Computer Science 162
David E. Culler
University of California, Berkeley
Alternate Final Exam
December 16, 2019

Name	
Student ID	
Login (`studentXXX`)	
TA Name and Section Time	
Name of Students to your Left and Right	
Name of Students in Front of You and Behind You	

This is a closed-book exam with two 2-sided handwritten pages of notes permitted. It is intended to be a 110 minute exam. You have 170 minutes to complete it. The number at the beginning of each question indicates the points for that question. Write all of your answers directly on this exam. Make your answers as concise as possible. If there is something in a question that you believe is open to interpretation, please raise your hand to request clarification. When told to open the exam, put your Student ID on every page and check that you have them all. The final pages are for reference.

By my signature below, I swear that this exam is my own work. I have not obtained answers or partial answers from anyone, and I promise not to discuss this exam with anyone prior to completion of the regular exam.

X _____

Grade Table (for instructor use only)

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	1	24	20	10	9	12	14	18	0	108
Score:										

1. (1 point) **Miscellaneous Questions**

   (a) (1 point) Write your Student ID on every page of the exam.

   (b) (1 bonus point) You will receive one point for completing the CS 162 Exit Survey.

   (c) (1 bonus point) The winners of the Pintos Music Contest will each receive one point.

2. (24 points) **Operating System Concepts: True/False**

Choose True or False for the questions below. You do not need to justify your answers.

(a) (1 point) If a process issues a `read` system call on an open file, the position pointer associated with the file is advanced by the number of bytes read into the buffer, even if the EOF is reached.

√ True ◯ False

(b) (1 point) If a fair scheduler, like Linux CFS, strives to meet a responsiveness target T by giving each task a quantum of T divided by the number of tasks, then the minimum granularity may cause it to exceed the target latency with a large number of tasks.

√ True ◯ False

(c) (1 point) Increasing the size of a cache will decrease its hit rate under any workload and eviction policy.

◯ True √ False

(d) (1 point) A single memory access may result in a TLB miss and a page fault.

√ True ◯ False

(e) (1 point) As a system's workload is increased past the half-power point, causing it to approach 100% utilization, response time typically increases due to queuing delays.

√ True ◯ False

(f) (1 point) LRU is an appropriate policy to evict page frames for demand paging.

◯ True √ False

(g) (1 point) In the x86 architecture, the MMU clears the use bit of page table entries that have not been accessed in a long time.

◯ True √ False

(h) (1 point) A device driver may interact with I/O devices by reading and writing to memory addresses reserved for memory-mapped I/O.

√ True ◯ False

(i) (1 point) In a typical file system, most files are small.

√ True ◯ False

(j) (1 point) The compiled kernel code dynamically assigns the inumber for the file system root (`/`) inode.

◯ True √ False

(k) (1 point) In the FAT file system, there is no limit on how large a file can be, except for the size of the underlying storage device. (Ignore any limits due to a fixed-size "file length" field.)

√ True ◯ False

(l) (1 point) In a Unix FFS-style inode-based file system, there is no limit on how large a file can be, except for the size of the underlying storage device. (Ignore any limits due to a fixed-size "file length" field.)

◯ True √ False

(m) (1 point) In the Windows NTFS file system sequential allocation on disk is enabled by First Fit Block Allocation.

◯ True √ False

(n) (1 point) The low-level file API (`read`, `write`) is accelerated by the kernel's buffer cache.
√ True ◯ False

(o) (1 point) The Transmission Control Protocol (TCP) allows each endpoint of a connection to have multiple outstanding unACKed packets at a time.
√ True ◯ False

(p) (1 point) In the Andrew File System (AFS), a user's modifications to a file *might* become visible to other users in the system as soon as the `close` operation completes.
√ True ◯ False

(q) (1 point) A group of machines can use the Two-Phase Commit (2PC) protocol to guarantee that they will perform a task atomically, i.e., either all or none.
√ True ◯ False

(r) (1 point) For consistent hashing each node chooses a random value in the range of a hash function on the key and shares it with the other nodes to partition the key space with no particular pattern.
√ True ◯ False

(s) (1 point) The virtual machine monitor maintains the shadow page table by setting the guest page tables as read only, so that writes to the guest page table trap and the VMM is able to update both the guest and shadow page tables.
√ True ◯ False

(t) Containers handle package dependences for collections of processes, whereas Kubernetes introduces pods as a mechanism for co-locating a related set of containers on a set of (virtualized) resources.
√ True ◯ False

(u) (1 point) Neither containers nor virtual machines isolate a collection of processes from all the rest of the processes.
◯ True √ False

(v) (1 point) The socket abstraction is dictated by the transport layer of the network stack to allow a client and a server to communicate over HTTP.
◯ True √ False

(w) (1 point) The total number of virtual pages is always greater than or equal to the total number of physical page frames.
◯ True √ False

(x) (1 point) A distributed key/value store can use a master directory server to direct client put/get requests to at most one of multiple replicas.
◯ True √ False

3. (20 points) **Operating System Concepts: Short Answer**

There are 24 points of questions below. You must answer 20 points worth of questions. Clearly cross out or leave blank the ones that should not be graded.

Fill in the blanks correctly for the questions below. You do not need to provide justifications.

(a) (1 point) A process may request the operating system to perform a privileged action on its behalf by issuing a/an _____system call_____.

(b) (1 point) A scheduling algorithm that provides proportional shares is _____Stride, Lottery, or Completely Fair Scheduling_____.

(c) (1 point) The subset of its virtual address space that a process is actively using over a given time interval is called a/an _____working set_____.

(d) (1 point) To evict a page to disk, the operating system finds which page table entries map to pages that are _____not recently used_____.

(e) (1 point) A transaction is a/an _____atomic_____ sequence of reads/writes.

(f) (1 point) A group of n machines can maintain safety and liveness in the face of fail-stop failures as long as at most _____a minority of (or $\lfloor \frac{n-1}{2} \rfloor$)_____ node(s) is/are faulty.

(g) (2 points) Components of a file system

i. (1 point) What are the four components of a file system?

Solution: Directory Structure, Index, Storage Blocks, and Free Blocks

ii. (1 point) Which component(s) of a file system might the kernel access to handle a `write` system call? Assume the file descriptor corresponds to a regular file (not an I/O device, directory, or soft link).

Solution: Index, Storage Blocks, and Free Blocks

(h) (2 points) In the Redo Logging scheme discussed in class, which logged operations must be performed during recovery and which must be discarded?

Solution: Operations for committed transactions must be performed. Those for uncommitted transactions are discarded.

(i) (2 points) The concept of *dispatch*—passing a request to the appropriate request handler for processing—arises in a variety of systems. Give two examples of dispatch *in software systems* that we studied in class.

Solution: Any two of the following would receive full credit: interrupt vector table, system call handler, RPC server (or HTTP server), and Linux VFS. Other examples that we didn't cover in class are language polymorphism (e.g., vtables), opcode dispatch in bytecode interpreters, and protocol dispatch in network stacks.

(j) (3 points) Before you build a buffer cache, you run measurements and find that on average, it takes 10 ms to perform each file operation. With the buffer cache you can perform an operation in 1 ms when it does not have to go to disk. At most what fraction of the operations can go to disk for the average operation time to be below 2 ms?

Solution: $10x + (1 - x) < 2 \implies x < \frac{1}{9}$. We also accepted $\frac{1}{10}$, as the problem can also be read as including the 1 ms on cache misses. We stated a clarification for this at the exam, but ultimately we decided to accept both answers.

(k) (1 point) How will your buffer cache influence the I/O queuing delays experienced by applications?

Solution: It will decrease them because the bandwidth is higher and the latency lower.

(l) (2 points) In distributed file systems, like AFS and NFS, which component is responsible for providing consistency?

Solution: The server is responsible for notifying clients when data they may have cached has been overwritten by another client.

(m) (2 points) When either the client or server closes a connection socket, how is the other party notified? How do they reach agreement to close?

Solution: One notifies the other with a FIN message. They agree by each sending a FIN-ACK.

(n) (2 points) In order for complex data objects to be passed as arguments or results in an RPC over a socket to a possibly remote machine, what needs to be done to the data?

Solution: It needs to be serialized into a canonical form that can be reconstructed by the other participant.

(o) (2 points) What should a client do after establishing a TCP connection with a server, but before sending user identity and password over that connection?

Solution: It should (1) verify the identity of the server it is connecting to, and (2) establish a secure channel over the TCP connection.

4. (10 points) **Pintos**

Choose True or False for the questions below according to Pintos, the operating system you used in the projects for this class. You do not need to justify your answers.

(a) (1 point) Accesses to kernel memory (`PHYS_BASE` and above) in the system call handler and interrupt handlers operate directly on physical addresses, bypassing all page tables.

◯ True √ False

(b) (1 point) It is inherently unsafe to call `sema_down` on a zero-value semaphore in a thread while interrupts are disabled.

◯ True √ False

(c) (1 point) Interrupts are disabled while Pintos switches threads (i.e., while executing the `switch_threads` function).

√ True ◯ False

(d) (1 point) On a successful call to `pagedir_clear_page`, Pintos flushes the TLB.

√ True ◯ False

(e) (1 point) The memory for the page table of a process is allocated from the user pool.

◯ True √ False

(f) (1 point) If two processes simultaneously hold open file descriptors corresponding to the same file, then two instances of an in-memory inode exist for that file.

◯ True √ False

Answer the questions below according to Pintos, the operating system you used in the projects for this class. You do not need to justify your answers.

(g) (1 point) If Pintos is implemented with a global lock around the file system, as in Project 1, what is the maximum number of outstanding disk requests at any one time?

(g) 1

(h) (1 point) If Pintos is implemented with a 64-sector buffer cache without a global lock, as in Project 3, what is the maximum number of outstanding disk requests at any one time?

(h) 64

(i) (1 point) When scheduling timer expires, causing the `timer_interrupt` function to be invoked, in which stack does the `timer_interrupt` function execute?

Solution: It executes in the kernel stack of the currently executing thread.

(j) (1 point) While a thread is suspended (another thread is running), what is stored on its kernel stack?

Solution: The `switch_threads` frame is at the top of the stack, including the thread's registers.

5. (9 points) **Multi-Threaded Processes**

In the CS 162 projects, each Pintos process consists of a single thread of execution. Suppose now that we want to make processes in Pintos multi-threaded. We will represent a process control block using `struct process` and we will represent a thread control block using `struct thread`.

(a) (1 point) How must the `exit` system call change to handle multi-threaded processes?

> **Solution:** If a thread issues an `exit` system call, then that thread, *and all other threads in its process*, should exit.

(b) (5 points) **Fill in `struct thread` and `struct process` below.** Here are some guidelines:

- Account for each field in the current `struct thread` (see detachable reference material for this definition).
- Your design should support the full lifecycle of a thread/process: creation, system call/page fault handling, and exit.

```
struct thread {
    tid_t tid;
    enum thread_status status;
    uint8_t* stack;
    int priority;
    struct list_elem allelem;
    struct list_elem elem;
    struct list_elem processelem;
    struct process* process;
    unsigned magic;
};
```

```
struct process {
    pid_t pid;
    char name[16];
    uint32_t* pagedir;
    struct list threads;
    struct lock process_lock;
};
```

(c) (1 point) Suppose we would like to implement file descriptors in this system, as you did in Project 1. Which structure(s) must you modify to implement this?

◯ `struct thread` only √ `struct process` only ◯ both `structs`

(d) (1 point) Suppose we would like to implement non-busy-waiting sleep in this system, as you did in Project 2. Which structure(s) must you modify to implement this?

√ `struct thread` only ◯ `struct process` only ◯ both `structs`

(e) (1 point) Suppose we would like to implement a current working directory in this system, as you did in Project 3. Which structure(s) must you modify to implement this?

◯ `struct thread` only √ `struct process` only ◯ both `structs`

Commentary:

The `int priority;` and `char name[16];` field can go in either `struct` or be omitted entirely (the `name` isn't strictly needed, and the `priority` is not used by the default scheduler).

Our purpose in asking about the `exit` system call at the beginning is to help you realize that you need a way to read the `struct process` given the `struct thread`, and also a way to read the `struct thread`s for a given `struct process`. This motivates our example solution where each struct holds a reference to the other. If `struct list_elem allelem;` is placed in `struct thread`, then you need a `struct process* process;` field in `struct thread`. If `struct list_elem allelem;` is placed in `struct process`, then you need a `struct list_elem processelem;` in `struct thread` and a `struct list threads;` in `struct process`. Having both a process pointer in `struct thread` (or a PID field) and a list of threads in `struct process` is the most efficient solution, but not strictly necessary since you could iterate over all the all thread list. `unsigned magic;` must be the last field in `struct thread`. Finally, a lock is needed in `struct process` to synchronize process state when accessed by the threads in that process.

6. (12 points) **Inode Design**

Alice, Benjamin, Catherine, and David would like to design a file system for Pintos as follows. The on-disk inode has 10 direct pointers. The inode does not have any indirect pointers, but instead may point to another on-disk inode, which provides links to the remaining bytes of the file. The inode that it points to may point to yet another inode, and so on. **Note that an inode may point to another inode even if it is not completely full. This does not mean that the file is sparse, only that the remaining bytes in the inode are unused.**

(a) (1 point) What is the size, in bytes, of the largest file that can be represented by a single inode, without using a pointer to another inode?

(a) `5120 (or 10 * BLOCK_SECTOR_SIZE)`

(b) (1 point) List an access pattern that this inode structure supports efficiently.

Solution: Sequential reads and writes through a file.

(c) (1 point) List one similarity between this inode structure and the FAT file system.

Solution: Accesses to bytes at an offset in middle of a file require disk accesses linear in the byte offset.

(d) (1 point) List one difference between this inode structure and the FAT file system.

Solution: Whereas FAT stores a linked list of blocks, each node of the linked list for this inode structure references many blocks directly.

(e) (2 points) Complete `struct inode_disk` for this design. Do *not* assume any useful file data is stored in the `magic` or `unused` fields. You may not need all of the blank lines. Ensure that `sizeof(struct inode_disk) == BLOCK_SECTOR_SIZE`. Note that `off_t` is defined as follows: `typedef int32_t off_t;`.

```
struct inode_disk {
    off_t length; // length of data in THIS inode, not including linked inodes
    uint32_t isdir;

    block_sector_t direct[10];

    block_sector_t linked; // inumber of linked inode

    unsigned magic;
    uint8_t unused[ BLOCK_SECTOR_SIZE - (4 + 4 + 40 + 4 + 4) ];
};
```

(f) (6 points) The group is using the following in-memory inode structure. Note that **there is no bounce buffer within the in-memory inode** (this is different from the Pintos starter code, which has the `struct inode_disk data;` element in `struct inode`).

```
struct inode {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    bool removed;                       /* True if deleted, false otherwise. */
    int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
};
```

Based on this information, implement `byte_to_sector` for this group's inode design.

- Assume the syscall handler acquires a global file system lock, as in Project 1.
- You may call read/write from disk directly; do not use a buffer cache.
- Assume there is enough stack space to store a buffer of `BLOCK_SECTOR_SIZE` bytes.
- Assume that files are not sparse.

```
/* Returns the block device sector that contains byte offset POS
   within a file. Returns INVALID_SECTOR if the file does not contain
   data for a byte at offset POS (e.g., POS is past the end of file). */
block_sector_t byte_to_sector(const struct inode *inode, off_t pos) {
    ASSERT(inode != NULL);
    block_sector_t rv = INVALID_SECTOR;
    struct inode_disk bounce;

    block_read(fs_device, inode->sector, &bounce);

    while (pos >= bounce.length) {

        if (bounce.linked == INVALID_SECTOR) {

            return rv; // end of file

        }

        pos -= bounce.length;

        block_read(fs_device, bounce.linked, &bounce);

    }

    rv = bounce.direct[pos / BLOCK_SECTOR_SIZE];

    return rv;
}
```

7. (14 points) **Condition Variables**

Condition variables in the Pintos starter code are implemented using semaphores, and semaphores are implemented directly. In this problem, you will implement condition variables directly, **without using semaphores**.

Your condition variable should work in the same way as the existing implementation in Pintos (i.e., Mesa semantics). **Your implementation does not have to work with priority scheduling (Project 2).**

(a) (2 points) First, define `struct condition` and `cond_init` appropriately.

```
struct condition {
    struct list waiters;

};
void cond_init(struct condition* cond) {
    ASSERT(cond != NULL);

    list_init(&cond->waiters);

}
```

(b) (5 points) Implement `cond_wait`. Do **not** use semaphores.

```
void cond_wait(struct condition* cond, struct lock* lock) {
    ASSERT(cond != NULL);
    ASSERT(lock != NULL);
    ASSERT(!intr_context());
    ASSERT(lock_held_by_current_thread(lock));
    struct thread* t = thread_current();

    enum intr_level old_state = intr_disable();

    lock_release(lock);

    list_push_back(&cond->waiters, &t->elem);

    thread_block();

    lock_acquire(lock);

    intr_set_level(old_state);


}
```

(c) (5 points) Finally, implement `cond_signal`. Do **not** use semaphores.

```
void cond_signal(struct condition* cond, struct lock* lock) {
    ASSERT(cond != NULL);
    ASSERT(lock != NULL);
    ASSERT(!intr_context());
    ASSERT(lock_held_by_current_thread(lock));
    struct thread* t = thread_current();

    enum intr_level old_state = intr_disable();

    if (!list_empty(&cond->waiters)) {

        struct list_elem* entry = list_pop_front(&cond->waiters);

        struct thread* waiter = list_entry(entry, struct thread, elem);

        thread_unblock(waiter);

    }

    intr_set_level(old_state);
}
```

(d) (1 point) How would you implement `cond_broadcast`? Starting from your implementation of `cond_signal`, explain what changes you would make.

Solution: Change the `if` statement to a `while` loop.

(e) (1 point) Implementing condition variables directly makes it possible to implement a semaphore using locks and condition variables. Explain one **disadvantage** of this approach, other than performance, compared to implementing semaphores directly.

Solution: A semaphore implemented using locks and condition variables cannot be safely used in external interrupt context. In contrast, a semaphore that is implemented directly by disabling interrupts can be used safely in external interrupt context.

Commentary:

The solution given above is defensive in how it disables interrupts. If you assume that the condition variable is always used correctly according to how we learned in class (each condition variable is associated with a single lock that is held during all operations on that condition variable), then it isn't necessary to disable interrupts in `cond_signal`, and in `cond_wait`, it's fine to only disable interrupts around releasing the lock and blocking the thread. **For the purposes of grading, we considered this to be a valid solution.** That said, it is better design to disable interrupts as done in the example solution above, because *the memory safety of the condition variable should, ideally, not depend on how the user of the condition variable acquires locks.* If the user uses a different lock with `cond_signal` than they do with `cond_wait`, we don't want to concurrently modify the list of waiters in two different threads because we were depending on the locks being the same.

Surprisingly, about 12% of students attempted to implement `cond_wait` with a spin loop. I'd like to clarify here that *a busy-waiting solution is strictly inferior to one that properly puts the waiting thread to sleep. You should avoid busy-waiting solutions where possible in your code. There really is no good reason to busy wait for code that's (1) in the kernel, and (2) on a uniprocessor system, which is exactly the case of Pintos (unless you need to wait for a very small duration where scheduling costs are significant).* That said, we did not impose any penalty for busy-waiting, because the problem did not explicitly disallow it. Students who busy-waited, however, typically scored lower than students who did not, because a proper busy-waiting implementation of condition variables is trickier to implement.

A few students attempted to block a thread by acquiring a lock twice from the same thread. I'd like to clarify that *you should never attempt to block a thread by acquiring a lock, and then attempt to unblock it by releasing a lock from another thread. In Pintos, if a thread holds a lock and then attempts to acquire it, the result is a kernel panic.* It is indeed the case that some lock implementations, especially implementations of userspace locks, can be used this way. Even so, using locks this way is extremely bad practice. Even if we were to ignore this and implement a condition variable this way, an additional obstacle to this approach is that locks are stateful, whereas condition variables are not. Thus, a correct implementation needs one lock per waiting thread, not one lock for the condition variable. The condition variable would have to have a list of locks, and each thread would have to be put to sleep by double-acquiring a different lock from other threads. See the Pintos implementation, which uses semaphores, as an example of how this is done. No student who attempted this approach implemented it correctly.

For the final part of this question (disadvantage of implementing a semaphore out of locks and condition variables), many students wrote that it is more complex to do that than to implement semaphores directly. This is highly debatable. It is common to build synchronization primitives out of others. If you need a complex type of lock, it is more natural to build it out of locks and condition variables than it is to implement it by disabling interrupts. Think about how we implemented reader-writer locks and bounded buffers in lecture. We awarded some partial credit to answers that gave complexity as a reason, along with a concrete reason or example. One concrete reason we gave full credit to is that locks would have to be reimplemented to no longer use a semaphore with this approach. A few other answers with debatable reasons but concrete reasons or examples were awarded half credit.

8. (18 points) **Paged Virtual Memory**

Consider an x86 computer with the following memory architecture:

- 32-bit virtual address space
- 32-bit physical address space
- 4 KiB page size
- Two-level page table
- 4-entry, fully associative TLB, unified for both instructions and data
- 32-bit page table entry (PTE) size

The first instruction to be executed is at the address `0x1004 0100`.

(a) (4 points) Virtual Address Structure

i. (2 points) Describe which bits of the 32-bit virtual address are used for each part of the virtual address translation.

bits [31 : 22] are the index into the root page table page

bits [21 : 12] are the index into the second level page table page

bits [11 : 0] are the byte offset in the page

ii. (2 points) For the virtual address of the first instruction, `0x1004 0100`, fill in the value of each line where appropriate or N/A where it is not. For each page accessed when fetching that instruction, give the byte offset of the reference into that page.

Index of entry in Root Page Table: 0x040 (or 0b0001 0000 00)

Byte offset of entry in Root Page Table: 0x100 (or 0b0001 0000 0000)

Index of entry in L2 Page Table: 0x040 (or 0b00 0100 0000)

Byte offset of PTE within L2 PT page: 0x100 (or 0b0001 0000 0000)

Code page offset: 0x100

Question continues on the next page →

The state of the machine described on the previous page is shown below. The contents of several frames of physical memory are shown on the right with physical addresses. On the left are several machine registers, including the `eip`, `esp`, and the page table base register (`cr3`), which contains the physical frame number of the root page table. The TLB is initially empty. Page table entries have the valid flag as the most significant bit and the physical frame number of valid entries in the low order bits. Other flags can be assumed to be zero for this problem.

Registers

cr3 (PTBR)	0x0001 0030
eip	
esp	0x1044 0100
eax	0x8001 0050
ebx	0x0001 0080
ecx	
edx	

Current Instruction

```
0x1004 0100 pushl %eax
```

TLB Contents

Valid	Tag	Frame

Physical Memory

Phys. Addr.	Contents
0x1001 0000	
...	
0x1001 0100	0x0001 0080
0x1001 0104	0x8001 5350
0x1001 0108	0x8001 0080
0x1002 0000	
...	
0x1002 0100	0x8001 0050
0x1002 0104	0x8001 0080
0x1002 0108	0x0001 0080
0x1003 0000	
...	
0x1003 0100	0x8001 0020
0x1003 0104	0x8001 0040
0x1003 0108	0x0001 0080
0x1004 0000	
...	
0x1004 0100	0x0001 0000
0x1004 0104	0x0001 0010
0x1004 0108	0x0001 0080
0x1005 0000	
...	
0x1005 0100	0x0001 5350
0x1005 0104	0x8001 5350
0x1005 0108	0x0001 0080

(b) (6 points) **You are to step through one instruction whose address and disassembly are shown above.** In the space provided below, write down the operation, address, and value associated with every memory operation associated with the instructions up to the point of the first page fault, by filling in the blank cells in the table. You should also update the state of the memory, registers, and TLB by over-writing the figure. **You may not need all of the rows provided in the table.**

Operation	Address	Value	Comment
N/A	N/A	N/A	Fetch instruction
Fetch root PT	`0x1003 0100`	`0x8001 0020`	Read valid PTE for top level page
Fetch L2 PT	`0x1002 0100`	`0x8001 0050`	Read valid PTE for code page
Add to TLB	Tag: `0x10040`	Val: `0x10050`	N/A
Read Instr.	`0x1005 0100`	`0x0001 5350`	Fetch `pushl %eax`
N/A	N/A	N/A	Write `eax` to `*esp`
Fetch root PT	`0x1003 0104`	`0x8001 0040`	Read valid PTE in top level page
Fetch L2 PT	`0x1004 0100`	`0x0001 0000`	Read invalid PTE for data page
Page Fault	`0x1044 0100`	N/A	N/A

Question continues on the next page ⟶

Commentary:

Unfortunately, the exam as originally given had a bug that prevented a page fault from occurring as intended. For the benefit of future semesters, I fixed the bug in the version posted to the course website. The bug in the original exam was that the longword in physical memory at address `0x1004 0100` was `0x8001 5350` instead of `0x0001 0000`, which corresponds to a valid PTE because the most significant bit is set. Thus, the `pushl %eax` instruction would actually have written to the physical page `0x15350`, instead of page faulting as intended. Fortunately, this affects only the very last step of the problem; our approach in grading was to give the last "page fault" point for free to anyone who fetched the correct entry from the L2 page table.

A minor inaccuracy in the problem is that the `pushl` instruction actually decrements `esp` and writes to the new value of `esp`. You saw this behavior on Homework 5. As this is not the point of this question, we decided to simplify it by (1) giving the entry in the table that says, "Write `eax` to `*esp`," and (2) choose a value of `esp` that is not close enough to a page boundary for decrementing by 4 to matter. Additionally, we omitted `eip` from the registers table to avoid confusion around whether `eip` points to the current instruction or the next one being executed. Unlike the page fault issue, these inaccuracies were deliberate.

(c) (2 points) Upon entering the page fault handler, the operating system determines that the fault occurs on reference to an unallocated page. Explain why the operating system should not read in a page from disk in this case and why it is reasonable for it to instead allocate a new page to this process.

Solution: There is no object on backing store for this page. It represents an unused segment of the process virtual address space. By pushing this value onto the stack, the process is implicitly extending its stack segment. It will need to allocate a page frame, update the PTE to refer to this page and restart the instruction.

(d) (2 points) Assume the first free frame is number `0x10010`. What changes to the memory does the operating system need to make to provide this page to the process to resolve this fault?

Solution: The PTE at physical 0x1004 0100 is updated to contain 0x8001 0010. Valid and frame 0x10010.

(e) (1 point) What does the operating system need to do with the page frame contents before returning from the page fault?

Solution: Clear its contents (i.e., `memset` to `0x00` or allocate with `PAL_ZERO`).

(f) (1 point) Does the operating system need to do anything to the TLB before returning from the page fault?

Solution: No. The previously invalid PTE would not have been present in the TLB. No need to flush.

(g) (2 points) Upon resuming from the page fault, describe what will happen in the user process (i.e., memory accesses, TLB accesses, etc.).

Solution: The `pushl %eax` instruction will be re-executed. This time it will hit in the TLB for the instruction fetch and load the instruction. It will miss in the TLB for the write, but the MMU will find a valid page table entry for the data page and load that translation into the TLB. It will store the value of `eax` to the data page and update `esp`.

9. (0 points) **Optional Questions**

   (a) (0 points) Having finished the exam, how do you feel about it? Check **all** that apply:

   ☐  ☐  ☐  ☐  ☐  ☐  ☐  ☐ 

   ☐  ☐  ☐ Other (please draw):

   (b) (0 points) If there's anything you'd like to tell the course staff (e.g., feedback about the class or exam, suspicious activity during the exam, new logo suggestions, etc.) you can write it on this page.

This page is intentionally left blank.

Do not write any answers on this page.

You may use it for *ungraded* scratch space.

```
/****************************** Threads ******************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
/****************************** Processes ******************************/
pid_t fork(void);          pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);
void exit(int status);
/************************** High-Level I/O **************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);
/************************** Low-Level I/O **************************/
int open(const char *pathname, int flags); (O_APPEND|O_CREAT|O_TMPFILE|O_TRUNC)
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);
/************************** Pintos Lists **************************/
void list_init(struct list *list);
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
struct list_elem *list_remove(struct list_elem *elem);
struct list_elem *list_pop_front(struct list *list);
struct list_elem *list_pop_back(struct list *list);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
```

```
/************************** Pintos Threads ****************************/
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);
void cond_init(struct condition *cond);
void cond_wait(struct condition *cond, struct lock *lock);
void cond_signal(struct condition *cond, struct lock *lock);
void cond_broadcast(struct condition *cond, struct lock *lock);
enum intr_level intr_get_level(void);
enum intr_level intr_set_level(enum intr_level);
enum intr_level intr_enable(void);
enum intr_level intr_disable(void);
bool intr_context(void);
void intr_yield_on_return(void);
tid_t thread_create(const char *name, int priority, void (*fn)(void *), void *aux);
void thread_block(void);
void thread_unblock(struct thread *t);
struct thread *thread_current(void);
void thread_exit(void) NO_RETURN;
void thread_yield(void);
struct thread {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
/************************** Pintos Memory *****************************/
void *palloc_get_page(enum palloc_flags); (PAL_ASSERT|PAL_ZERO|PAL_USER)
void *palloc_get_multiple(enum palloc_flags, size_t page_cnt);
void palloc_free_page(void *page);
```

```
void palloc_free_multiple(void *pages, size_t page_cnt);
#define PGBITS 12 /* Number of offset bits. */
#define PGSIZE (1 << PGBITS) /* Bytes in a page. */
unsigned pg_ofs(const void *va);          uintptr_t pg_no(const void *va);
void *pg_round_up(const void *va);        void *pg_round_down(const void *va);
#define PHYS_BASE 0xc0000000
uint32_t *pagedir_create (void);          void pagedir_destroy(uint32_t *pd);
bool pagedir_set_page(uint32_t *pd, void *upage, void *kpage, bool rw);
void *pagedir_get_page(uint32_t *pd, const void *upage);
void pagedir_clear_page(uint32_t *pd, void *upage);
bool pagedir_is_dirty(uint32_t *pd, const void *upage);
void pagedir_set_dirty(uint32_t *pd, const void *upage, bool dirty);
bool pagedir_is_accessed(uint32_t *pd, const void *upage);
void pagedir_set_accessed(uint32_t *pd, const void *upage, bool accessed);
void pagedir_activate(uint32_t *pd);
/************************ Pintos File Systems ***************************/
bool filesys_create(const char *name, off_t initial_size);
struct file *filesys_open(const char *name);
bool filesys_remove(const char *name);
struct file *file_open(struct inode *inode);
struct file *file_reopen(struct file *file);
void file_close(struct file *file);
struct inode *file_get_inode(struct file *file);
off_t file_read(struct file *file, void *buffer, off_t size);
off_t file_write(struct file *file, const void *buffer, off_t size);
bool inode_create(block_sector_t sector, off_t length);
struct inode *inode_open(block_sector_t sector);
block_sector_t inode_get_inumber(const struct inode *inode);
void inode_close(struct inode *inode);
void inode_remove(struct inode *inode);
off_t inode_read_at(struct inode *inode, void *buffer, off_t size, off_t offset);
off_t inode_write_at(struct inode *inode, const void *buffer, off_t size, off_t offset);
off_t inode_length(const struct inode *inode);
bool free_map_allocate(size_t cnt, block_sector_t *sectorp);
void free_map_release(block_sector_t sector, size_t cnt);
size_t bytes_to_sectors(off_t size);
struct inode_disk {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
};
/************************** Pintos Devices ******************************/
typedef uint32_t block_sector_t;
#define BLOCK_SECTOR_SIZE 512
void block_read(struct block *block, block_sector_t sector, void *buffer);
void block_write(struct block *block, block_sector_t sector, const void *buffer);
```