ch` to the "pipe" `p` (that is, the rendezvous with ID `p`), call `rendezvous(p, ch)`.

Most mappings will have these features.

Hide solution

Hide solution

QUESTION SH-1B. Can a rendezvous-pipe support all pipe features?

Show solution

No. For example, a rendezvous-pipe doesn't deliver a signal when a process tries to write to a closed pipe. Since the rendezvous-pipe doesn't distinguish between read and write ends, and since rendezvous aren't reference-counted like file descriptors, if a "writer" process exits without closing the rendezvous-pipe, a reader won't get EOF when they read—it will instead block indefinitely. Unlike pipes, which like all file descriptors are protected from access by unrelated processes, rendezvous aren't protected; anyone who can guess the rendezvous ID can use the rendezvous. Etc.

Hide solution

Hide solution

SH-2. Process management

Here's the skeleton of a shell function implementing a simple two-command pipeline, such as "`cmd1 | cmd2`".

```
void simple_pipe(const char* cmd1, char* const* argv1, const char* cmd2, char* const*
argv2) {
    int pipefd[2], r, status;

    [A]

    pid_t child1 = fork();
    if (child1 == 0) {
        [B]
        execvp(cmd1, argv1);
    }
    assert(child1 > 0);

    [C]

    pid_t child2 = fork();
    if (child2 == 0) {
        [D]
        execvp(cmd2, argv2);
    }
    assert(child2 > 0);

    [E]
}
```

And here is a grab bag of system calls.

```
[1] close(pipefd[0]);
[2] close(pipefd[1]);
[3] dup2(pipefd[0], STDIN_FILENO);
[4] dup2(pipefd[0], STDOUT_FILENO);
[5] dup2(pipefd[1], STDIN_FILENO);
[6] dup2(pipefd[1], STDOUT_FILENO);
[7] pipe(pipefd);
[8] r = waitpid(child1, &status, 0);
[9] r = waitpid(child2, &status, 0);
```

Your task is to assign system call IDs, such as "`1`", to slots, such as "`A`", to achieve several behaviors, including a correct pipeline and several incorrect pipelines. For each question:

- You may use each system call ID once, more than once, or not at all.
- You may use zero or more system call IDs per slot. Write them in the order they should appear in the code.
- You may assume that no signals are delivered to the shell process (so no system call ever returns an `EINTR` error).
- The `simple_pipe` function should wait for **both** commands in the pipeline to complete before returning.

QUESTION SH-2A. Implement a correct foreground pipeline.

[A]	[B] (child1)	[C]	[D] (child2)	[E]

Show solution

[A]	[B]	[C]	[D]	[E]
7	6, 1, 2 *or* 6, 2, 1 *or* 1, 6, 2		3, 1, 2	1, 2, 9 *or* 2, 1, 9 *etc.* (1, 2 come first)

or

[A]	[B]	[C]	[D]	[E]
7	6, 1, 2 *or* 6, 2, 1 *or* 1, 6, 2	2	3, 1	1, 9 *or* 1, 9

Hide solution

Hide solution

QUESTION SH-2B. Implement a pipeline so that, given arguments corresponding to “`echo foo | wc -c`”, the `wc` process reads “`foo`” from its standard input but does not exit thereafter. **For partial credit** describe in words how this might happen.

[A]	[B] (child1)	[C]	[D] (child2)	[E]

Show solution

Anything that doesn’t close the pipe’s write end will do it. Below we leave both ends of the pipe open in the shell. We could also enter just “3” in slot [D].

[A]	[B]	[C]	[D]	[E]
7	6, 1, 2		3, 1, 2	8, 9

Hide solution

Hide solution

QUESTION SH-2C. Implement a pipeline so that, given arguments corresponding to "`echo foo | wc -c`", "`foo`" is printed to the *shell*'s standard output and the `wc` process prints "`0`". (In a correctly implemented pipeline, "`wc`" would print `4`, which is the number of characters in "`foo\n`".) **For partial credit** describe in words how this might happen.

`[A]`	`[B] (child1)`	`[C]`	`[D] (child2)`	`[E]`

Show solution

Anything that doesn't redirect the left-hand side's standard output will do it. It is important that the *read* end of the pipe be properly redirected, or `wc` would block reading from the *shell's* standard input.

`[A]`	`[B]`	`[C]`	`[D]`	`[E]`
7	1, 2 (anything without 6)		3, 1, 2	1, 2, 8, 9

Hide solution

Hide solution

QUESTION SH-2D. Implement a pipeline that appears to work correctly on "`echo foo | wc -c`", but always blocks forever if the left-hand command outputs more than 65536 characters. **For partial credit** describe in words how this might happen.

`[A]`	`[B] (child1)`	`[C]`	`[D] (child2)`	`[E]`

Show solution

This happens when we execute the two sides of the pipe in series: first the left-hand side, then the right-hand side. Since the pipe contains 64KiB of buffering, this will often appear to work for left-hand sides that emit relatively few characters.

`[A]`	`[B]`	`[C]`	`[D]`	`[E]`
7	6, 1, 2	8	3, 1, 2	1, 2, 9

Hide solution

Hide solution

QUESTION SH-2E. Implement a pipeline so that, given arguments corresponding to "`echo foo | wc -c`", both `echo` and `wc` report a "Bad file descriptor" error. (This error, which corresponds to `EBADF`, is returned when a file descriptor is not valid or does not support the requested operation.) **For partial credit** describe in words how this might happen.

[A]	[B] (child1)	[C]	[D] (child2)	[E]

Show solution

Given these system calls, the only way to make this happen is to redirect the wrong ends of the pipe into stdin/stdout.

[A]	[B]	[C]	[D]	[E]
7	4, 1, 2		5, 1, 2	1, 2, 8, 9

Hide solution

Hide solution

SH-3. Processes

Consider the two programs shown below.

```
// Program 1
#include <cstdio>
#include <unistd.h>
int main() {
    printf("PID %d running prog1\n", getpid());
}
```

```
// Program 2
#include <cstdio>
#include <unistd.h>
int main() {
    char* argv[2];
    argv[0] = (char*) "prog1";
    argv[1] = nullptr;
    printf("PID %d running prog2\n", getpid());
    int r = execv("./prog1", argv);
    printf("PID %d exiting from prog2\n", getpid());
}
```

QUESTION SH-3A. How many different PIDs will print out if you run Program 2?

Show solution

One. `exec` doesn't change the process's PID.

Hide solution

Hide solution

QUESTION SH-3B. How many lines of output will you see?

Show solution

Two, as follows:

```
PID xxx running prog2
PID xxx running prog1
```

Hide solution

Hide solution

Now, let's assume that we change Program 2 to the following:

```
// Program 2B
#include <cstdio>
#include <unistd.h>
int main() {
    char* argv[2];
    argv[0] = (char*) "prog1";
    argv[1] = nullptr;

    printf("PID %d running prog2\n", getpid());
    pid_t p = fork();
    if (p == 0) {
        int r = execv("./prog1", argv);
    } else {
        printf("PID %d exiting from prog2\n", getpid());
    }
}
```

QUESTION SH-3C. How many different PIDs will print out if you run Program 2B?

Show solution

Two: the child has a different PID than the parent.

Hide solution

Hide solution

QUESTION SH-3D. How many lines of output will you see?

Show solution

Three, as follows:

```
PID xxx running prog2
PID yyy running prog1
PID xxx exiting from prog2
```

Hide solution

Hide solution

Finally, consider this version of Program 2.

```
// Program 2C
#include <cstdio>
#include <unistd.h>
int main() {
    char* argv[2];
    argv[0] = (char*) "prog1";
    argv[1] = nullptr;

    printf("PID %d running prog2\n", getpid());
    pid_t p = fork();
    pid_t q = fork();
    if (p == 0 || q == 0) {
        int r = execv("./prog1", argv);
    } else {
        printf("PID %d exiting from prog2\n", getpid());
    }
}
```

QUESTION SH-3E. How many different PIDs will print out if you run Program 2C?

Show solution

Four!

1. The initial `./prog2` prints its PID.
2. It forks once.
3. Both the initial `./prog2` and its forked child fork again, for four total processes.
4. All processes except the initial `./prog2` exec `./prog1`, which print their distinct PIDs.

Hide solution

Hide solution

QUESTION SH-3F. How many lines of output will you see?

Show solution

Five.

Hide solution

Hide solution

SH-4. Be a CS61 TF!

You are a CS61 teaching fellow. A student working on A4 is having difficulty getting pipes working. S/he comes to you for assistance. The function below is intended to traverse a linked list of commands, fork/exec the indicated processes, and hook up the pipes between commands correctly. The student has commented it reasonably, but is quite confused about how to finish writing the code. Can you help? Figure out what code to add at points A, B, and C.

```
#include "sh61.hh"

struct command {
    command *next; // Next in sequence of commands
    int argc;      // number of arguments
    int ispipe;    // pipe symbol follows this command
    char** argv;   // arguments, terminated by NULL
    pid_t pid;     // pid running this command
};

void do_pipes(command* c) {
    pid_t newpid;
    int havepipe = 0;   // We had a pipe on the previous command
    int lastpipe[2] = {-1, -1};
    int curpipe[2];
```

```
    do {
        if (c->ispipe) {
            int r = pipe(curpipe);
            assert(r == 0);
        }

        newpid = fork();
        assert(newpid >= 0);
        if (newpid == 0) {
            if (havepipe) {
                // There was a pipe on the last command; It's stored
                // in lastpipe; I need to hook it up to this process???
                // **** PART A ****
            }
            if (c->ispipe) {
                // The current command is a pipe -- how do I hook it up???
                // **** PART B ****
            }

            execvp(c->argv[0], c->argv);

            fprintf(stderr, "Exec failed\n");
            _exit(1);
        }

        // I bet there is some cleanup I have to do here!?
        // **** PART C ****

        // Set up for the next command
        havepipe = c->ispipe;
        if (c->ispipe) {
            lastpipe[0] = curpipe[0];
            lastpipe[1] = curpipe[1];
        }
        c->pid = newpid;
        c = c->next;
    } while (newpid != -1 && havepipe);
}
```

QUESTION SH-4A. What should go in the Part A space above, if anything?

Show solution

```
close(lastpipe[1]);
dup2(lastpipe[0], STDIN_FILENO);
close(lastpipe[0]);
```

Hide solution

Hide solution

QUESTION SH-4B. What should go in the Part B space above, if anything?

Show solution

```
close(curpipe[0]);
dup2(curpipe[1], STDOUT_FILENO);
close(curpipe[1]);
```

Hide solution

Hide solution

QUESTION SH-4C. What should go in the Part C space above, if anything?

Show solution

```
if (havepipe) {
    close(lastpipe[0]);
    close(lastpipe[1]);
}
```

Hide solution

Hide solution

SH-5. Spork

Patty Posix has an idea for a new system call, `spork`. Her system call combines `fork`, file descriptor manipulations, and `execvp`. It's pretty cool:

```
struct spork_file_action_t {
    int type; // equals SPORK_OPEN, SPORK_CLOSE, or SPORK_DUP2
    int fd;
    int old_fd;            // SPORK_DUP2 only
    const char* filename;  // SPORK_OPEN only
    int flags;             // SPORK_OPEN only
    mode_t mode;           // SPORK_OPEN only
};

pid_t spork(const char* file, const spork_file_action_t* file_actions, int
n_file_actions, char* argv[]);
```

Here's how `spork` works.

1. First, `spork` forks a child process.
2. The child process loops over the `file_actions` array (there are `n_file_actions` elements) and performs each file action in turn. A file action `fa` means different things depending on its type. Specifically:
    - `fa->type == SPORK_OPEN`: The child process opens the file named `fa->filename` with flags `fa->flags` and optional mode `fa->mode`, as if by `open(fa->filename, fa->flags, fa->mode)`. The opened file descriptor is given number `fa->fd`. (Note that this requires multiple steps, since the file must be first opened and then moved to `fa->fd`.)
    - `fa->type == SPORK_CLOSE`: The child process closes file descriptor `fa->fd`.
    - `fa->type == SPORK_DUP2`: The child process makes `fa->fd` a duplicate of `fa->old_fd`.
3. Finally, the child process execs the given `file` with argument list `argv`.
4. If all these steps succeed, then `spork` returns the child process ID. If any of the steps fails, then either `spork` returns –1 and creates no child, or the child process exits with status 127. In particular, if a file action fails, then the child process exits with status 127 (and does not call `exec`).

This function uses `spork` to print the number of words in a file to standard output.

```
void print_word_count(const char* file) {
    spork_file_action_t file_actions[1];
    file_actions[0].type = SPORK_OPEN;
    file_actions[0].fd = STDIN_FILENO;
    file_actions[0].filename = file;
    file_actions[0].flags = O_RDONLY;
    const char* argv[2] = {"wc", nullptr};
    pid_t p = spork("wc", file_actions, 1, argv);
    assert(p >= 0);
    waitpid(p, NULL, 0);
}
```

QUESTION SH-5A. Use `spork` to implement the following function.

```c
// Create a pipeline like `argv1 | argv2`.
// The pipeline consists of two child processes, one running the command with argument
// list `argv1` and one running the command with argument list `argv2`. The standard
// output of `argv1` is piped to the standard input of `argv2`.
// Return the PID of the `argv2` process or -1 on failure.
pid_t make_pipeline(char* argv1[], char* argv2[]);
```

Show solution

```c
pid_t make_pipeline(char* argv1[], char* argv2[]) {
    int pipefd[2];
    if (pipe(pipefd) < 0) {
        return -1;
    }
    spork_file_actions_t fact[3];
    fact[0].type = SPORK_DUP2;
    fact[0].fd = STDOUT_FILENO;
    fact[0].old_fd = pipefd[1];
    fact[1].type = SPORK_CLOSE;
    fact[1].fd = pipefd[0];
    fact[2].type = SPORK_CLOSE;
    fact[2].fd = pipefd[1];
    if (spork(argv1[0], fact, 3, argv1) < 0) {
        // this is optional:
        close(pipefd[0]);
        close(pipefd[1]);
        return -1;
    }
    close(pipefd[1]);
    fact[0].fd = STDIN_FILENO;
    fact[0].old_fd = pipefd[0];
    // fact[1] is already set up
    pid_t p = spork(argv2[0], fact, 2, argv2);
    close(pipefd[0]);
    return p;
}
```

Hide solution

Hide solution

QUESTION SH-5B. Now, *implement* `spork` in terms of system calls you already know. For full credit, make sure you catch all errors. Be careful of `SPORK_OPEN`.

Show solution

```c
pid_t spork(const char* file, const spork_file_action_t* fact, int nfact, char*
argv[]) {
    pid_t p = fork();
    if (p == 0) {
        for (int i = 0; i < nfact; ++i) {
            switch (fact[i].type) {
            case SPORK_OPEN: {
                int fd = open(fact[i].filename, fact[i].flags, fact[i].mode);
                if (fd < 0) {
                    _exit(127);
                }
                if (fd != fact[i].fd) {
                    if (dup2(fd, fact[i].fd) < 0
                        || close(fd) < 0) {
                        _exit(127);
                    }
                }
                break;
            }
            case SPORK_DUP2:
                if (dup2(fact[i].old_fd, fact[i].fd) < 0) {
                    _exit(127);
                }
                break;
            case SPORK_CLOSE:
                if (close(fact[i].fd) < 0) {
                    _exit(127);
                }
                break;
            default:
                _exit(127);
            }
        }
        execvp(file, argv);
        _exit(127);
    }
    return p;
}
```

Errors that don't need to be handled: `close`, `close` in SPORK_OPEN case, `type` is unknown (we didn't say what to do in that case).

Hide solution

Hide solution

QUESTION SH-5C. Can `fork` be implemented in terms of `spork`? Why or why not?

Show solution

> No, it can't, because `fork` makes a copy of the process's *current* memory state and file descriptor table, while `spork` always calls `execvp` (which creates a fresh process) or exits.
>
> Hide solution
>
> Hide solution

QUESTION SH-5D. At least one of the file action types is *redundant*, meaning a `spork` caller could simulate its behavior using the other action types and possibly some additional system calls. Say which action types are redundant, and briefly describe how they could be simulated.

Show solution

> `SPORK_OPEN` is redundant: it can be implemented by running `open` in the parent (before calling `spork`), creating a new actions list with a `SPORK_DUP2`/`SPORK_CLOSE` pair (to `dup2` the opened fd into place and then `close` the opened fd), calling `spork` with that new actions list, and then `close`ing the opened fds in the parent.
>
> `SPORK_CLOSE` is also redundant, because you could set the close-on-exec bit in the parent. However, you cannot fully simulate an arbitrary file-actions list using just close-on-exec, because a `SPORK_CLOSE` can cause a later file action to deterministically fail before the `exec`.
>
> Hide solution
>
> Hide solution

SH-6. File descriptor facts

Here are twelve file descriptor-oriented system calls.

`accept`	`bind`	`close`	`connect`	`dup2`	`listen`
`open`	`pipe`	`read`	`select`	`socket`	`write`

⚠️ *The `accept`, `bind`, `connect`, `listen`, and `socket` system calls are covered in the synchronization unit.*

QUESTION SH-6A. Which of these system calls may cause the number of open file descriptors to increase? List all that apply.

Show solution

`accept`, `dup2`, `open`, `pipe`, `socket`

Hide solution

Hide solution

QUESTION SH-6B. Which of these system calls may close a file descriptor? List all that apply. Note that some system calls might close a file descriptor even though the total number of open file descriptors remains the same.

Show solution

`close`, `dup2`

Hide solution

Hide solution

QUESTION SH-6C. Which of these system calls can block? List all that apply.

Show solution

`accept`, `connect`, `read`, `select`, `write`

The following system calls can also block but only in rare situations, such as closing a file descriptor on an NFS file system, or for short times, such as opening a file on disk or binding a Unix-domain socket on a disk file system: `bind`, `open`, `close`.

I don't believe the others—`dup2`, `listen`, `pipe`, `socket`—ever meaningfully block.

Hide solution

Hide solution

QUESTION SH-6D. Which system calls can open at least one file descriptor where that file descriptor is suitable for both reading and writing? List all that apply.

Show solution

`open` (`O_RDWR`), `accept`, `socket`.

`connect` is also OK to mention, even though it doesn't create a new file descriptor.

`dup2` is also OK to mention, even though it doesn't really "open" a file descriptor (though it does unambiguously cause the number of open file descriptors to increase).

Hide solution

Hide solution

QUESTION SH-6E. ⚠️ Which system calls must a network server make in order to receive a connection on a well-known port? List all that apply **in order**, first to last. Avoid unnecessary calls.

Show solution

`socket`, `bind`, `listen`, `accept`

Hide solution

Hide solution

QUESTION SH-6F. ⚠️ Which system calls must a network *client* make in order to (1) connect to a server, (2) send a message, (3) receive a reply, and (4) close the connection? List all that apply **in order**, first to last. Avoid unnecessary calls.

Show solution

`socket`, `connect`, `write`, `read`, `close`

Hide solution

Hide solution

SH-7. Duplication

Mark Zuckerberg hates duplicates (because Winklevii). He especially hates the `dup2` system call, because he can't remember its order of arguments.

⚠️ *Some parts of this question rely on material from the synchronization unit.*

QUESTION SH-7A. What *is* the order of arguments for `dup2`? Is it (A) `dup2(oldfd, newfd)` or (B) `dup2(newfd, oldfd)`? (Here, `oldfd` is the pre-existing file descriptor.)

Show solution

`oldfd, newfd`

Hide solution

Hide solution

Mark wants to make `dup2` obsolete by changing other system calls. He wants to:

- Replace `open(const char* path, int oflag, [mode_t mode])` with `openonto(int fd, const char* path, int oflag, [mode_t mode])`. This system call behaves like `open`, but rather than choosing a previously-unused file descriptor, it uses file descriptor number `fd`.
- Replace `pipe(int fd[2])` with `pipeonto(readfd, writefd)`, which uses the specified file descriptor numbers for the pipe's read and write ends.
- Add `nextfd()`, which returns the lowest-numbered currently-unused file descriptor.

These system calls can fail, meaning they return –1 and set `errno` to an error code.

- If `openonto` or `pipeonto` fails, the process's file descriptor table is unchanged.
- If `openonto` succeeds, it returns its `fd` argument.
- If an `fd` argument is out of range (e.g., less than 0), `openonto` and `pipeonto` return –1 and set `errno` to `EBADF`.
- `nextfd` can return –1 and set `errno` to `EMFILE` if too many file descriptors are open.

QUESTION SH-7B. Assuming a single-threaded process, show how to implement `open`'s functionality in terms of these new system calls. Don't worry about `mode`. Unix's `open` cannot set `errno` to `EBADF`; neither should yours.

```
int open(const char* path, int oflag) {
```

Show solution

```
    int fd = nextfd();
    if (fd != -1) {
        fd = openonto(fd, path, oflag);
    }
    return fd;
```

Hide solution

Hide solution

QUESTION SH-7C. ⚠️ Your `open` implementation likely has a race condition if used in a multithreaded process: a bug can occur if two threads call `open` at about the same time. Explain this race condition briefly (two or three sentences max).

Show solution

Two threads can use the same `nextfd`.

Hide solution

Hide solution

QUESTION SH-7D. ⚠️ Solve this race condition using synchronization objects. You may introduce global variables, which we'll assume are initialized. Again, be careful to handle error conditions properly.

```
int open(const char* path, int oflag) {
```

Show solution

```
    extern std::mutex m;
    std::scoped_lock guard(m);
    ...previous code...
    return fd;
```

Hide solution

Hide solution

QUESTION SH-7E. Can these system calls (without `dup` or `dup2`) implement a shell pipeline? Why or why not? Be brief.

Show solution

Nope: there is no way to shift the pipe ends onto stdin/stdout without replacing the shell's stdin/stdout.

Hide solution

Hide solution

Sheryl Sandberg is sympathetic to Mark's psychological issues, but suggests instead that he replace `dup` and `dup2` with:

- `fdswap(int fd1, int fd2)`. This swaps the *meanings* of two file descriptors. Thus, after `fdswap(fd1, fd2)`, `fd1` references the file structure previously referenced by `fd2`, and vice versa.

QUESTION SH-7F. Complete the following function, using at least `pipe`, `fork`, `execvp`, `fdswap`, and `close`. **Do not** use `dup`, `dup2`, or `pipeonto`, and don't worry too much about error conditions. Make sure to implement pipe hygiene. **Hint:** The programs in `cs61-lectures/synch2` might be useful references.

```
// simplepipeline_fdswap(cmd1, cmd2)
//    Fork and execute the pipeline `cmd1 | cmd2`. Return the `pid_t` corresponding to
`cmd2`
//    (or return -1 with an appropriate error code if the pipeline could not be
created).

pid_t simplepipeline_fdswap(const char* cmd1, const char* cmd2) {
    char* const cmd1_argv[] = { (char*) cmd1, nullptr };
    char* const cmd2_argv[] = { (char*) cmd2, nullptr };
```

Show solution

```
    int pfd[2];
    if (pipe(pfd) != 0) {
        return -1;
    }
    int pid1 = fork();
    if (pid1 == 0) {
        fdswap(pfd[1], 1);
        close(pfd[1]);
        close(pfd[0]);
        execvp(cmd1, cmd1_argv);
    }
    int pid2 = fork();
    if (pid2 == 0) {
        fswap(pfd[0], 0);
        close(pfd[0]);
        close(pfd[1]);
        execvp(cmd2, cmd2_argv);
    }
    close(pfd[0]);
    close(pfd[1]);
    return pid2;
```

Hide solution

Hide solution

Synchronization exercises

Exercises that seem less appropriate this year, or which cover topics that we haven't covered in class, should be marked with ⚠️. However, we may have missed some.

SYNCH-1. Threads

The following code performs a matrix multiplication, `c = ab`, where `a`, `b`, and `c` are all square matrices of dimension `sz`. It uses the cache-friendly *ikj* index ordering.

```
struct square_matrix {
    size_t sz;
    double* v;    // array of `sz * sz` doubles, row-major order

    double& elt(size_t i, size_t j) {
        assert(i < this->sz && j < this->sz);
        return this->v[i * this->sz + j];
    }
};

void matrix_multiply(square_matrix& c, square_matrix& a, square_matrix& b) {
    assert(a.sz == b.sz && b.sz == c.sz);
    for (size_t x = 0; x != c.sz * c.sz; ++x) {
        c.v[x] = 0.0;
    }
    for (size_t i = 0; i != c.sz; ++i) {
        for (size_t k = 0; k != c.sz; ++k) {
            for (size_t j = 0; j != c.sz; ++j) {
                c.elt(i, j) += a.elt(i, k) * b.elt(k, j);
            }
        }
    }
}
```

But matrix multiplication is a naturally parallelizable problem.

QUESTION SYNCH-1A. Complete this code, which should perform the multiplication using `c.sz` parallel threads, one per *row* of `c`.

```cpp
void matrix_multiply_thread(square_matrix& c, square_matrix& a, square_matrix& b,
                            size_t i) {
    /* YOUR CODE HERE */
}

void matrix_multiply_p(square_matrix& c, square_matrix& a, square_matrix& b) {
    assert(a.sz == b.sz && b.sz == c.sz);
    for (size_t x = 0; x != c.sz * c.sz; ++x) {
        c.v[x] = 0.0;
    }
    std::vector<std::thread> ts;
    for (size_t i = 0; i != c.sz; ++i) {
        ts.push_back(std::thread(matrix_multiply_thread, std::ref(c), std::ref(a),
std::ref(b), i));
    }
    for (size_t i = 0; i != c.sz; ++i) {
        ts[i].join();
    }
}
```

Show solution

```cpp
void matrix_multiply_thread(square_matrix& c, square_matrix& a, square_matrix&
b,
                            size_t i) {
    for (size_t k = 0; k != c.sz; ++k) {
        for (size_t j = 0; j != c.sz; ++j) {
            c.elt(i, j) += a.elt(i, k) * b.elt(k, j);
        }
    }
}
```

Hide solution

Hide solution

For the next two parts, consider this alternate code for parallel `matrix_multiply`, and assume a correct `matrix_multiply_thread` implementation.

```
void matrix_multiply_alt(square_matrix& c, square_matrix& a, square_matrix& b) {
    assert(a.sz == b.sz && b.sz == c.sz);
    for (size_t x = 0; x != c.sz * c.sz; ++x) {
        c.v[x] = 0.0;
    }
    std::vector<std::thread> ts;
    for (size_t i = 0; i != c.sz; ++i) {
        ts.push_back(std::thread(matrix_multiply_thread, std::ref(c), std::ref(a),
std::ref(b), i));
        ts[i].join();
    }
}
```

QUESTION SYNCH-1B. True or false? `matrix_multiply_alt` produces correct results.

Show solution

True.

Hide solution

Hide solution

QUESTION SYNCH-1C. How does `matrix_multiply_alt` differ from `matrix_multiply_p`?

Show solution

`matrix_multiply_alt` does not run threads for different rows in parallel.

Hide solution

Hide solution

QUESTION SYNCH-1D. On single-core machines, the *kij* order performs almost as fast as the *ikj* order. Describe the changes that would be required to make `matrix_multiply_p` and `matrix_multiply_thread` use *kij* order (and produce correct results).

Show solution

For *incorrect* results, a simple change suffices:

```
void matrix_multiply_thread(square_matrix& c, square_matrix& a, square_matrix&
b,
                            size_t k) {
    for (size_t i = 0; i != c.sz; ++i) {
        for (size_t j = 0; j != c.sz; ++j) {
            c.elt(i, j) += a.elt(i, k) * b.elt(k, j);
        }
    }
}
```

But this isn't right because multiple threads can access `c.elt(i, j)` at the same time using reads *and* writes, violating the fundamental law of synchronization. Atomic operations or locking would be required to make this correct.

Hide solution

Hide solution

SYNCH-2. Synchronization and concurrency

Most synchronization objects have at least two operations. Mutual-exclusion locks support lock and unlock; condition variables support wait and signal. One of the earliest synchronization objects invented, the *semaphore*, supports P and V.

In this problem, you'll work with a synchronization object with only *one* operation, which we call a **hemiphore**. Hemiphores behave like the following; **it is very important that you understand this pseudocode.**

```
struct hemiphore {
    int value = 0;

    // Block until the hemiphore has `this->value >= bound`, then ATOMICALLY
    // increment its value by `delta`.
    void H(int bound, int delta) {
        // This is pseudocode; a real hemiphore implementation would block, not spin,
and would
        // ensure that the test and the increment happen in one atomic step.
        // You may assume that `this->value` never overflows.
        while (this->value < bound) {
            sched_yield();
        }
        this->value += delta;
    }
};
```

Application code should access the hemiphore only through the `H` operation.

QUESTION SYNCH-2A. Use hemiphores to implement mutual-exclusion locks. Fill out the code below. (You may not need to fill in every empty slot. You may use standard C constants; for example, `INT_MIN` is the smallest possible value for a variable of type `int`, which on an x86-64 machine is −2147483648.)

```
struct mutex {
    hemiphore h;
    // YOUR CODE HERE (if necessary)

    // Initialize the mutex.
    mutex() {
        // YOUR CODE HERE (if necessary)
    }

    // Lock the mutex.
    void lock() {
        // YOUR CODE HERE
    }

    // Unlock the mutex, which must be locked.
    void unlock() {
        // YOUR CODE HERE
    }
};
```

Show solution

No additional members are necessary. However, we don't want to use a *positive* number to represent the locked state. The `H` operation blocks if the value is *too small*. The locked state should cause blocking, so the locked state should have a *smaller* value than the unlocked state.

```
struct mutex {
    hemiphore h;
    void lock() {
        this->h.H(0, -1);
    }
    void unlock() {
        this->h.H(-1, 1);
    }
};
```

Hide solution

Hide solution

QUESTION SYNCH-2B. Use hemiphores to implement condition variables. Fill out the code below. You may assume that the implementation of `mutex` is your hemiphore-based implementation from above (so, for instance, `wait` may access the hemiphore `m.h`). See the Hints at the end of the question.

```
struct condition_variable {
    mutex internal;
    hemiphore h;
    // YOUR CODE HERE (if necessary)

    // Initialize the condition variable.
    condition_variable() {
        // YOUR CODE HERE (if necessary)
    }

    // Wake up one thread waiting on the condition variable (if any are waiting).
    void notify_one() {
        // YOUR CODE HERE
    }

    // Block until the condition variable is signaled. The mutex argument `m` must be
locked by
    // the current thread; it is unlocked before the wait begins and re-locked after the
wait ends.
    // There must be no sleep-wakeup race conditions: if thread 1 has `m` locked and
executes
    // `cv.wait(m)`, no other thread is waiting on `cv`, and thread 2 executes
`m.lock();
    // cv.notify_one(); m.unlock()`, then thread 1 will always receive the signal (i.e.,
wake up).
    void wait(mutex& m) {
        // A true C++ condition_variable would take a `std::unique_lock`—we elide that
here.
        // YOUR CODE HERE
    }
};
```

Hints. For full credit:

- If no thread is waiting on condition variable `cv`, then `cv.notify_all()` should have no effect.
- Assume *N* threads are waiting on condition variable `cv`. Then *N* calls to `cond_signal(c)` are both necessary and sufficient to wake them all up.
- Your solution must not add new sleep-wakeup race conditions to the user's code. (That is, no sleep-wakeup race conditions unless the user uses mutexes incorrectly.)

Show solution

Some care is required to avoid sleep–wakeup races. A solution involves counting the number of waiting threads, using a number protected by the `internal` mutex.

```
struct condition_variable {
    mutex internal;
    hemiphore h;
    int nwaiting = 0;

    void notify_one() {
        this->internal.lock();
        if (this->nwaiting > 0) {
            this->h.H(INT_MIN, 1);
            this->nwaiting -= 1;
        }
        this->internal.unlock();
    }

    void wait(mutex& m) {
        this->internal.lock();
        this->nwaiting += 1;
        this->internal.unlock();
        m.unlock();                // note: Must unlock AFTER incrementing
`nwaiting`
        this->h.H(1, -1);
        m.lock();
    }
};
```

Hide solution

Hide solution

QUESTION SYNCH-2C. Use C++ standard mutexes and condition variables to implement hemiphores. Fill out the code below; see the hints after the question.

```
struct hemiphore {
    std::mutex m;
    std::condition_variable_any cv;
    int value = 0;
    // YOUR CODE HERE (if needed)

    // Initialize the hemiphore.
    hemiphore() {
        // YOUR CODE HERE (if needed)
    }

    // Block until the hemiphore has `this->value >= bound`, then ATOMICALLY
    // increment its value by `delta`.
    void H(int bound, int delta) {
        // YOUR CODE HERE
    }
};
```

Show solution

```
struct hemiphore {
    std::mutex m;
    std::condition_variable_any cv;
    int value = 0;

    void H(int bound, int delta) {
        this->m.lock();
        while (this->value < bound) {
            this->cv.wait(this->m);
        }
        this->value += delta;
        if (delta > 0) {
            this->cv.notify_all();
        }
        this->m.unlock();
    }
};
```

Hide solution

Hide solution

QUESTION SYNCH-2D. Consider the following two threads, which use a shared hemiphore `h` with initial value 0.

```
Thread 1                      Thread 2
```

```
h.H(1000, 1);                   while (1) {
printf("Thread 1 done\n");          h.H(0, 1);
                                    h.H(0, -1);
                                }
```

Thread 2 will never block, and the hemiphore's value will alternate between 1 and 0. Thread 1 will never reach the `printf`, because the hemiphore's value never reaches 1000. However, in most people's first implementation of hemiphores using standard mutexes and condition variables, Thread 1 *will not block*. Every call to `h.H` in Thread 2 will effectively wake up Thread 1. Though Thread 1 will then check the hemiphore's value and immediately go back to sleep, doing so wastes CPU time.

Design an implementation of hemiphores using pthread mutexes and condition variables that solves this problem. In your revised implementation, Thread 1 above should block forever. For full credit, write C++ code (without worrying too much about C++ syntax). For partial credit, write pseudocode or English describing your design.

Hint. One working implementation uses a vector of "waiter" objects, where each waiter object is on a different thread's stack, as initially sketched below. You can use such objects or not as you please.

```
struct hemiphore_waiter {
    // YOUR CODE HERE (if necessary)
};

struct hemiphore {
    std::mutex m;
    int value = 0;
    std::vector<hemiphore_waiter*> waiters;
    // YOUR CODE HERE (if necessary)

    hemiphore() {
        // YOUR CODE HERE (if necessary)
    }

    void H(int bound, int delta) {
        hemiphore_waiter hw;
        // YOUR CODE HERE
    }
};
```

Show solution

This is a bit of a tough one. The key idea is to introduce a condition variable *per waiting thread*.

```
struct hemiphore_waiter {
    int bound;
    std::condition_variable_any cv;
};

struct hemiphore {
    std::mutex m;
    int value = 0;
    std::vector<hemiphore_waiter*> waiters;

    void H(int bound, int delta) {
        hemiphore_waiter hw;
        hw.bound = bound;
        this->m.lock();
        while (this->value < bound) {
            this->waiters.push_back(&hw);
            hw.cv.wait(this->m);
        }
        this->value += delta;
        // Wake up *only* those threads that should be woken.
        // This loop is written using iterators; there are many other styles.
        for (auto it = this->waiters.begin(); it != this->waiters.end(); ) {
            if (this->value >= it->bound) {
                it->cv.notify_all();
                it = this->waiters.erase(it);
            } else {
                ++it;
            }
        }
        this->m.unlock();
    }
};
```

Hide solution

Hide solution

SYNCH-3. Pipes and synchronization

In the following questions, you will implement a mutex using a pipe, and a limited type of pipe using a mutex.

QUESTION SYNCH-3A. In this question, you are to implement mutex functionality using a pipe. Fill in the definitions of the `mutex` operations. You may assume that no errors occur.

```
struct pipe_mutex {
    int fd[2];
    // YOUR CODE HERE (if necessary)

    pipe_mutex() {
        int r = pipe(this->fd);
        assert(r == 0);
        // YOUR CODE HERE (if necessary)
    }

    void lock() {
        // YOUR CODE HERE
    }

    void unlock() {
        // YOUR CODE HERE
    }
};
```

Show solution

The most natural way to implement this is to use `read` as the blocking operation, which requires an *unlocked* pipe mutex's pipe contain a byte (so that `read` on such a pipe doesn't block).

```
struct pipe_mutex {
    int fd[2];
    // YOUR CODE HERE (if necessary)

    pipe_mutex() {
        int r = pipe(this->fd);
        assert(r == 0);
        ssize_t n = write(this->fd, "!", 1);
        assert(n == 1);
    }

    void lock() {
        char ch;
        while (read(this->fd, &ch, 1) != 1) {
        }
    }

    void unlock() {
        ssize_t n = write(this->fd, "!", 1);
        assert(n == 1);
    }
};
```

Hide solution

Hide solution

In the next questions, you will help implement pipe functionality using an in-memory buffer and a mutex. This "mutex pipe" will only work between threads of the same process (in contrast to a regular pipe, which also works between processes). An initial implementation of mutex pipes is as follows; you will note that it contains no mutexes.

```
        struct mutex_pipe {
/* 1*/      char bbuf_[BUFSIZ];
/* 2*/      size_t bpos_;
/* 3*/      size_t blen_;

            mutex_pipe() {
/* 4*/          this->bpos_ = this->blen_ = 0;
/* 5*/          memset(this->bbuf_, 0, BUFSIZ);
            }

            // Read up to `sz` bytes from this mutex_pipe into `buf` and return the number of bytes
            // read. If no bytes are available, wait until at least one byte can be read.
            ssize_t read(char* buf, size_t sz) {
/* 6*/           size_t pos = 0;
/* 7*/           while (pos < sz && (pos == 0 || this->blen_ != 0)) {
/* 8*/               if (this->blen_ != 0) {
/* 9*/                   buf[pos] = this->bbuf_[this->bpos_];
/*10*/                   ++this->bpos_;
/*11*/                   this->bpos_ = this->bpos_ % BUFSIZ;
/*12*/                   --this->blen_;
/*13*/                   ++pos;
/*14*/               }
/*15*/           }
/*16*/           return pos;
            }

            // Write up to `sz` bytes from `buf` into this mutex_pipe and return the number of bytes
            // written. If no space is available, wait until at least one byte can be written.
            ssize_t write(const char* buf, size_t sz) {
/*17*/          size_t pos = 0;
/*18*/          while (pos < sz && (pos == 0 || this->blen_ < BUFSIZ)) {
/*19*/              if (this->blen_ != BUFSIZ) {
/*20*/                  size_t bindex = this->bpos_ + this->blen_;
/*21*/                  bindex = bindex % BUFSIZ;
/*22*/                  this->bbuf_[bindex] = buf[pos];
/*23*/                  ++this->blen_;
/*24*/                  ++pos;
/*25*/              }
/*26*/          }
/*27*/          return pos;
            }
        };
```

QUESTION SYNCH-3B. What's another name for this data structure?

Show solution

This is a bounded buffer.

Hide solution

Hide solution

It would be wise to work through an example. For example, assume `BUFSIZ == 4`, and figure out how the following calls would behave.

```
mutex_pipe mp;
mp.write("Hi", 2);
mp.read(buf, 4);
mp.write("Test", 4);
mp.read(buf, 3);
```

First let's reason about this code in the absence of threads.

QUESTION SYNCH-3C. Which of the following changes could, if made in isolation, result in undefined behavior when a mutex pipe was used? Circle all that apply.

1. Removing line 4
2. Removing line 5
3. Removing "`|| this->blen_ < BUFSIZ`" from line 18
4. Removing line 21
5. Removing lines 23 and 24

Show solution

Removing line 4 or line 21 will cause undefined behavior. The others will create correct, but not undefined, behavior.

Hide solution

Hide solution

QUESTION SYNCH-3D. Which of the following changes could, if made in isolation, cause a `mutex_pipe::read` to return incorrect data (that is, the byte sequence produced by `read` will not equal the byte sequence passed to `write`)? Circle all that apply.

1. Removing line 4

2. Removing line 5
3. Removing “`|| this->blen_ < BUFSIZ`” from line 18
4. Removing line 21
5. Removing lines 23 and 24

Show solution

> #1, #3, and #4 could cause a `read` to return incorrect data. #2 (removing line 5) has no effect. #5 will not cause `read` to return incorrect data, as you’ll see in the next question.
>
> Hide solution
>
> Hide solution

QUESTION SYNCH-3E. Which of the following changes could, if made in isolation, cause a call to `mutex_pipe::write` to never return (when a correct implementation would return)? Circle all that apply.

1. Removing line 4
2. Removing line 5
3. Removing “`|| this->blen_ < BUFSIZ`” from line 18
4. Removing line 21
5. Removing lines 23 and 24

Show solution

> The obvious one is that removing lines 23–24 will cause `pos` and `this->blen_` to never increment, which will cause the `write` `while` loop to spin forever. #1 and #4 cause undefined behavior, which could have any effect, including an infinite loop in a later `write`.
>
> Hide solution
>
> Hide solution

QUESTION SYNCH-3F. Write an invariant for a `mutex_pipe`’s `blen_` member. An invariant is a statement about the value of `blen_` that is always true. Write your invariant in the form of an assertion; for full credit give the most specific true invariant you can. (“`blen_` is a unsigned integer” is unspecific, but true; “`blen_ == 4`” is specific, but false.)

Show solution

```
assert(this->blen_ < BUFSIZ)
```

Hide solution

Hide solution

QUESTION SYNCH-3G. Write an invariant for `bpos_`. For full credit give the most specific true invariant you can.

Show solution

```
assert(this->bpos_ < BUFSIZ)
```

Hide solution

Hide solution

In the remaining questions, you will add synchronization objects and operations to make your mutex pipe work in a multithreaded program.

QUESTION SYNCH-3H. Add a `std::mutex` to the `mutex_pipe` and use it to protect the mutex pipe from race condition bugs. For full credit, your solution *must not deadlock*—if one thread is reading from a pipe and another thread is writing to the pipe, then both threads must eventually make progress. Describe all changes required, with reference to specific line numbers.

Show solution

It's important to lock the mutex before accessing any shared state. It's also important not to spin in a `while` loop while holding a mutex; we therefore have an initial loop that spins until data is available for `read` (or space is available for `write`).

- Add `std::mutex m` to `pipe_mutex`.
- Add `this->m.lock()` after lines 6 and 17.
- Add `this->m.unlock()` after lines 15 and 26.
- Add the following loop after the `m.lock()` in `read`:

```
while (this->blen_ == 0 && sz != 0) {
    this->m.unlock();
    sched_yield();     // give other threads a chance to run (not necessary)
    this->m.lock();
}
```

- Add a similar loop after the `m.lock()` in `write`:

```
while (this->blen_ == BUFSIZ && sz != 0) {
    this->m.unlock();
    sched_yield();     // give other threads a chance to run (not necessary)
    this->m.lock();
}
```

Hide solution

Hide solution

QUESTION SYNCH-3I. Your solution to the last question likely has poor utilization. For instance, a thread that calls `mutex_pipe` on an empty mutex pipe will spin forever, rather than block. Introduce one or more condition variables so that `read` will block until data is available. Write one or more snippets of C code and give line numbers after which the snippets should appear.

Show solution

- Add `std::condition_variable nonempty` to `struct mutex_pipe`.
- Instead of the loop suggested above in `read`, use:

```
while (this->blen_ == 0 && sz != 0) {
    this->nonempty.wait(this->m);
}
```

- Add the following code to the end of `write`, anywhere before the `return`:

```
this->nonempty.notify_all();
```

Hide solution

Hide solution

SYNCH-4. Race conditions

Most operating systems support process *priority levels*, where the kernel runs higher-priority processes more frequently than lower-priority processes. A hypothetical Unix-like operating system called "Boonix" has two priority levels, *normal* and *batch.* A Boonix parent process changes the priority level of one of its children with this system call:

- **`int setbatch(pid_t p)`**
  Sets process `p` to have batch priority. All future children of `p` will also have batch priority. Returns 0 on success, –1 on error. Errors include `ESRCH`, if `p` is not a child of the calling process.

Note that a process cannot change its own batch status.

You're writing a Boonix shell that can run commands with batch priority. If `c->isbatch` is nonzero, then `c` should run with batch priority, as should its children. Your `command::make_child` function looks like this:

```
void command::make_child() {
    ... // maybe create a pipe
    this->pid = fork();
    if (this->pid == 0) {
        ... // handle pipes and redirections
        (void) execvp(...);
        perror("execvp");
        _exit(EXIT_FAILURE);
    }
    assert(this->pid > 0);
    if (this->isbatch) {
        setbatch(this->pid);
    }
    ... // clean up pipes and such
}
```

This shell has two race conditions, one more serious.

QUESTION SYNCH-4A. In some cases, a child command will change to batch priority after it starts running. Briefly describe how this can occur.

Show solution

> The child's `execvp` might complete, and its new program start running, before the parent shell calls `setbatch`.
>
> Hide solution
>
> Hide solution

QUESTION SYNCH-4B. In some cases, a child command, or one of its own forked children, could run *forever* with normal priority. Briefly describe how this can occur.

Show solution

> As in part A, the child might `execvp` and run before the parent calls `setbatch`. If the child itself forks, then the resulting grandchild will have normal priority; the `setbatch` system call will not catch it.
>
> Hide solution
>
> Hide solution

In the remaining questions, you will fix these race conditions in three different ways. The first uses a new system call:

- **`int isbatch()`**
  Returns 1 if the calling process has batch priority, 0 if it has normal priority.

QUESTION SYNCH-4C. Use `isbatch` to prevent both race conditions. Write a snippet of C code and give the line number after which it should appear. You should need one code snippet.

Show solution

> Add this in the child, before `execvp` (after line 3 or 4):
>
> ```
> while (this->isbatch && !isbatch()) {
> }
> ```
>
> Hide solution
>
> Hide solution

QUESTION SYNCH-4D. Use the `pipe` system call and friends to prevent both race conditions. Write snippets of C code and give the line numbers after which they should appear. You should need several snippets. Make sure you clean up any extraneous file descriptors before running the command or returning from `command::make_child`.

Show solution

In the parent, before `fork` (after line 1):

```
int bpipe[2];
if (this->isbatch) {
    int r = pipe(bpipe);
    assert(r == 0);
}
```

In the child, before `execvp` (after line 3 or 4):

```
char ch;
while (this->isbatch && read(bpipe[0], &ch, 1) != 1) {
}
close(bpipe[0]);
close(bpipe[1]);
```

In the parent, after calling `isbatch` and before exiting:

```
if (this->isbatch) {
    ssize_t n = write(bpipe[1], "!", 1);
    assert(n == 1);
    close(bpipe[0]);
    close(bpipe[1]);
}
```

Other solutions are possible too.

Hide solution

Hide solution

QUESTION SYNCH-4E. Why should the `pipe` solution be preferred to the `isbatch` solution? A sentence, or the right single word, will suffice.

Show solution

It blocks! Which improves utilization relative to the `isbatch` polling-based solution.

Hide solution

Hide solution

QUESTION SYNCH-4F. Suggest a change to the `setbatch` system call's behavior that could fix both race conditions, and say how to use this new `setbatch` in `start_command`. Write one or more snippets of C code and give the line numbers after which they should appear.

Show solution

A very simple change would be to allow a process to set its own batchness. Then get rid of the call in the parent. In the child, before `execvp`:

```
if (c->isbatch) {
    setbatch(getpid());
}
```

Hide solution

Hide solution

SYNCH-5. Minimal minimal minimal synchronization synchronization synchronization

Minimalist composer Philip Glass, who prefers everything minimal, proposes the following implementation of condition variables based on mutexes. He's only implementing `wait` and `notify_one` at first.

```
struct pg_condition_variable {
    std::mutex cvm;

    pg_condition_variable() {
        // start the mutex in LOCKED state!
        this->cvm.lock();
    }

    void wait(std::mutex& m) {
        m.unlock();
        this->cvm.lock();   // will block until a thread calls `notify_one`
        m.lock();
    }

    void notify_one() {
        this->cvm.unlock();
    }
};
```

Philip wants to use his condition variables to build a bank, where accounts support these operations:

- **`void pg_acct::deposit(unsigned amt)`**
  Adds `amt` to `this->balance`.
- **`void pg_acct::withdraw(unsigned amt)`**
  Blocks until `this->balance >= amt`; then deducts `amt` from `this->balance` and returns.

Here's Philip's code.

```
      struct pg_acct {
          unsigned long balance;
          std::mutex m;
          pg_condition_variable cv;

          void deposit(unsigned amt) {
/*D1*/        this->m.lock();
/*D2*/        this->balance += amt;
/*D2*/        this->cv.notify_one();
/*D2*/        this->m.unlock();
          }

          void withdraw(unsigned amt) {
/*W1*/        this->m.lock();
/*W2*/        while (this->balance < amt) {
/*W3*/            this->cv.wait(this->m);
/*W4*/        }
/*W5*/        this->balance -= amt;
/*W6*/        this->m.unlock();
          }
      };
```

Philip's friend Pauline Oliveros just shakes her head. "You got serious problems," she says, pointing at this section of the C++ standard:

> The expression `m.unlock()` shall…have the following semantics:
>
> *Requires:* The calling thread shall own the mutex.

This means that the when `m.unlock()` is called, the calling thread must have previously locked the mutex, with no intervening unlocks. The penalty for deviance is undefined behavior.

QUESTION SYNCH-5A. Briefly explain how Philip's code can trigger undefined behavior.

Show solution

> There are so many bad undefined behaviors here. For instance, two sequential calls to `deposit` will call `cv.notify_one()` twice, which calls `cv.cvm.unlock()` twice.
>
> Hide solution
>
> Hide solution

To fix this problem, Philip changes his condition variable and account to use a new type, `fast_mutex`, instead of `std::mutex`. This type is inspired by Linux's "fast" mutexes. It's OK to unlock a `fast_mutex` more than once, and it's OK to unlock a `fast_mutex` on a different thread than the thread that locked it.

A `fast_mutex` has one important member, `value`, which can be 0 (unlocked) or 1 (locked).

Below, we've begun to write out an execution where Philip's code is called by two threads. We write the line numbers each thread executes and the values in `a` after each line. We've left lines blank for you to fill in if you need to.

T1	T2	`a.balance`	`a.m.value`	`a.cv.cvm.value`
Initial state:		5	0	1
`a.deposit(10)`…	`a.withdraw(12)`…			
	after W1	5	1	1
after D1		5	1	1
(T1 blocks on `a.m`)				

QUESTION SYNCH-5B. Assuming T2's call to `withdraw` eventually completes, what are the final values for `a.balance`, `a.m.value`, and `a.cv.cvm.value`?

Show solution

The execution will complete as follows:

T1	T2	`a.balance`	`a.m.value`	`a.cv.cvm.value`
Initial state:		5	0	1
`a.deposit(10)`…	`a.withdraw(12)`…			
	after W1	5	1	1
after D1		5	1	1
(T1 blocks on `a.m`)				
	W2	5	1	1
	W3	5	0	1
	(T2 blocks on `a.cv.cvm`)	5	0	1
(T1 unblocks)		5	1	1
D2		15	1	1
D3		15	1	0
D4		15	0	0
	(T2 unblocks)	15	1	1
	W2	15	1	1
	W5	3	1	1
	W6	3	0	1

The values are 3, 0, and 1, respectively.

Hide solution

Hide solution

QUESTION SYNCH-5C. In such an execution, which line of code (W1–5) unblocks Thread T1?

Show solution

Line W3, which calls `pg_condition_variable::wait`, which unlocks `a.m`.

Hide solution

Hide solution

QUESTION SYNCH-5D. In such an execution, which, if any, line(s) of code (D1–4 and/or W1–5) set `a->cv.cvm.value` to zero?

Show solution

> Line D3, which calls `pg_condition_variable::notify_one`.
>
> Hide solution
>
> Hide solution

QUESTION SYNCH-5E. For any collection of `deposit` and `withdraw` calls, Philip's code will always ensure that the balance is valid. (There are other problems—a `withdraw` call might block forever when it shouldn't—but the balance will be OK.) Why? List all that apply.

1. Access to `balance` is protected by a condition variable.
2. Access to `balance` is protected by a mutex.
3. Arithmetic expressions like `this->balance += amt;` have atomic effect.

Show solution

> #2.
>
> Hide solution
>
> Hide solution

SYNCH-6. Weensy threads

Betsy Ross is changing her WeensyOS to support threads. There are many ways to implement threads, but Betsy wants to implement threads using the `ptable` array. "After all," she says, "a thread is just like a process, except it *shares* the address space of some other process!"

Betsy has defined a new system call, `sys_create_thread`, that starts a new thread running a given thread function, with a given argument, and a given initial stack pointer:

```
typedef void* (*thread_function)(void*);
pid_t sys_create_thread(thread_function f, void* arg, void* stack_ptr);
```

The system call's return value is the ID of the new thread, which Betsy thinks should use the process ID space.

Betsy's kernel contains the following code:

```
// in syscall()
case SYSCALL_FORK:
    return handle_fork(current);

case SYSCALL_CREATE_THREAD:
    return handle_create_thread(current);


uint64_t handle_fork(proc* p) {
    // Find a free process; return `nullptr` if all out
    proc* np = find_free_process();
    if (!np) {
        return -1;
    }

    // Copy the input page table and allocate new pages using `vmiter`
    np->pagetable = copy_pagetable(p->pagetable);
    if (!np->pagetable) {
        return -1;
    }

    // Finish up
    np->regs = p->regs;
    np->regs.reg_rax = 0;
    np->state = P_RUNNABLE;
    return np->pid;
}

uint64_t handle_create_thread(proc* p) {
    // Whoops! Got a revolution to run, back later
    return -1;
}
```

QUESTION SYNCH-6A. Complete her `handle_create_thread` implementation. Assume for now that the thread function never exits, and don't worry about reference counting issues (for page tables, for instance). You may use the helper functions shown above, including `find_free_process` and `copy_pagetable`, if you need them; or you may use any functions or objects from the WeensyOS handout code, including `vmiter` and `kalloc`.

Recall that system call arguments are passed according to the x86-64 calling convention: first argument in `%rdi`, second in `%rsi`, third in `%rdx`, etc.

Show solution

The code is a lot like `fork`, except that (1) the new thread *shares* the same page table as the calling `proc`, (2) the new thread's registers are populated from the calling threads's arguments.

```
uint64_t handle_create_thread(proc* p) {
    proc* np = find_unused_process();
    if (!np) {
        return -1;
    }
    np->pagetable = p->pagetable;
    np->regs = p->regs;
    np->regs.reg_rip = p->regs.reg_rdi;
    np->regs.reg_rdi = p->regs.reg_rsi;
    np->regs.reg_rsp = p->regs.reg_rdx;
    np->state = P_RUNNABLE;
    return np->pid;
}
```

Hide solution

Hide solution

QUESTION SYNCH-6B. Betsy's friend Prince Dimitri Galitzin thinks Betsy should give processes even more flexibility. He suggests that the `sys_create_thread` system call should take a full set of registers (as `x86_64_registers*`), rather than just a new instruction pointer and a new stack pointer. That way, the creating thread can supply *all* registers to the new thread. But Betsy points out that this design would allow a thread to violate kernel isolation by providing carefully-planned register values for `x86_64_registers`.

Which x86-64 registers could be used in Dimitri's design to violate kernel isolation? List all that apply.

1. `reg_rax`, which contains the thread's `%rax` register.
2. `reg_rip`, which contains the thread's instruction pointer.
3. `reg_cs`, which contains the thread's privilege level, which is 3 for unprivileged.
4. `reg_rflags`, which contains the `EFLAGS_IF` flag, which indicates that the thread runs with interrupts enabled.
5. `reg_rsp`, which contains the thread's stack pointer.

Show solution

On WeensyOS, only #3 and #4 can cause violations of kernel isolation. Carefully-chosen #2 and #5 might cause the thread to crash, but that doesn't violate kernel isolation. Changes to #1 will have little if any effect.

Hide solution

Hide solution

Now Betsy wants to handle thread exit. She introduces two new system calls, `sys_exit_thread` and `sys_join_thread`:

```
void sys_exit_thread(void* exit_value);
void* sys_join_thread(pid_t thread);
```

`sys_exit_thread` causes the calling thread to exit with the given exit value. `sys_join_thread` behaves like `pthread_join` or `waitpid`. If `thread` corresponds is a thread of the same process, and `thread` has exited, `sys_join_thread` cleans up the thread and returns its exit value; otherwise, `sys_join_thread` returns `(void*) -1`.

(The `exit_value` feature differs from C++ threads, which don't have exit values.)

QUESTION SYNCH-6C. Is the `sys_join_thread` specification blocking or polling?

Show solution

Polling.

Hide solution

Hide solution

Betsy makes the following changes to WeensyOS internal structures to support thread exit.

1. She adds a `void* p_exit_value` member to `struct proc`.
2. She adds a new process state, `P_EXITED`, that corresponds to exited threads.

QUESTION SYNCH-6D. Complete the case for `SYSCALL_EXIT_THREAD`. (Don't worry about the last thread in a process; you may assume it always calls `sys_exit` rather than `sys_exit_thread`.)

```
case SYSCALL_EXIT_THREAD:
```

Show solution

```
    current->state = P_EXITED;
    current->exit_value = current->regs.reg_rdi;
    schedule();
```

Hide solution

Hide solution

QUESTION SYNCH-6E. Complete the following helper function.

```
// Test whether `test_pid` is the PID of a thread in the same process as `p`.
// Return 1 if it is; return 0 if `test_pid` is an illegal PID, it corresponds to
// a freed process, or it corresponds to a thread in a different process.
int is_thread_in(pid_t test_pid, proc* p) {
```

Show solution

The key thing to note is that threads in the same process will share `pagetable`, and threads in different processes will always have different `pagetable`s.

```
    return test_pid >= 0 && test_pid < NPROC && ptable[test_pid]->pagetable ==
p->pagetable;
```

Hide solution

Hide solution

QUESTION SYNCH-6F. Complete the case for `SYSCALL_JOIN_THREAD` in `syscall()`. Remember that a thread may be successfully joined at most once: after it is joined, its PID is made available for reallocation.

```
case SYSCALL_JOIN_THREAD:
```

Show solution

```
    pid_t t = current->regs.reg_rdi;
    if (is_thread_in(t, current) && ptable[t].state == P_EXITED) {
        ptable[t].state = P_FREE;
        return ptable[t].exit_value;
    } else {
        return -1;
    }
```

Hide solution

Hide solution

QUESTION SYNCH-6G. Advanced extra credit. In Weensy threads, if a thread returns from its thread function, it will execute random code, depending on what random garbage was stored in its initial stack in the return address position. But Betsy thinks she can implement better behavior entirely at user level, where the value returned from the thread function will automatically be passed to `sys_thread_exit`. She wants to make two changes:

1. She'll write a *two- or three-instruction function* called `thread_exit_vector`.
2. Her `create_thread` library function will write a *single 8-byte value* to the thread's new stack before calling `sys_create_thread`.

Explain how this will work. What instructions will `thread_exit_vector` contain? What 8-byte value will `create_thread` write to the thread's new stack? And where will that value be written relative to `sys_create_thread`'s `stack_ptr` argument?

Show solution

```
thread_exit_vector:
    movq %rax, %rdi
    jmp sys_exit_thread
```

Or:

```
    movq %rax, %rdi
    movq $SYSCALL_EXIT_THREAD, %rax
    syscall
```

The 8-byte value will equal the address of `thread_exit_vector`, and it will be placed in the return address slot of the thread's new stack. So it will be written starting at address `stack_top`.

Hide solution

Hide solution

SYNCH-7. Fair synchronization

C++ standard mutexes are *unfair*: some threads might succeed in locking the mutex more often than others. For example, a simple experiment on Linux shows that if threads repeatedly try to lock the same mutex, some threads lock the mutex 1.13x more often than others. (Other kinds of lock are even less fair: some threads can lock a spinlock 3.91x more often than others!)

QUESTION SYNCH-7A. What is the name of the problem that would occur if one particular thread *never* locked the mutex, even though other threads locked and unlocked the mutex infinitely often?

Show solution

Starvation.

Hide solution

Hide solution

To avoid unfairness, threads must take turns. One fair mutex implementation is called a *ticket lock*; this (incorrect, unsynchronized) code shows the basic idea.

```
struct ticket_mutex {
    unsigned now = 0;    // “now serving”
    unsigned next = 0;   // “next ticket”

    void lock() {
        unsigned t = this->next;     // mark this thread’s place in line
        ++this->next;                // next thread gets new place
        while (this->now != t) {     // wait for my turn
        }
    }

    void unlock() {
        ++this->now;
    }
};
```

QUESTION SYNCH-7B. Describe an instance of undefined behavior that could occur if multiple threads called `ticket_mutex::lock` on the same ticket mutex at the same time.

Show solution

Both threads might execute `++this->next` at the same time, violating the Fundamental Law of Synchronization.

Hide solution

Hide solution

QUESTION SYNCH-7C. Fix `lock` and `unlock` using C++ atomics. Alternately, for partial credit, say which regions of code must be executed atomically.

Show solution

It’s important to combine the assignment to `unsigned t` with the increment, like this for instance:

```
struct ticket_mutex {
    std::atomic<unsigned> now = 0;    // “now serving”
    std::atomic<unsigned> next = 0;   // “next ticket”

    void lock() {
        unsigned t = this->next.fetch_add(1);
        while (this->now != t) {
        }
    }
    void unlock() {
        ++this->now;
    }
};
```

Hide solution

Hide solution

The ticket lock implementation above uses polling. That will perform well if critical sections are short, but blocking is preferable if critical sections are long. Here’s a different ticket lock implementation:

```
      struct ticket_mutex {
/*T1*/    unsigned now;
/*T2*/    unsigned next;
/*T3*/    std::mutex m;

          void lock() {
/*L1*/        this->m.lock();
/*L2*/        unsigned t = this->next++;
/*L3*/        while (this->now != t) {
/*L4*/            this->m.unlock();
/*L5*/            sched_yield();
/*L6*/            this->m.lock();
/*L7*/        }
/*L8*/        this->m.unlock();
          }

          void unlock() {
/*U1*/        this->m.lock();
/*U2*/        ++this->now;
/*U3*/        this->m.unlock();
          }
      };
```

This ticket lock implementation uses `std::mutex`, which blocks, but the implementation itself uses polling.

QUESTION SYNCH-7D. Which line or lines of code mark this implementation as using polling?

Show solution

> Line L5, which calls `sched_yield()`.
>
> Hide solution
>
> Hide solution

QUESTION SYNCH-7E. Change the implementation to truly block. Include line numbers indicating where your code will go.

Show solution

> As with most block-on-condition requirements, this calls for a condition variable.
>
> After line T3, add `std::condition_variable_any cv`.
> Instead of L4–L6, put `this->cv.wait(this->m);`.
> After line U2, put `this->cv.notify_all();`.
>
> Hide solution
>
> Hide solution

QUESTION SYNCH-7F. Most solutions to part E wake up blocked threads more than is strictly necessary. The ideal number of blocking calls is **one**: each thread should block at most once and wake up only when its turn comes. But the simplest correct solution will wake up each blocked thread a number of times proportional to *the number of blocked threads*.

Change your solution so that when there are 4 or fewer threads, every thread wakes up only when its turn comes. (Your solution must, of course, work correctly for any number of threads.) If your solution already works this way, you need not do anything here.

Show solution

A simple solution is to have 4 conditions, each corresponding to "a ticket equal to this condition's index (mod 4) is available."

After line T3, add `std::condition_variable_any cv[4]`.
Instead of L4–L6, put `this->cv[t % 4].wait(this->m);`.
Replace line U2 with `auto t = ++this->now;`. (It's important to remember the actual new value of `now`.)
After line U2, put `this->cv[t % 4].notify_all()`.

Hide solution

Hide solution

NET-1. Networking

QUESTION NET-1A. Which of the following system calls should a programmer expect to sometimes block (i.e., to return after significant delay)? Circle all that apply.

1. `socket`
2. `read`
3. `accept`
4. `listen`
5. `connect`
6. `write`
7. `usleep`
8. None of these

Show solution

#2 `read`, #3 `accept`, #5 `connect`, #6 `write`, #7 `usleep`.

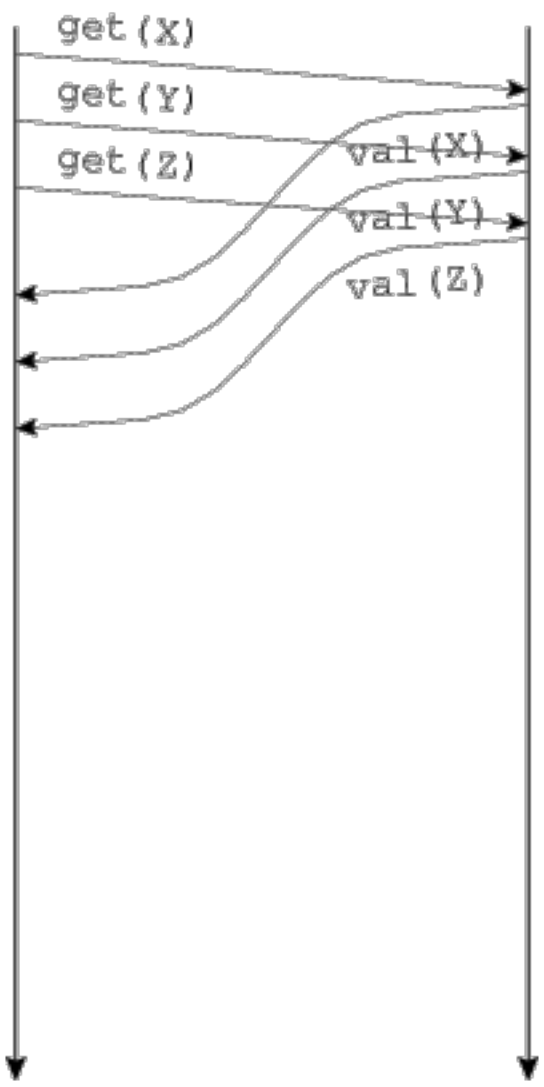
Hide solution

Hide solution

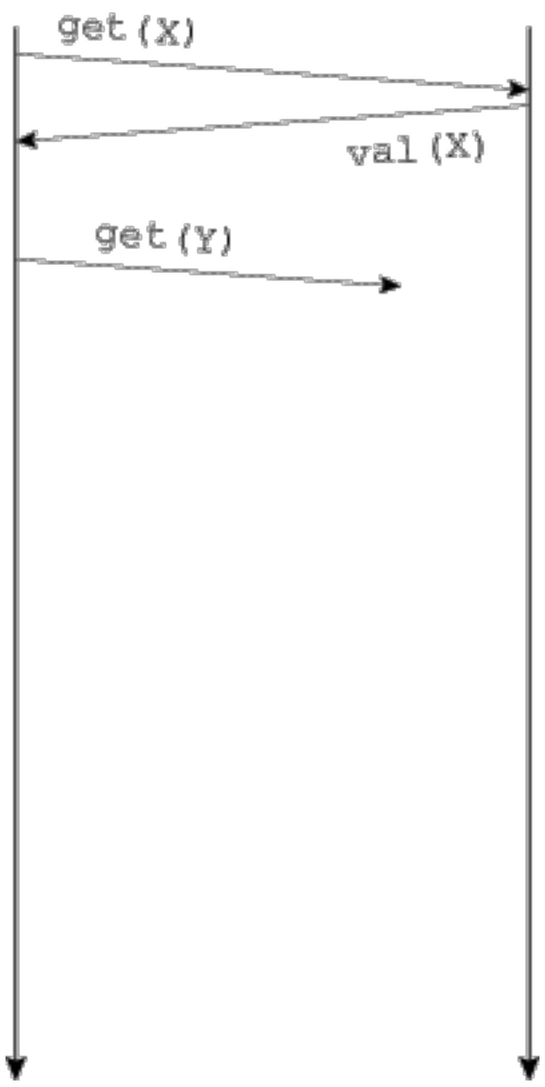
QUESTION NET-1B. ⚠️ Below are seven message sequence diagrams demonstrating the operation of a client–server RPC protocol. A request such as "get(X)" means "fetch the value of the object named X"; the response contains that value. Match each network property or programming strategy below with the diagram with which it best corresponds. You will use every diagram once.

1. Loss
2. Delay
3. Reordering

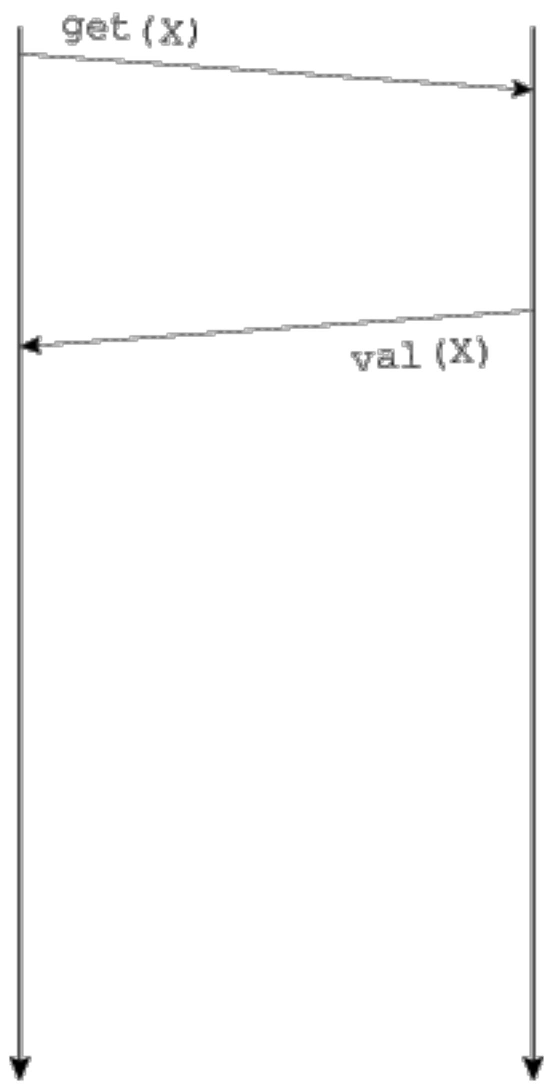
4. Duplication
5. Batching
6. Prefetching
7. Exponential backoff



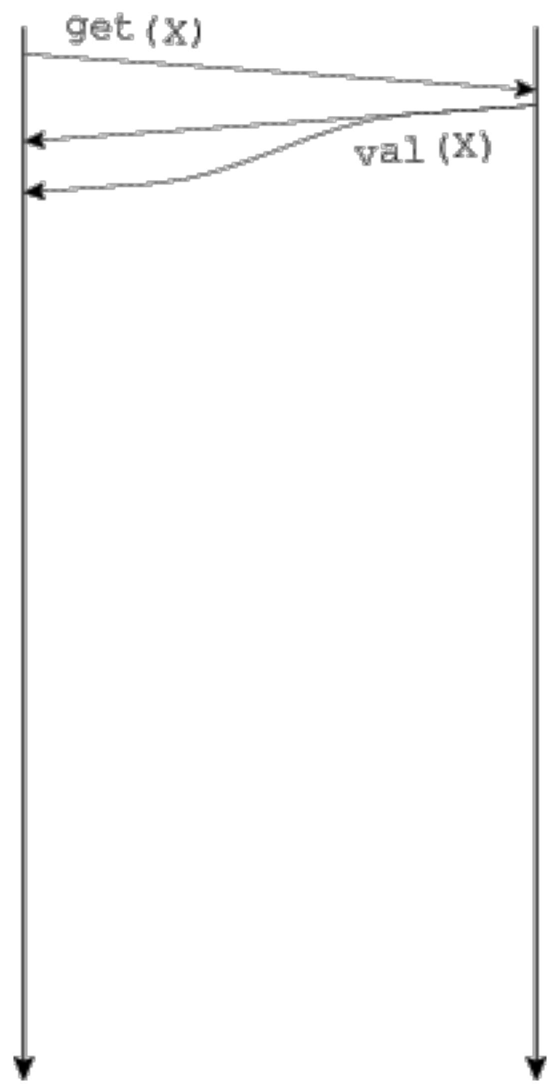
A



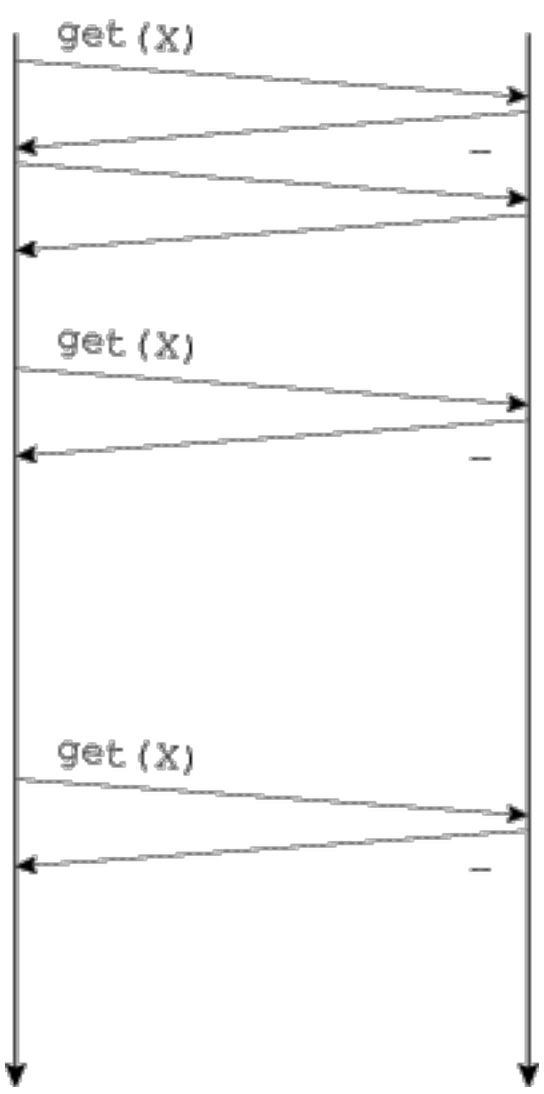
B



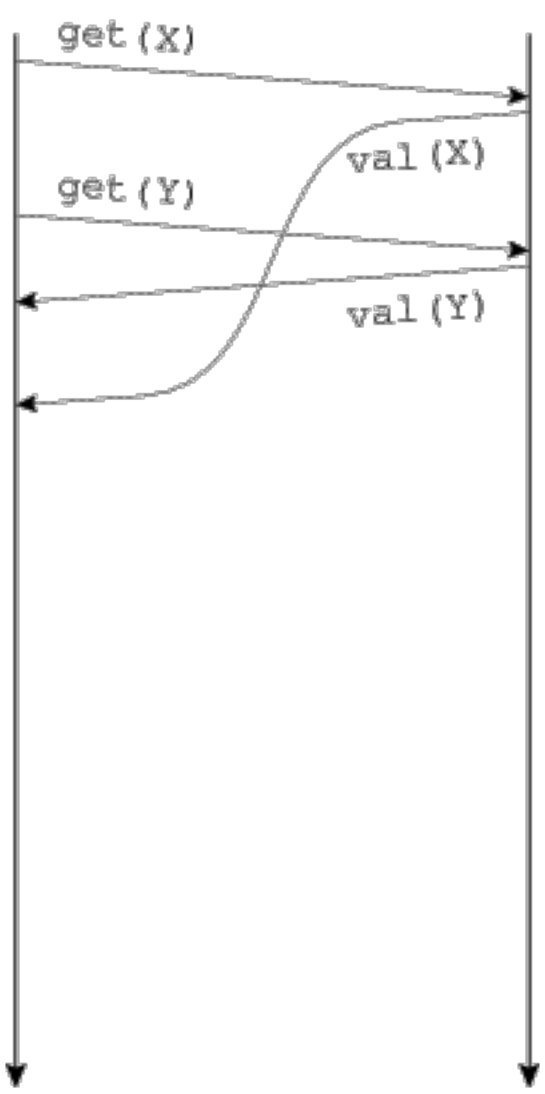
C



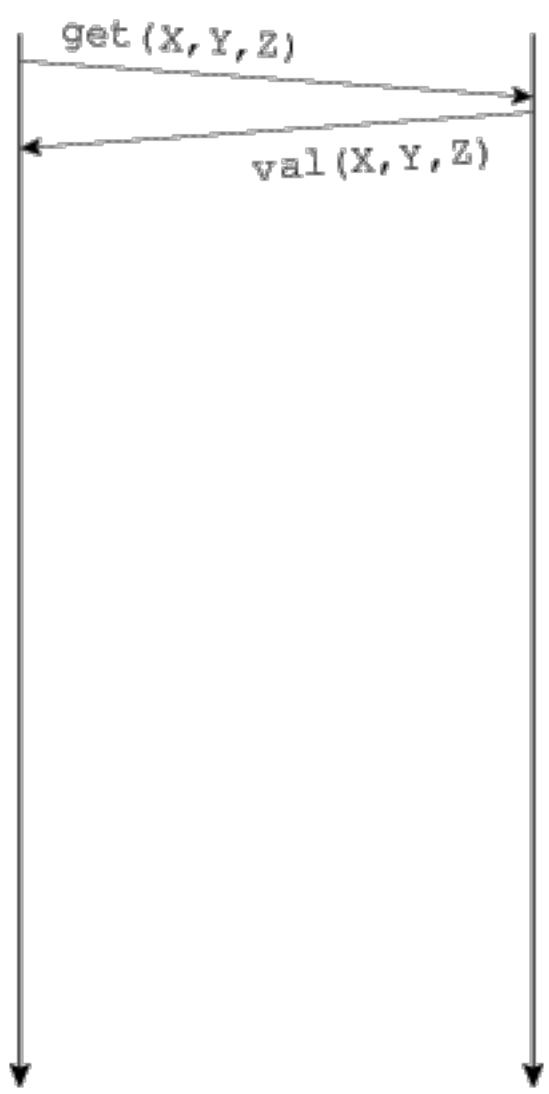
D



E



F



G

Show solution

#1—B, #2—C, #3—F, #4—D, #5—G, #6—A, #7—E
(A—#6, B—#1, C—#2, D—#4, E—#7, F—#3, G—#5)

While G could also represent prefetching, A definitely does not represent batching at the RPC level—each RPC contains one request—so under the rule that each diagram is used once, we must say G is batching and A is prefetching.

Hide solution

Hide solution

QUESTION NET-1C. List some resources that a DoS attack on a network server might exhaust.

Show solution

At least: file descriptors, memory (stack), processes/threads. There're a lot of correct answers, though! You can run out of virtual memory or even physical memory.

Hide solution

Hide solution

QUESTION NET-1D. A server sets up a socket to listen on a connection. When a client wants to establish a connection, how does the server manage the multiple clients? In your answer indicate what system call or calls are used and what they do.

Show solution

The server calls `accept` on a listening file descriptor. This creates a new file descriptor that is particular to the connection with a particular client, giving the server uses a different fd for each client.

Hide solution

Hide solution

Miscellaneous exercises

Exercises that seem less appropriate this year, or which cover topics that we haven't covered in class, should be marked with ⚠️. However, we may have missed some.

MISC-1. Git

Edward Snowden is working on a CS61 problem set and he has some git questions.

QUESTION MISC-1A. The CS61 staff has released some new code. Which commands will help Edward get the code from code.seas.harvard.edu into his repository? Circle all that apply.

1. `git commit`
2. `git add`
3. `git push`
4. `git pull`

Show solution

> #4
>
> Hide solution
>
> Hide solution

QUESTION MISC-1B. Edward has made some changes to his code. He hasn't run git since making the changes. He wants to upload his latest version to code.seas.harvard.edu. Put the following git commands in an order that will accomplish this goal. You won't necessarily use every command. You may add flags to a command (but you don't have to). If you add flags, tell us what they are.

1. `git commit`
2. `git add`
3. `git push`
4. `git pull`

Show solution

> #2, #1, #3; or "#1 with `-a`", #3
>
> Hide solution
>
> Hide solution

Edward Snowden's partner, Edward Norton, has been working on the problem set also. They've been working independently.

At midnight on October 10, here's how things stood. The `git log` for the partners' shared code.seas.harvard.edu repository looked like this. The committer is listed in (parentheses).

```
52d44ee Pset release. (kohler)
```

The `git log` for Snowden's local repository:

```
3246d07 Save Greenwald's phone number (snowden)
8633fbd Start work on a direct-mapped cache (snowden)
52d44ee Pset release. (kohler)
```

The `git log` for Norton's local repository:

```
81f952e try mmap (norton)
52d44ee Pset release. (kohler)
```

At noon on October 11, their shared GitHub repository has this log:

```
d446e60 Increase cache size (snowden)
b677e85 use mmap on mmappable files (norton)
b46cfda Merge branch 'master' of code.seas.harvard.edu:~TheTrueHOOHA/cs61/TheTrueHOOHAs-cs61-psets.git
        (norton)
81f952e try mmap (norton)
3246d07 Save Greenwald's phone number (snowden)
8633fbd Start work on a direct-mapped cache (snowden)
52d44ee Pset release. (kohler)
```

QUESTION MISC-1C. Give an order for these commands that could have produced that log starting from the midnight October 10 state. You might not use every command, and you might use some commands more than once. Sample (incorrect) answer: "1 4 4 5 2."

1. snowden: `git commit -a`
2. snowden: `git push`
3. snowden: `git pull`
4. norton: `git commit -a`
5. norton: `git push`
6. norton: `git pull`

Show solution

- #2 (snowden push)
- [#5 (norton push—OPTIONAL; this push would fail)]
- #6 (norton pull) (We know that Snowden pushed first, and Norton pulled before pushing, because Norton committed the merge) [CIRCLE FOR 1D]
- [#4 (norton commit—OPTIONAL for the merge commit; the merge commit will happen automatically if there are no conflicts] [ALLOW CIRCLE FOR 1D]
- #4 (norton commit for b677e85)
- #5 (norton push)
- #3 (snowden pull—snowden pulls before committing because there is no merge)
- #1 (snowden commit for d446e60)
- #2 (snowden push)

Hide solution

Hide solution

QUESTION MISC-1D. In your answer to Part C, circle the step(s) where there might have been a merge conflict.

Show solution

(see above)

Hide solution

Hide solution

MISC-2. Debugging

QUESTION MISC-2A. Match each tool or technique with a debugging situation for which it is well suited. Produce the best overall match that uses each situation exactly once.

1. strace	A. Investigating segmentation faults
2. gdb	B. Finding memory leaks
3. valgrind --tool=memcheck	C. Checking your assumptions and verifying invariants
4. printf statements	D. Discovering I/O patterns
5. assert	E. Displaying program state

Show solution

1—D, 2—A, 3—B, 4—E, 5—C

Hide solution

Hide solution

MISC-3. Pot Pourri

QUESTION MISC-3A. What does the following instruction place in %eax?

```
sarl $31, %eax
```

Show solution

It fills eax with the sign bit of eax (i.e., all 0's or all 1's)

Hide solution

Hide solution

QUESTION MISC-3B. True/False: A direct-mapped cache with N slots can handle any reference string with < N distinct addresses with no misses except for compulsory misses.

Show solution

False

Hide solution

Hide solution

QUESTION MISC-3C. What is 1 (binary) TB in hexadecimal?

Show solution

1 TB = 2^40 = 1 followed by 40 zeros: so those 0's turn into the 10 hex 0's preceded by a 1: 0x10000000000

Hide solution

Hide solution

QUESTION MISC-3D. Write the answer to the following in hexadecimal:

```
0xabcd + 12
```

Show solution

12 = 0xC; 0xD + 0xC = (25 = 0x19), so the answer is 0xABD9

Hide solution

Hide solution

QUESTION MISC-3E. True/False: The garbage collector we discussed is conservative, because it only runs when we tell it to.

Show solution

False (conservative because it never reclaims something it shouldn't, but might not reclaim things it could).

Hide solution

Hide solution

QUESTION MISC-3F. True/False: Given the definition int array[10] the following two expressions mean the same thing: `&array[4] and array

- 4`.

Show solution

True

Hide solution

Hide solution

QUESTION MISC-3G. Using the matrix multiply from lecture 12, in what order should you iterate over the indices i, j, and k to achieve the best performance.

Show solution

ikj

Hide solution

Hide solution

QUESTION MISC-3H. True/False: fopen, fread, fwrite, and fclose are system calls.

Show solution

> False; they are calls to the standard IO library.
>
> Hide solution
>
> Hide solution

QUESTION MISC-3I. Which do you expect to be faster on a modern Linux OS, insertion sorting into a linked list of 1000 elements or into an array of 1000 elements?

Show solution

> The array.
>
> Hide solution
>
> Hide solution

QUESTION MISC-3J. What does the hardware do differently when adding signed versus unsigned numbers?

Show solution

> Nothing
>
> Hide solution
>
> Hide solution

MISC-4. Debugging

In the following short-answer questions, you have access to five debugging tools: `top`, `strace`, `gdb`, sanitizers, and `man`. You can't change program source code or use other tools. Answer the questions briefly (a couple sentences at most).

QUESTION MISC-4A. You are given a program that appears to "get stuck" when run. How would you distinguish whether the program blocked forever (e.g., made a system call that never returned) or entered an infinite loop?

Show solution

You can use `top`: does it report the process is using 100% of the CPU?

You can use `strace`: is the last thing in the strace an incomplete system call?

Hide solution

Hide solution

QUESTION MISC-4B. You are given a program that uses more memory while running than you expect. How would you tell whether the program leaks memory?

Show solution

Compile with a leak sanitizer and check if it reports any memory leaks when the program exits.

Hide solution

Hide solution

QUESTION MISC-4C. You are given a program that produces weird answers. How would you check if it invoked undefined behavior?

Show solution

Sanitizers.

Hide solution

Hide solution

QUESTION MISC-4D. You are given a program that blocks forever. How would you tell where the program blocked (which function called the blocking system call)?

Show solution

Run it under `gdb`. When it blocks, hit Ctrl-C and then enter `backtrace`/`bt` to get a backtrace.

Or use `strace`.

Hide solution

Hide solution

QUESTION MISC-4E. You are given a program that takes a long time to produce a result. How would you tell whether the program was using system calls unintelligently?

Show solution

> Run it under `strace` and look for stupidity, such as many system calls that report errors, many system calls that are redundant, lots of `read`s that return short counts, etc.
>
> Hide solution
>
> Hide solution

QUESTION MISC-4F. You are given a program that exits with a system call error, but doesn't explain what happened in detail. How would you find what error condition occurred and understand the conditions that could cause that error?

Show solution

> Run it under `strace` to find the error condition: look for a system call that returned the error. Then use `man` on that system call and read about the error (the `errno` description).
>
> Hide solution
>
> Hide solution

MISC-5. Miscellany

QUESTION MISC-5A. True or false in conventional Unix systems?

1. File descriptors are often used to communicate among processes on the same machine.
2. File descriptors are often used to communicate among processes on different machines.
3. File descriptors are often used to communicate with persistent storage.
4. File descriptors are often used to access primary memory.
5. File descriptors are often used to create child processes.

Show solution

> 1, 2, and 3 are true.
>
> Hide solution
>
> Hide solution

QUESTION MISC-5B. Match each numbered process isolation feature with the lettered hardware feature that helps enforce it. Use each hardware feature once (make the best match you can).

1. Protected control transfer (processes can transfer control to the kernel only at defined entry points)

2. Memory protection (one process cannot modify another process's memory)
3. Interrupt protection (only the kernel can disable interrupts)
4. CPU protection (the kernel always regains control of the CPU eventually)

A. Traps
B. Privileged mode (dangerous instructions fault unless the CPU is in privileged mode)
C. Timer interrupts
D. Page tables

Show solution

> 1—A, 2—D, 3—B, 4—C
>
> Hide solution
>
> Hide solution

The remaining questions refer to the following lines of code.

```
1.  close(fd);
2.  connect(fd, sockaddr, socklen);
3.  listen(fd);
4.  mmap(nullptr, 4096, PROT_READ, MAP_SHARED, fd, 0);
5.  read(fd, buf, 4096);
6.  write(fd, buf, 4096);
```

QUESTION MISC-5C. If a program executes the following line without error, which of those lines could be executed next without error? List all numbers that apply.

```
fd = open("/home/cs61user/cs61-psets/pset6/pong61.c", O_RDWR);
```

Show solution

> 1, 4, 5, 6
>
> Hide solution
>
> Hide solution

QUESTION MISC-5D. If a program executes the following line without error, which lines could be executed next without error? List all numbers that apply.

```
fd = socket(AF_INET, SOCK_STREAM, 0);
```

Show solution

1, 2, 3

Hide solution

Hide solution

QUESTION MISC-5E. If a program executes the following lines without error, which lines could be executed next without error? List all numbers that apply.

```
pipe(pipefd);
fd = pipefd[0];
```

Show solution

1, 5

Hide solution

Hide solution

MISC-6. More Miscellany

QUESTION MISC-6A. True or false: Any C++ arithmetic operation has a well-defined result.

Show solution

False: signed integer overflow is undefined.

Hide solution

Hide solution

QUESTION MISC-6B. True or false: Any x86-64 processor instruction has a well-defined result.

Show solution

True

Hide solution

Hide solution

QUESTION MISC-6C. True or false: By executing a trap instruction, a process can force an operating system kernel to execute arbitrary code.

Show solution

False (unless the kernel is really badly written!)

Hide solution

Hide solution

QUESTION MISC-6D. True or false: By manipulating process memory and registers, an operating system kernel can force a process to execute arbitrary instructions.

Show solution

True: the OS kernel has full system privilege.

Hide solution

Hide solution

QUESTION MISC-6E. True or false: All signals are sent explicitly via the `kill()` system call.

Show solution

False—some are sent for other reasons, such as the user hitting Control-C or a child process exiting.

Hide solution

Hide solution

QUESTION MISC-6F. True or false: An operating system's buffer cache is generally fully associative.

Show solution

True

Hide solution

Hide solution

QUESTION MISC-6G. True or false: The least-recently-used eviction policy is more useful for very large files that are read sequentially than it is for stacks.

Show solution

> False
>
> Hide solution
>
> Hide solution

QUESTION MISC-6H. True or false: Making a cache bigger can *lower* its hit rate for a given workload.

Show solution

> True—that's Bélády's anomaly.
>
> Hide solution
>
> Hide solution

QUESTION MISC-6I. True or false: x86-64 processor caches are coherent (i.e., always appear to contain the most up-to-date values).

Show solution

> True
>
> Hide solution
>
> Hide solution

QUESTION MISC-6J. True or false: A socket file descriptor supports either reading or writing, but not both.

Show solution

> False; it supports both
>
> Hide solution
>
> Hide solution

MISC-7. Pot Pourri

Parts A-D pertain to the data structures and hexdump output shown here.

```c
struct x {
    unsigned long ul;
    unsigned short us;
    unsigned char uc;
} *sp;
```

```
// Hexdump output of some program running on the appliance
08c1b008  e9 11 cf d0 0d d0 3f f3  63 61 74 00 0d f0 fe ca  |......?.cat.....|
08c1b018  5e ea 15 0d de c0 ad de                           |^.......|
```

You are told that `sp` = 0x08c1b008.

QUESTION MISC-7A. What is the value (in hex) of `sp->ul`?

Show solution

0xd0cf11e9

Hide solution

Hide solution

QUESTION MISC-7B. What is the value (in hex) of `sp->uc`?

Show solution

0x3f

Hide solution

Hide solution

QUESTION MISC-7C. At what address will you find the string "cat"?

Show solution

0x08c1b010

Hide solution

Hide solution

QUESTION MISC-7D. If the bytes after the string "cat" comprise an array of 3 integers, what is the value (in hex) of the integer at index 1 of that array?

Show solution

0x0d15ea5e

Hide solution

Hide solution

QUESTION MISC-7E. What is the following binary value expressed in hexadecimal: 01011010?

Show solution

0x5a

Hide solution

Hide solution

QUESTION MISC-7F. What is the value of the hex number 0x7FF in decimal?

Show solution

255 + 7*256 == 8*256 – 1 == 2*4*256 – 1 == 2*1024 – 1 == 2047

Hide solution

Hide solution

QUESTION MISC-7G. Is 0x98765432 a valid return from malloc?

Show solution

No, because it isn't aligned properly—malloc will always return a pointer whose alignment could work for any basic type, which on x86-64, means the last digit must be 0.

Hide solution

Hide solution

QUESTION MISC-7H. What is the minimum number of x86 instruction bytes you need to write an infinite loop?

Show solution

Two bytes: 0xeb 0xfe

Hide solution

Hide solution

QUESTION MISC-7I. True or False: Every declaration in C++ code allocates space for an object.

Show solution

False. Extern declarations, such as function declarations or `extern int x;`, don't allocate space.

Hide solution

Hide solution

QUESTION MISC-7J. True or False: Processes cannot share physical memory in WeensyOS.

Show solution

False; after step 5, child processes share read-only physical memory with their parents.

Hide solution

Hide solution

For parts K–O, assume we are running on the appliance and we initialize `ival`, `p`, and `q` as shown below. Write the value of the expression—you may express the values in hex if that's simpler, just be sure to prefix them with 0x to make it clear that you are doing so. For True/False questions, there is no need to correct or provide a counterexample for any statements that are false.

```
int ival[4] = {0x12345678, 0x9ABCDEF0, 0x13579BDF, 0x2468ACE0};
int* p = &ival[0];
int* q = &ival[3];
int* x = p + 1;
char* cp = (char*) (q - 2);
```

QUESTION MISC-7K. `q - p`

Show solution

3

Hide solution

Hide solution

QUESTION MISC-7L. `((char*) q - (char*) p)`

Show solution

12

Hide solution

Hide solution

QUESTION MISC-7M. `x - p`

Show solution

1

Hide solution

Hide solution

QUESTION MISC-7N. `*((short*) ((char*) x + 2))`

Show solution

0x9ABC

Hide solution

Hide solution

QUESTION MISC-7O. `*cp`

Show solution

0xF0

Hide solution

Hide solution

QUESTION MISC-7P. What system call allows you to block on a collection of file descriptors?

Show solution

`select` (also `poll`, `pselect`, `epoll`, …)

Hide solution

Hide solution

QUESTION MISC-7Q. What system call creates a communication channel that can only be used among related processes?

Show solution

`pipe`

Hide solution

Hide solution

QUESTION MISC-7R. ⚠️ What system call can change the attributes of a file descriptor so you can poll on it rather than block?

Show solution

`fcntl`

Hide solution

Hide solution

QUESTION MISC-7S. What system call produces a file descriptor on which a server can exchange messages with a client?

Show solution

`socket` (or `accept`)

Hide solution

Hide solution

QUESTION MISC-7T. True or False: A program and a process are the same thing.

Show solution

False

Hide solution

Hide solution

MISC-8. CS61 in Real Life

QUESTION MISC-8A. The CS61 Staff have built a jet (the NightmareLiner) modeled on the Boeing Dreamliner. Unfortunately, they modeled it just a bit too closely on the Dreamliner, which needs to be rebooted periodically to avoid failure. In the case of the NightmareLiner, it needs to be rebooted approximately every 16 days. Your job is to use what you've learned in CS61 about data representation to hypothesize why.

Hint: There are 86,400,000 ms in a day. 86,400,000 is between 2^{26} and 2^{27}.

Show solution

OK, 16 is 2^4 and 27+4 is 31 -- it looks to me like the have a signed 32-bit number somewhere and if they don't reboot, it overflows.

Hide solution

Hide solution

Google recently discovered (and reported) a bug in the GNU libc implementation of `getaddrinfo`. This function can perform RPC calls, which involve sending and receiving messages. In some cases, `getaddrinfo` failed to check that a received message could fit inside a buffer variable located on the stack (2048 bytes).

QUESTION MISC-8B. True or false: This flaw means `getaddrinfo` will always execute undefined behavior.

Show solution

False: it only executes undefined behavior if the received message exceeds the buffer size of 2048.

Hide solution

Hide solution

QUESTION MISC-8C. Give an example of a message that will cause `getaddrinfo` to exhibit undefined behavior.

Show solution

A message larger than the buffer (2048 bytes).

Hide solution

Hide solution

QUESTION MISC-8D. Briefly describe the contents of a message that would cause the `getaddrinfo` function to return to address 0x400012988 rather than to its caller.

Show solution

In the message, the value 0x400012988 should appear beyond the 2048-byte limit of the buffer so that it ends up overwriting the return value on the stack (for example, a message that is 4096 bytes long, and the second half of the message contains repeated instances of 0x400012988).

Hide solution

Hide solution

QUESTION MISC-8E. This code used to appear in the Linux kernel:

```
1. struct tun_struct *tun = ...;   // This is a valid assignment;
2. struct sock *sk = tun->sk;
3. if (!tun)
4.     return POLLERR;             // This is an error return
```

The compiler removed lines 3 and 4. Why was that a valid thing for the compiler to do?

Show solution

Dereferencing a null pointer is undefined behavior. Since tun is dereferenced in line 2, the compiler assumes that it cannot be null; therefore it is free to remove the check in line 3 and the accompanying code in line 4.

Hide solution

Hide solution

MISC-9. Miscellany

QUESTION MISC-9A. Name the property that implies a process cannot cause the kernel to execute code at an arbitrary address.

Show solution

Kernel isolation. "Process isolation" is certainly acceptable too.

Hide solution

Hide solution

QUESTION MISC-9B. True or false: It's safe to call any C library function from a signal handler.

Show solution

False

Hide solution

Hide solution