

Name (Last, First): Answer Key

This practice exam covers the exercises of Weeks 6 and 7, 20a_asm_bridge through 33k_paging. It consists of 5 questions on 24 pages; be sure you have the entire exam before starting. The point value of each question is indicated at its beginning; the entire exam has 100 points. Individual parts of a multi-part question are generally assigned approximately the same point value; exceptions are noted.

This exam is closed book. The one reference you may use is the printed course Cheatsheet. No electronic devices of any kind. You may not share material with classmates during the exam.

You are not expected to have memorized any assembly instruction, address, or bit layout, or any of the exercises' code. Each question prints what it uses.

Be concise and clearly indicate your answer. Presentation and simplicity of your answers may affect your grade. Answer each question in the space following the question. If you find it necessary to continue an answer elsewhere, indicate clearly the location of its continuation and label its continuation with the question number and subpart if appropriate.

You should read through all the questions first, then pace yourself.

If you need to ask a question during the exam, please raise your hand and I will come to your desk. Do not come to me.

The questions begin on the next page.

Problem	Possible	Score
1	20	
2	20	
3	20	
4	20	
5	20	
Total	100	

1. (_____/20 points)

Short Questions

(a) (5 points) Exercise 21r reaches a device's registers through a raw pointer. `base.add(offset)` is the address `offset` bytes past `base`. These are three of its functions:

```
pub const THR: usize = 0;    // Transmit Holding Register: a byte stored here is sent
pub const LSR: usize = 5;    // Line Status Register: the state of the device

pub unsafe fn write_reg(base: *mut u8, offset: usize, value: u8) {
    core::ptr::write_volatile(base.add(offset), value)
}

pub unsafe fn read_reg(base: *const u8, offset: usize) -> u8 {
    core::ptr::read_volatile(base.add(offset))
}

pub fn write_reg_checked(regs: &mut [u8], offset: usize, value: u8) -> bool {
    if offset >= regs.len() {
        return false;
    }
    unsafe { write_reg(regs.as_mut_ptr(), offset, value); }
    true
}
```

A test uses an 8-byte array in place of the device:

```
let mut device = [0u8; 8];
let ok = write_reg_checked(&mut device, 9, b'!'); // 1
let base = device.as_mut_ptr();                  // 2
write_reg(base, THR, b'K');                        // 3
unsafe { write_reg(base, 9, b'!') };               // 4
let v = unsafe { read_reg(base, LSR) };            // 5
```

- (i) Exactly one of the numbered lines does not compile. Which one, and what does the compiler object to?
- (ii) Line 1 passes an offset that is past the end of the array, and line 2 makes a raw pointer. Neither line needs `unsafe`. Why not? What value does `ok` get?
- (iii) Line 4 compiles. What does it do when it runs? `write_reg` is declared `unsafe fn`: what does that say about who is responsible for what line 4 does?

(i) Line 3. `write_reg` is an `unsafe fn`, and this call is not inside an `unsafe block`.

(E0133: call to unsafe function requires unsafe function or block)

(ii) Line 1: `write_reg_checked` is a safe fn. It checks `9 >= 8` itself and returns before any store, so `ok = false` and device is untouched. Line 2: making a raw pointer only computes an address. Nothing is loaded or stored until someone uses it.

(iii) It stores `!` at `base + 9`, one byte past the end of the 8-byte array, on top of whatever sits next to it. No error and no panic: undefined behavior. “`unsafe fn`” means the caller promises that `base + offset` is a real, writable byte. The `unsafe block` on line 4 made that promise, and it was false. `write_reg` did exactly what it was told.

(b) (5 points) The UART sets bit 5 of its Line Status Register, `LSR_TX_IDLE`, when it is ready to accept another byte. A student writes a `putc` that waits for that bit. It uses plain pointer accesses, where `base` is the UART's address and `THR` and `LSR` are the offsets from part (a):

```
pub const LSR_TX_IDLE: u8 = 1 << 5;

pub unsafe fn putc_wait(base: *mut u8, c: u8) {
    while *base.add(LSR) & LSR_TX_IDLE == 0 {}    // A: wait until the UART is ready
    *base.add(THR) = c;                            // B: send the byte
}
```

It passes every test. The tests point `base` at an array, and they are built without optimization.

- (i) An optimizing compiler assumes that memory changes only when the program stores to it. Given that, what may it do with the load on line A? What may it do with the store on line B when the kernel runs `putc_wait(base, b'h')` and then `putc_wait(base, b'i')`? For each, say what a person watching the terminal would see.
- (ii) Rewrite lines A and B so the compiler may do neither. Use `read_reg` and `write_reg` from part (a).
- (iii) 21r's `copy_bytes` copies one buffer in RAM to another with `*dst.add(i) = *src.add(i)`. Should it use volatile accesses as well? Why or why not?

(i) Line A: nothing in the loop stores to the LSR, so the compiler may load it once, before the loop (hoisting). If the UART was not ready at that instant, the loop becomes “jump to myself”: the kernel hangs, and the terminal shows nothing more, even after the UART is ready.

Line B: the store of 'h' is followed by another store to the same address, with no load in between. That is a dead store, so it may be deleted. The terminal shows i instead of hi.

(rustc -O really does both: one load, and a single store of 'i'.)

(ii)

```
while read_reg(base, LSR) & LSR_TX_IDLE == 0 {}    // A
write_reg(base, THR, c);                            // B
```

volatile = do this access exactly as written, every time.

(iii) No. A buffer in RAM changes only when the program stores to it, so the compiler's assumption is true there, and it is welcome to make the copy fast. volatile is for device registers, which change (and act) on their own.

(c) (5 points) This is the kernel of exercise 30k, with its lines numbered:

```

1  #![no_std]
2  #![no_main]
3
4  use core::panic::PanicInfo;
5
6  #[panic_handler]
7  fn panic(_info: &PanicInfo) -> ! {
8      loop {}
9  }
10
11 #[no_mangle]
12 pub extern "C" fn _entry() -> ! {
13     loop {}
14 }
```

Each change in the table below is made to this file on its own, and the kernel is rebuilt. Each build fails with one of these five messages:

- A. error[E0601]: ``main` function not found in crate `rv6``
- B. error: cannot find macro ``println`` in this scope
- C. error[E0308]: mismatched types: expected ``!``, found ``()``
- D. error[E0463]: can't find crate for ``std``
- E. error: ``#[panic_handler]` function required, but not found`

(i) Fill in the table. Each message is used once.

Change	Message	Why, in a few words
Delete line 1	D	no OS here, so there is no std to link
Delete line 2	A	rustc wants a main for a C runtime to call
Delete lines 6 to 9	E	a panic needs a handler, and std had it
Delete line 8	C	! = never returns; an empty body returns ()
Put <code>println!("hi");</code> above line 13	B	<code>println!</code> belongs to std (it needs an OS)

(ii) A student types line 1 as `#![no_std]`, without the `!`. The build fails, and one of its errors is message D again. Why does the attribute no longer do its job?

(ii) Without the `!` it is an outer attribute: it applies only to the one item that follows it. `#![...]` is an inner attribute: it applies to the whole crate, which is what `no_std` has to do.

(d) (5 points) Exercise 20a calls its assembly routine `bytecopy` through this declaration. `extern "C"` means that the arguments are passed in the registers `a0`, `a1`, `a2`, in the order they are declared.

```
extern "C" {
    pub fn bytecopy(dst: *mut u8, src: *const u8, n: usize);
}

bytecopy:
    beqz a2, 2f          # a0 = dst, a1 = src, a2 = n
                        # if a2 == 0, go to the ret
1:
    lb    t0, 0(a1)      # load one byte from the address in a1
    sb    t0, 0(a0)      # store it at the address in a0
    addi a0, a0, 1        # a0 = a0 + 1
    addi a1, a1, 1        # a1 = a1 + 1
    addi a2, a2, -1       # a2 = a2 - 1
    bnez a2, 1b          # if a2 != 0, go back to 1
2:
    ret
```

A student changes only the Rust declaration, to

```
pub fn bytecopy(dst: *mut u8, n: usize, src: *const u8);
```

and calls `unsafe { bytecopy(dst, 14, src) }`, where `dst` is `0x8010_3000` and `src` is `0x8010_2000`. Both are in RAM, which runs from `0x8000_0000` to `0x8800_0000`. There is no memory and no device at any address below `0x1000`.

- (i) The kernel builds with no error and no warning. Why can the compiler not catch this mistake?
- (ii) What do `a0`, `a1` and `a2` hold when `bytecopy`'s first instruction runs?
- (iii) What does the first `lb` try to do, and what happens? If that load had worked, how many bytes would the loop try to copy?
- (iv) Which keyword at the call marks that the caller is vouching for something the compiler cannot check?

(i) The `extern` block only declares a name and a signature. The body is assembly, which `rustc` never looks at, so there is nothing to check the signature against. The signature is a promise.

(ii) Rust follows its own declaration: `a0 = 0x8010_3000`, `a1 = 14`, `a2 = 0x8010_2000`.

(iii) The assembly still reads `a1` as `src`: it loads a byte from address 14, where there is nothing. The load faults, there is no trap handler yet, and the machine hangs (QEMU times out with no output). Had it worked: `n = a2 = 0x8010_2000`, about 2 billion bytes.

(iv) `unsafe`

2. (_____/20 points)

The Assembly Bridge

Exercise 20a calls RISC-V assembly from Rust. These two tables hold everything about RISC-V that this problem uses.

Register	What it is for	Across a call
ra	return address: call sets it, ret jumps to it	caller-saved
sp	stack pointer	callee-saved
a0–a7	arguments, in order; a0 also carries the return value	caller-saved
t0–t6	temporaries	caller-saved
s0–s11	saved registers	callee-saved

A function you call may destroy any *caller-saved* register. It must hand back every *callee-saved* register holding what it held at the call.

Instruction	Meaning
addi rd, rs, imm	rd = rs + imm , where imm is a constant
lb rd, off(rs)	load 1 byte from address rs + off into rd , <i>sign-extended</i> : the byte's top bit is copied into the 56 bits above it
sb rs2, off(rs1)	store the low byte of rs2 at address rs1 + off
ld rd, off(rs)	load 8 bytes (a 64-bit doubleword) from address rs + off into rd
sd rs2, off(rs1)	store the 8 bytes of rs2 at address rs1 + off
beqz rs, label	go to label if rs == 0
bnez rs, label	go to label if rs != 0
la rd, symbol	rd = the address of symbol
j label	go to label
call symbol	ra = the address of the next instruction; go to symbol
ret	go to the address in ra

1: and 2: are local labels. 1b means the nearest 1: looking backward, and 2f the nearest 2: looking forward. This is 20a's `bytecopy`:

```
# bytecopy(dst, src, n): copy n bytes from src to dst
bytecopy:                                # a0 = dst, a1 = src, a2 = n
    beqz a2, 2f                            # nothing to do
1:
    lb  t0, 0(a1)                          # load one byte from src
    sb  t0, 0(a0)                          # store it to dst
    addi a0, a0, 1                         # advance both pointers
    addi a1, a1, 1
    addi a2, a2, -1                        # one fewer byte to go
    bnez a2, 1b                            # loop while any remain
2:
    ret
```

(a) (6 points) Memory at address `0x8010_2000` holds the three bytes `52 C9 0A` (in hex), and the kernel calls `bytecopy(dst, src, 3)` with `dst = 0x8010_3000` and `src = 0x8010_2000`. Fill in the table with the value of each register every time the `bnez` is reached, just before it runs. Give `t0` in hex as a full 64-bit value.

bnez is reached	a0	a1	a2	t0
the first time	<code>0x8010_3001</code>	<code>0x8010_2001</code>	<code>2</code>	<code>0x0000_0000_0000_0052</code>
the second time	<code>0x8010_3002</code>	<code>0x8010_2002</code>	<code>1</code>	<code>0xFFFF_FFFF_FFFF_FFC9</code>
the third time	<code>0x8010_3003</code>	<code>0x8010_2003</code>	<code>0</code>	<code>0x0000_0000_0000_000A</code>

- Which bytes of memory has `bytecopy` changed when it returns, and what do they hold?
- One of the three loads sign-extended its byte into a value that is not the byte's value. Did that make the copy wrong? Explain, using the table of instructions.

(i) The three bytes at `0x8010_3000`, `0x8010_3001`, `0x8010_3002` now hold `52`, `C9`, `0A`. Nothing else: `a2` reached `0` after exactly three trips, so the byte at `0x8010_3003` was not touched.

(ii) No. `C9 = 1100 1001` has its top bit set, so `lb` copied that `1` into the upper 56 bits: `t0 = 0xFFFF_FFFF_FFFF_FFC9` (`-55`, not `201`).

But `sb` stores only the low byte of `t0`, which is still `C9`. The extra ones never reach memory, so the byte arrives unchanged.

(It would matter if the code added or compared `t0`. Then the load should be `lbu`, which fills with zeros.)

(b) (7 points) Here is `bytecopy` again. Change it into

```
pub fn wordcopy(dst: *mut u64, src: *const u64, n: usize);
```

which copies `n` 64-bit words (8 bytes each) from `src` to `dst`. Cross out what you change and write the replacement next to it.

```
bytecopy wordcopy:                                # a0 = dst, a1 = src, a2 = n
    beqz a2, 2f

1:
    lbld    t0, 0(a1)
    sbsd    t0, 0(a0)
    addi a0, a0, 18
    addi a1, a1, 18
    addi a2, a2, -1          ← no change: n counts words, not bytes
    bnez a2, 1b

2:
    ret
```

- (i) The kernel calls your `wordcopy(dst, src, 2)` with `dst = 0x8010_5000` and `src = 0x8010_4000`. What do `a0`, `a1` and `a2` hold when it returns?
- (ii) The Rust function that made that call was keeping values in `t1`, `a3` and `s1`. Going only by the calling convention, which of the three may it still rely on after the call? Why?
- (iii) `wordcopy` never moves `sp` and saves nothing on the stack. Give the two facts about `wordcopy` that make that legal.

(i) Two trips, each adding 8 to both pointers: `a0 = 0x8010_5010`, `a1 = 0x8010_4010`, `a2 = 0`.

(ii) Only `s1`: it is callee-saved, so any function must hand it back unchanged. `t1` and `a3` are caller-saved: a callee may destroy them, so the caller must not count on them. (This `wordcopy` happens to leave them alone, but the caller is not allowed to know that.)

(iii) 1. It is a leaf: it calls nothing, so `ra` still holds its caller's return address when `ret` runs. 2. It uses only a and t registers, which are caller-saved, so it owes nothing back.

(c) (7 points) 20a's `baby_swch` saves four registers in one `Ctx` and loads them from another. This is its code, the routine `co_entry` that the test switches to, and the test itself, shortened. The questions are on the next page.

```
#[repr(C)]
pub struct Ctx {
    pub ra: usize,      // at offset 0
    pub sp: usize,      // at offset 8
    pub s0: usize,      // at offset 16
    pub s1: usize,      // at offset 24
}

extern "C" {
    pub fn baby_swch(old: *mut Ctx, new: *const Ctx);
}

baby_swch:                # a0 = old, a1 = new
    sd    ra, 0(a0)
    sd    sp, 8(a0)
    sd    s0, 16(a0)
    sd    s1, 24(a0)
    ld    ra, 0(a1)
    ld    sp, 8(a1)
    ld    s0, 16(a1)
    ld    s1, 24(a1)
    ret

co_entry:                  # this instruction is at address 0x8000_099A
    la    t0, SEEN_S0
    sd    s0, 0(t0)        # record the s0 and s1 this code was started with
    la    t0, SEEN_S1
    sd    s1, 0(t0)
    la    a0, CO_CTX
    la    a1, MAIN_CTX
    call  baby_swch        # call 2
1:  j     1b              # this instruction is at address 0x8000_09CA

static mut MAIN_CTX: Ctx = Ctx::zero();    // all four fields start as 0
static mut CO_CTX: Ctx = Ctx::zero();

fn check_baby_swch() -> bool {
    unsafe {
        let ctx = addr_of_mut!(CO_CTX);
        (*ctx).ra = co_entry as *const () as usize;    // 0x8000_099A
        (*ctx).sp = /* the top of a second stack */;    // 0x8000_8298
        (*ctx).s0 = 0xC0FFEE;
        (*ctx).s1 = 0xBEEF;

        // addr_of_mut!(X) and addr_of!(X) are raw pointers to the static X
        asm::baby_swch(addr_of_mut!(MAIN_CTX), addr_of!(CO_CTX));    // call 1

        // The instruction after call 1 is at address 0x8000_1070. From there on,
        // the test checks that SEEN_S0 is 0xC0FFEE and SEEN_S1 is 0xBEEF.
        ...
    }
}
```

When `check_baby_swch` makes call 1, its registers hold `sp = 0x8000_7230`, `s0 = 0` and `s1 = 0`.

(c), continued. The code is on the previous page.

- (i) Fill in the table with the registers' values at each moment, and the address of the instruction that runs next.

Moment	ra	sp	s0	s1	Runs next
baby_swtnh is about to run its <code>ret</code> , the first time	<i>0x8000_099A</i>	<i>0x8000_8298</i>	<i>0xC0FFEE</i>	<i>0xBEEF</i>	<i>0x8000_099A</i>
baby_swtnh is about to run its <code>ret</code> , the second time	<i>0x8000_1070</i>	<i>0x8000_7230</i>	<i>0</i>	<i>0</i>	<i>0x8000_1070</i>

- (ii) What four values does `MAIN_CTX` hold at the first moment in the table? What does `CO_CTX.ra` hold at the second, and which instruction put that value in `ra`?
- (iii) A student adds a field, `pub id: usize`, at the top of `Ctx`, above `ra`, and changes nothing else. Everything still compiles, and `id` is 0 in both structs. Where does the first `ret` jump now, and what is in `sp` when it does? What does this say about the numbers 0, 8, 16 and 24 in the assembly?

(i) The first `ret` uses the `ra` just loaded from `CO_CTX`, so it lands in `co_entry`, on the second stack. The second `ret` uses the `ra` loaded from `MAIN_CTX`: back to the line after `call 1`, on the first stack, with `main`'s `s0` and `s1` (both 0) back as well.

(ii) `MAIN_CTX = ra 0x8000_1070, sp 0x8000_7230, s0 0, s1 0`: what `check_baby_swtnh` had at `call 1` (`ra` was set by that call).

`CO_CTX.ra = 0x8000_09CA`, the `j 1b`. `co_entry`'s own "call `baby_swtnh`" (`call 2`) set `ra` to the next instruction, and the first `sd` saved it.

(iii) The assembly still loads `ra` from offset 0, which is now `id`: `ra = 0`, so `ret` jumps to `address 0`. Nothing is there: the machine faults and hangs. `sp` comes from offset 8, which now holds what Rust stored in `.ra`: `sp = 0x8000_099A`, the address of `co_entry`.

The offsets are welded into the instructions. `#[repr(C)]` keeps the fields in declared order, but nobody checks that the assembly's numbers match the struct. Change one and you must change the other.

3. (_____/20 points)

From Reset to kmain

Exercise 31k boots the kernel. QEMU's virt machine, started with `-bios none`, copies the kernel into RAM and jumps to address `0x8000_0000`. Three entries of the machine's memory map:

0x0010_0000	the test finisher, a 32-bit register: storing 0x5555 to it powers QEMU off
0x1000_0000	the UART (serial port): a byte stored here is sent to the terminal
0x8000_0000	RAM: 128 MiB, up to 0x8800_0000

The linker script `kernel.ld` lays the kernel out in this order. The addresses are from one build of the 31k solution:

```
0x8000_0000 .text      code, .entry first: _entry at 0x8000_0000, kmain at 0x8000_0038
0x8000_1000 .rodata    constants; the string literals are its first bytes, up to 0x8000_1034
0x8000_1288 .data      statics that start with a value (8 bytes)
0x8000_1290 .bss       statics that start as zeros: STACK0 is all of it, 0x4000 bytes
0x8000_5290 end        the first address past the kernel image
```

This is `entry.rs`, with the address of each instruction, and `main.rs` with the first two instructions the compiler generated for `kmain`:

```
const STACK_SIZE: usize = 4096 * 4;                                // 0x4000

#[no_mangle]
static mut STACK0: [u8; STACK_SIZE] = [0; STACK_SIZE];

#[no_mangle]
#[link_section = ".entry"]
pub unsafe extern "C" fn _entry() -> ! {
    asm!(
        "la sp, {stack}",      // at 0x8000_0000
        "li t0, {size}",      // at 0x8000_0008
        "add sp, sp, t0",      // at 0x8000_000A
        "call kmain",         // at 0x8000_000C; the instruction after it is at 0x8000_0014
        stack = sym STACK0,    // {stack} is the symbol STACK0
        size = const STACK_SIZE, // {size} is the constant 0x4000
        options(noreturn),
    );
}

#[no_mangle]
pub extern "C" fn kmain() -> ! {
    uart::puts("\nr6 is booting...\n");
    uart::puts("OSLINGS:PASS\n");
    testdev::exit_success();
}

kmain:
    addi sp, sp, -16    # make a 16-byte frame
    sd  ra, 8(sp)      # save ra in the frame
    ...                # then the calls to puts
```

<code>la rd, symbol</code>	<code>rd</code> = the address of <code>symbol</code>	<code>add rd, rs1, rs2</code>	<code>rd</code> = <code>rs1</code> + <code>rs2</code>
<code>li rd, imm</code>	<code>rd</code> = the constant <code>imm</code>	<code>addi rd, rs, imm</code>	<code>rd</code> = <code>rs</code> + <code>imm</code>
<code>call symbol</code>	<code>ra</code> = the address of the next instruction; go to <code>symbol</code>		
<code>sd rs2, off(rs1)</code>	store the 8 bytes of <code>rs2</code> at address <code>rs1</code> + <code>off</code>		

(a) (7 points) `_entry` is the *boot trampoline*: the few instructions that must run before any Rust can. At reset, `sp`, `t0` and `ra` hold nothing useful. Fill in the table with each register's value after the instruction has run. Write ? where a register still holds nothing useful.

After this instruction	sp	t0	ra
<code>la sp, STACK0</code>	<code>0x8000_1290</code>	?	?
<code>li t0, 0x4000</code>	<code>0x8000_1290</code>	<code>0x4000</code>	?
<code>add sp, sp, t0</code>	<code>0x8000_5290</code>	<code>0x4000</code>	?
<code>call kmain</code>	<code>0x8000_5290</code>	<code>0x4000</code>	<code>0x8000_0014</code>
<code>kmain's addi sp, sp, -16</code>	<code>0x8000_5280</code>	<code>0x4000</code>	<code>0x8000_0014</code>

- (i) At what address does `kmain's sd ra, 8(sp)` store? Is that address inside `STACK0`? Show how you know.
- (ii) The value saved there is never loaded again. Why not?
- (iii) Why does `sp` have to start at the high end of `STACK0`, when `STACK0` names its low end?

(i) $sp + 8 = 0x8000_5280 + 8 = 0x8000_5288$. `STACK0` runs from `0x8000_1290` up to (not including) `0x8000_1290 + 0x4000 = 0x8000_5290`, so yes: it is the top 8 bytes of the array.

(ii) `kmain` never returns: its type is `!>`, and its last act is `exit_success`, which powers the machine off. The saved `ra` (`0x8000_0014`) is where a `ret` would go, and no `ret` ever runs. Nothing meaningful is at `0x8000_0014` anyway, which is why `_entry` says `noreturn`.

(iii) The stack grows down: every `call` subtracts from `sp` (the `addi` above) and stores above the new `sp`. Starting at the top keeps all of that inside the array. A symbol names where an array begins, its lowest address, so `_entry` has to add the size.

(b) (6 points) A student's `_entry` leaves out its third instruction, `add sp, sp, t0`. Nothing else changes, and `kmain` begins with the same two instructions.

- (i) What is `sp` when `kmain` starts? At what address does `kmain`'s `sd ra, 8(sp)` store now?
- (ii) Is that address inside `STACK0`? Using the layout two pages back, say what lives at that address. Then `kmain` calls `puts`, which calls `putc`, and each makes a stack frame of its own. What do those frames overwrite?
- (iii) Built and run under QEMU, this kernel prints `OSLINGS:PASS` and exits normally. Why did none of those stores cause a fault? Why is the passing test not evidence that this `_entry` is correct?

(i) `sp` is never raised: `sp = 0x8000_1290`, the bottom of `STACK0`. `kmain`'s `addi` makes it `0x8000_1280`, so `ra` is stored at `0x8000_1288`.

(ii) No. It is 8 bytes below `STACK0`: that address is `.data`, the kernel's initialized statics. Every deeper call subtracts more from `sp`, so the frames of `puts` and `putc` land lower still: in `.rodata`, the kernel's own constants.

`STACK0 0x8000_1290` ← `sp` starts here, and only goes down
`.data 0x8000_1288` ← `kmain`'s saved `ra`
`.rodata 0x8000_1000` ← the frames of `puts`, `putc`, ...

(iii) Nothing protects that memory. The MMU is off and the kernel runs in machine mode, so `.data` and `.rodata` are ordinary writable RAM: a store there just works. No fault, no message.

It passed by luck. In this build the frames reached down 240 bytes, to `0x8000_11A0`, and the strings it prints sit lower (they end at `0x8000_1034`), so they survived. A deeper call chain, a bigger frame, or a different link order, and the kernel overwrites data it is about to use. A test that checks the output cannot see where the stack was.

(c) (7 points) Once `kmain` runs, printing and powering off are stores to device addresses:

```
const UART0: *mut u8 = 0x1000_0000 as *mut u8;    const TEST_FINISHES: *mut u32 = 0x10_0000 as *mut u32;

pub fn putc(c: u8) {                               pub fn exit_success() -> ! {
    unsafe { write_volatile(UART0, c); }           unsafe { write_volatile(TEST_FINISHES, 0x5555); }
}                                                  loop {}
}

pub fn puts(s: &str) {
    for byte in s.bytes() {
        putc(byte);
    }
}
```

- (i) `kmain` runs `uart::puts("OK\n")`. The ASCII codes are `O = 0x4F`, `K = 0x4B` and `newline = 0x0A`. List the stores the CPU makes to the device, in order, giving the address and the value of each. They all go to one address: why does each store not simply replace the one before it?

Three 1-byte stores, all to 0x1000_0000: 0x4F, then 0x4B, then 0x0A.

That address is a device register, not memory. Each store hands the UART a byte to send; nothing is kept there to be replaced. (That is why they are volatile: plain stores to one address would look dead and be merged.)

- (ii) Using `exit_success` as your model, write `exit_failure`. It reports a failure code to the same register: the 32-bit value to store has `0x3333` in its low 16 bits and `code` in the 16 bits above them. (`code` as `u32` widens `code` to 32 bits.)

```
pub fn exit_failure(code: u16) -> ! {

    unsafe {
        write_volatile(TEST_FINISHES, 0x3333 | ((code as u32) << 16));
    }
    loop {}

}
```

- (iii) What 32-bit value does `exit_failure(2)` store? What does the compiler report if you leave out the `loop {}`, and why?

$2 \ll 16 = 0x0002_0000$, OR $0x3333 = 0x0002_3333$

Without the loop: `E0308`, mismatched types: expected ``!``, found ``()``. It promises never to return, and falling off the end would return.

4. (_____/20 points)

The Free List

This is the page allocator of exercise 32k:

```
pub const PGSIZE: usize = 4096;           // 0x1000

#[repr(C)]
struct Run {
    next: *mut Run,
}

static mut FREELIST: *mut Run = ptr::null_mut();

extern "C" {
    static end: u8;           // from the linker script: the first address past the kernel image
}

fn pgroundup(addr: usize) -> usize {
    (addr + PGSIZE - 1) & !(PGSIZE - 1)
}

pub unsafe fn init() {
    let start = &end as *const u8 as usize;    // the address of end
    free_range(start, PHYSTOP);
}

unsafe fn free_range(start: usize, stop: usize) {
    let mut p = pgroundup(start);
    while p + PGSIZE <= stop {
        kfree(p as *mut u8);
        p += PGSIZE;
    }
}

pub unsafe fn kfree(pa: *mut u8) {
    let r = pa as *mut Run;
    (*r).next = FREELIST;
    FREELIST = r;
}

pub unsafe fn kalloc() -> *mut u8 {
    let r = FREELIST;
    if !r.is_null() {
        FREELIST = (*r).next;
    }
    r as *mut u8
}
```

!x is bitwise NOT, and `ptr::null_mut()` is the null pointer. A `Run` is 8 bytes, so `(*r).next` is the first 8 bytes of the page that `r` points at.

To keep the numbers small, suppose this kernel runs on a machine with only 64 KiB of RAM, from `0x8000_0000` up to `PHYSTOP = 0x8001_0000`. The linker put `end` at `0x8000_C2A8`.

(a) (7 points) `kmain` calls `kalloc::init()`.

- (i) Compute `pgroundup(end)`, showing the add and the mask. Which pages does `free_range` pass to `kfree`, in order?

$0x8000_C2A8 + 0xFFF = 0x8000_D2A7$; $\& !0xFFF$ clears the low 12 bits $\rightarrow$

$0x8000_D000$

`kfree(0x8000_D000)`, `kfree(0x8000_E000)`, `kfree(0x8000_F000)`: then `p = 0x8001_0000` and `p + PGSIZE > stop`, so the loop ends. 3 pages.

- (ii) Fill in the state of the allocator when `init` returns. Write `null` for a null pointer.

FREELIST	first 8 bytes of page 0x8000_D000	first 8 bytes of page 0x8000_E000	first 8 bytes of page 0x8000_F000
<u>$0x8000_F000$</u>	<u><code>null</code></u>	<u>$0x8000_D000$</u>	<u>$0x8000_E000$</u>

- (iii) The kernel then runs the six lines in the table, in order. Give what each call returns, and `FREELIST` after it.

Line	returns	FREELIST afterwards
<code>let a = kalloc();</code>	<u>$0x8000_F000$</u>	<u>$0x8000_E000$</u>
<code>let b = kalloc();</code>	<u>$0x8000_E000$</u>	<u>$0x8000_D000$</u>
<code>kfree(a);</code>	(nothing)	<u>$0x8000_F000$</u>
<code>let c = kalloc();</code>	<u>$0x8000_F000$</u>	<u>$0x8000_D000$</u>
<code>let d = kalloc();</code>	<u>$0x8000_D000$</u>	<u><code>null</code></u>
<code>let e = kalloc();</code>	<u><code>null</code></u>	<u><code>null</code></u>

- (iv) After the last line, what do the first 8 bytes of page `0x8000_F000` hold? Nothing erased them when the page was handed out. Why does the allocator not care?

$0x8000_D000$, the link `kfree(a)` wrote there (the list was `F000` $\rightarrow$ `D000` then). It is stale: `kalloc` read it once, when it popped the page, and nothing on the list points at `F000` now, so the allocator never reads it again. The page's owner (c) may overwrite all 4096 bytes.

(b) (6 points) A student writes the last two lines of `kfree` in the other order:

```
pub unsafe fn kfree(pa: *mut u8) {
  let r = pa as *mut Run;
  FREELIST = r;
  (*r).next = FREELIST;
}
```

(i) Redo `init` on the 64 KiB machine with this `kfree`. Fill in the table.

FREELIST	first 8 bytes of page 0x8000_D000	first 8 bytes of page 0x8000_E000	first 8 bytes of page 0x8000_F000
0x8000_F000	0x8000_D000	0x8000_E000	0x8000_F000

(ii) What do the first two calls to `kalloc` return? Explain with a drawing of the list that `kalloc` follows.

(iii) The exercise's self-test makes these five checks, in order, and stops at the first that fails. Which one fails, and why do the ones before it pass?

- | | |
|--|---|
| 1. <code>a = kalloc()</code> is not null | [fail] <code>kalloc</code> returned null |
| 2. <code>a</code> is page-aligned and inside RAM | [fail] page is misaligned or outside RAM |
| 3. every byte of page <code>a</code> can be written and read back | [fail] page did not hold what we wrote |
| 4. <code>b = kalloc()</code> is not null, and <code>b != a</code> | [fail] second <code>kalloc</code> reused or failed |
| 5. after <code>kfree(b)</code> , the next <code>kalloc()</code> returns <code>b</code> again | [fail] <code>free/alloc</code> did not recycle the page |

(i) Each page ends up pointing at itself: `FREELIST` already names the page when the link is written, so the link records the page, not the old head.

(ii) 0x8000_F000, and 0x8000_F000 again.

`FREELIST` $\rightarrow$ `F000` $\rightarrow$ `F000` $\rightarrow$ `F000` $\rightarrow$... a one-page loop

`kalloc` sets `FREELIST = (*r).next`, which is `F000` itself, so the head never moves.

`D000` and `E000` are lost: nothing points at them.

(iii) Check 4: "second `kalloc` reused or failed", because `b == a`. Checks 1 to 3 pass because `F000` is a real page: not null, page-aligned, inside RAM, and writable. One allocation looks perfect; the bug needs two.

(c) (7 points) Back to the correct `kfree`.

- (i) Write `count_free`, which returns the number of pages on the free list without changing the list. Use `kalloc` as your model for the raw-pointer syntax.

```
pub unsafe fn count_free() -> usize {

    let mut n = 0;
    let mut r = FREELIST;      // a copy: FREELIST itself never changes
    while !r.is_null() {
        n += 1;
        r = (*r).next;
    }
    n
}

}
```

- (ii) What does your `count_free` return right after `init` on the 64 KiB machine of part (a)? What does it do on the list that the student's `kfree` built in part (b)?
- (iii) `kfree` returns nothing: it has no way to fail. Where does it find the memory to record that a page is free? Why does that matter most when the kernel is nearly out of memory?

(ii) Part (a): 3 ($F000 \rightarrow E000 \rightarrow D000 \rightarrow \text{null}$).

Part (b): it never returns. `F000`'s next is `F000`, so `r` is never null: `n` counts up forever. (A list with a loop in it has no end to find.)

(iii) In the page being freed. A free page holds nothing anyone needs, so its own first 8 bytes store the link to the next free page: the list lives inside the memory it describes (an intrusive list).

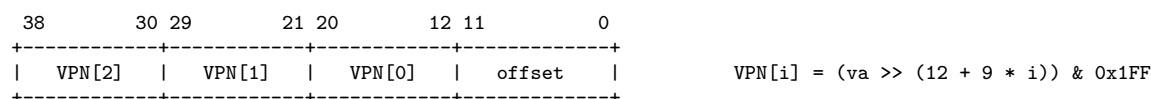
So `kfree` needs no memory of its own: no table to grow and nothing to allocate, and it cannot fail. That matters most when memory is short, because that is exactly when the kernel frees things. A free that could fail for lack of memory would fail just when it is needed.

(On the real 128 MiB machine, with end at `0x8000_6E38`, `count_free` returned `32761` after `init`.)

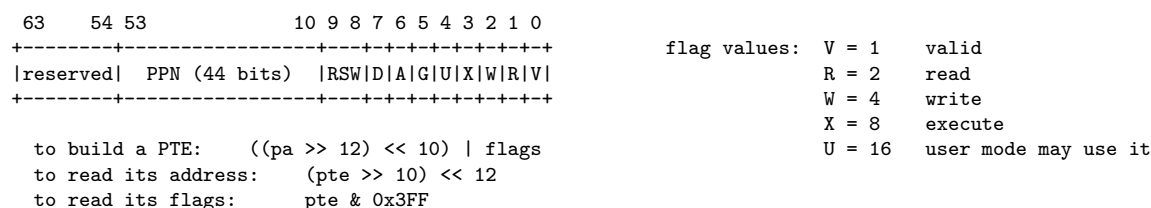
5. (_____/20 points)

Sv39 Page Tables

Exercise 33k builds the page tables that translate a virtual address into a physical one. An Sv39 virtual address is three 9-bit indices over a 12-bit offset:



A page table is one 4096-byte page holding 512 entries of 8 bytes each. VPN[2] picks an entry of the root table, VPN[1] an entry of a level-1 table, and VPN[0] an entry of a level-0 table. Each entry (a PTE) packs a physical page number (PPN) above ten bits of flags:



A valid PTE with R, W and X all clear is a *branch*: its address is the next table. A valid PTE with any of R, W, X set is a *leaf*: its address is the mapped page.

(a) (5 points) Work these by hand.

- (i) Split the virtual address 0x20_4060_5ABC into VPN[2], VPN[1], VPN[0] (in decimal) and the offset.
- (ii) Build the PTE for the page at physical address 0x8001_2000 with flags V, R and X.
- (iii) Decode the PTE 0x2000_4C17: its flags, its address, and whether it is a leaf or a branch. May a user program store to that page?
- (iv) table is a *mut Pte that points at a page table at 0x8040_7000, and a Pte is 8 bytes. What address is table.add(128)?

(i) offset = 0xABC. $va \gg 12 = 0x204_0605$, $\& 0xFF = 0x005 \rightarrow VPN[0] = 5$
 $va \gg 21 = 0x1_0203$, $\& 0xFF \rightarrow VPN[1] = 3$ $va \gg 30 = 0x81 \rightarrow$
 $VPN[2] = 129$

(ii) $pa \gg 12 = 0x8_0012$. $\ll 10$ (append 00, then $\times 4$) = $0x2000_4800$. OR
 0xB (V + R + X) $\rightarrow$ $0x2000_480B$

(iii) low 10 bits = $0x017 = 1_0111 = \underline{V R W U}$. $\gg 10 = 0x8_0013$, $\ll 12 =$
 $0x8001_3000$.

A leaf (R and W are set). Yes: W and U are both set.

(iv) .add counts 8-byte entries: $128 \times 8 = 0x400$ bytes past $0x8040_7000 \rightarrow$
 $0x8040_7400$

Parts (b) and (c) use this code: 33k's `vm.rs`, and the function its test uses to read the tables the way the MMU will. `PTE_V`, `PTE_R`, `PTE_W`, `PTE_X` and `PTE_U` are the flag values 1, 2, 4, 8 and 16.

```
pub struct Pte(pub usize); // one page-table entry
impl Pte {
    pub const fn new(pa: usize, flags: usize) -> Pte { Pte(((pa >> 12) << 10) | flags) }
    pub const fn pa(self) -> usize { (self.0 >> 10) << 12 }
    pub const fn flags(self) -> usize { self.0 & 0x3ff }
    pub const fn is_valid(self) -> bool { self.0 & PTE_V != 0 }
}

const fn px(level: usize, va: usize) -> usize { // VPN[level] of va
    (va >> (12 + level * 9)) & 0x1fff
}

// Return a pointer to the level-0 PTE for va. Where a table is missing: make one
// if alloc is true, and return null if it is false.
pub unsafe fn walk(mut table: *mut Pte, va: usize, alloc: bool) -> *mut Pte {
    let mut level = 2;
    while level > 0 {
        let pte = table.add(px(level, va));
        if (*pte).is_valid() {
            table = (*pte).pa() as *mut Pte;
        } else {
            if !alloc { return ptr::null_mut(); }
            let page = kalloc::kalloc();
            if page.is_null() { return ptr::null_mut(); }
            ptr::write_bytes(page, 0, PGSIZE); // fill the new table with zeros
            *pte = Pte::new(page as usize, PTE_V);
            table = page as *mut Pte;
        }
        level -= 1;
    }
    table.add(px(0, va))
}

// Map size bytes of virtual addresses starting at va to physical addresses starting at pa.
pub unsafe fn mappages(table: *mut Pte, va: usize, size: usize, pa: usize, perm: usize)
    -> Result<(), ()> {
    let mut a = pgrounddown(va); // round down to a page boundary
    let last = pgrounddown(va + size - 1);
    let mut pa = pa;
    loop {
        let pte = walk(table, a, true);
        if pte.is_null() {
            return Err(());
        }
        *pte = Pte::new(pa, perm | PTE_V);
        if a == last {
            break;
        }
        a += PGSIZE;
        pa += PGSIZE;
    }
    Ok(())
}

// What the MMU will do, in software.
unsafe fn translate(root: *mut Pte, va: usize) -> Option<usize> {
    let pte = walk(root, va, false);
    if pte.is_null() || !(*pte).is_valid() {
        return None;
    }
    Some((*pte).pa() | (va & (PGSIZE - 1)))
}
```

(b) (9 points) The root table is the page at 0x8040_0000, and every entry in it is zero. The allocator's free list is

FREELIST -> 0x8040_7000 -> 0x8040_2000 -> 0x8040_5000 -> 0x8040_9000 -> ...

As in Problem 4, `kalloc()` takes the page at the head of the list. The kernel makes two calls, with `root` pointing at the root table:

```
mappages(root, 0x1000_0000, PGSIZE, 0x1000_0000, PTE_R | PTE_W); // call 1: the UART's page
mappages(root, 0x1020_3000, 2 * PGSIZE, 0x8005_0000, PTE_R | PTE_X); // call 2: two pages of code
```

To save you some arithmetic, here are the indices of the three virtual pages:

Virtual page	VPN[2]	VPN[1]	VPN[0]
0x1000_0000	0	128	0
0x1020_3000	0	129	3
0x1020_4000	0	129	4

(i) The two calls write six PTEs. Fill in the blanks. The first row is done for you.

Call	In the table at	Index	Value written	Branch or leaf	It points at
1	0x8040_0000	0	0x2010_1C01	branch	0x8040_7000
1	0x8040_7000	128	0x2010_0801	branch	0x8040_2000
1	0x8040_2000	0	0x0400_0007	leaf	0x1000_0000
2	0x8040_7000	129	0x2010_1401	branch	0x8040_5000
2	0x8040_5000	3	0x2001_400B	leaf	0x8005_0000
2	0x8040_5000	4	0x2001_440B	leaf	0x8005_1000

(ii) How many pages did `kalloc` hand out during the two calls? Why did call 2 need fewer new tables than call 1, even though it maps more pages?

(i) Row 4: 0x8040_5000 >> 12 = 0x8_0405; << 10 → 0x2010_1400; OR V → 0x2010_1401.

Row 5: 0x8005_0000 >> 12 = 0x8_0050; << 10 → 0x2001_4000; OR (R + X + V = 0xB) → 0x2001_400B. Row 6 is the next page: PPN + 1, so + 0x400.

(ii) 3: 0x8040_7000 and 0x8040_2000 for call 1, and 0x8040_5000 for call 2. All three virtual pages have VPN[2] = 0, so call 2 found root[0] already valid and followed it to the level-1 table that call 1 made. Only the level-0 table for VPN[1] = 129 was missing, and both code pages share it.

(b), continued. Use the tables you filled in on the previous page.

- (iii) Translate the virtual address 0x1020_4ABC to a physical address by hand. Show the entry you use at each level and what you learn from it.

0x1020_4ABC is in page 0x1020_4000: VPN = 0, 129, 4, and the offset is 0xABC.

root 0x8040_0000, entry 0 = 0x2010_1C01: V only, a branch → table 0x8040_7000

0x8040_7000, entry 129 = 0x2010_1401: V only, a branch → table 0x8040_5000

0x8040_5000, entry 4 = 0x2001_440B: V R X, a leaf → page 0x8005_1000

pa = 0x8005_1000 | 0xABC = 0x8005_1ABC (same last three hex digits as the question)

- (iv) Neither of these virtual addresses was mapped: 0x1020_5000, whose indices are 0, 129, 5, and 0x1040_0000, whose indices are 0, 130, 0. For each, does walk(root, va, false) return null or a pointer to an entry? What does translate return, and which of the two tests in its if catches it?

0x1020_5000: both tables exist, so walk returns a pointer, to entry 5 of the table at 0x8040_5000. That entry is 0, so !(*pte).is_valid() catches it → None.

0x1040_0000: entry 130 of the level-1 table is 0, and alloc is false, so walk gives up and returns null. pte.is_null() catches it → None.

Two ways to be unmapped: the table is missing, or the table is there and the entry is empty. translate needs both tests.

- (v) The satp register holds the paging mode in bits 63 to 60, where 8 means Sv39, and the PPN of the root table in bits 43 to 0. What value in satp would tell the MMU to use this root table?

PPN = 0x8040_0000 >> 12 = 0x8_0400; mode 8 in the top hex digit →

0x8000_0000_0008_0400

(c) (6 points) The kernel must check any address that a user program hands it before using it.

- (i) Using `translate` as your model, write `walkaddr`. If the page that `va` is in is mapped *and* user mode may use it (`PTE_U`), it returns that page's physical address, without the offset. Otherwise it returns 0.

```
unsafe fn walkaddr(root: *mut Pte, va: usize) -> usize {

    let pte = walk(root, va, false);
    if pte.is_null() || !(*pte).is_valid() {
        return 0; // not mapped
    }
    if (*pte).flags() & PTE_U == 0 {
        return 0; // mapped, but not for user mode
    }
    (*pte).pa()

}
```

- (ii) What does your `walkaddr` return for `0x1020_4ABC` with the tables of part (b)? Would the MMU translate that address for the kernel itself?
- (iii) A student deletes the `ptr::write_bytes` line from `walk`. The page at `0x8040_5000` was used as a buffer before it was freed, and its 8-byte word at index 5 still holds `0x41`. The student's kernel makes the two calls of part (b). What does `translate(root, 0x1020_5000)` return now, and why? The same kernel passes every test on a freshly booted machine, where a page from `kalloc` holds zeros except for the free-list link in its first 8 bytes. Why does that link not cause the same trouble?

(ii) 0. The leaf is `V R X` with no `U`: mapped, but not for user mode. The MMU would translate it for the kernel (to `0x8005_1ABC`): “mapped” and “the user may touch it” are different questions.

(iii) Some(0). The page becomes the level-0 table with its old contents still in it. Entry 5 is `0x41 = 0100 0001`: bit 0 (`V`) is set, so `is_valid()` says yes, and `pa() = (0x41 >> 10) << 12 = 0`. An address nobody mapped now “translates”, to physical address 0.

On a fresh boot the only stale word is the link: the address of another page. A page address ends in 000, so its bit 0 is clear, and the entry reads as invalid, the same as a zero. The bug waits for a recycled page.

Continue your answers here if necessary.