Computer Science 162
Sam Kumar
University of California, Berkeley
Final Exam
August 14, 2020

Name	
Student ID	

This is an open-book exam. You may access existing materials, including online materials (such as the course website). During the exam, you may **not** communicate with other people regarding the exam questions or answers in any capacity. For example, posting to a forum asking for help is prohibited, as is sharing your exam questions or answers with other people (or soliciting this information from them).

You have 170 minutes to complete it. If there is something in a question that you believe is open to interpretation, please make a private post on Piazza asking for clarification. If you run into technical issues during the exam, join our open Zoom call and we will try to help you out. The final page is for reference.

We will overlook minor syntax errors in grading coding questions. You do not have to add the necessary `#include` statements at the top.

Grade Table (for instructor use only)

Question:	1	2	3	4	5	6	7	8	9	10	11	12	Total
Points:	1	10	26	8	8	16	18	14	13	20	26	0	160
Score:													

1. (1 point) Please check the boxes next to the following statements after you have read through and agreed with them.

   □ I will complete this quiz with integrity, doing the work entirely on my own.

   □ I will NOT attempt to obtain answers or partial answers from any other people.

   □ I will NOT post questions about this quiz on online platforms such as StackOverflow.

   □ I will NOT discuss any information about this quiz until 24 hours after it is over.

   □ I understand the consequences of violating UC Berkeley's Code of Student Conduct.

2. (10 points) **Operating System Concepts**

Choose **either** True or False for the questions below. You do not need to provide justifications. **Each student received 10 questions, chosen randomly from the ones below.**

(a) (1 point) If a file system does not use a buffer cache (i.e., all writes go directly to the storage device), then it does not need recovery mechanisms (e.g., journaling).

○ True

○ False

(b) (1 point) Reed-Solomon Codes allow a message of size n to be split into m fragments of size approximately $\frac{n}{k}$, such that *any* k of the fragments are enough to recover the original message (for $m > k$).

○ True

○ False

(c) (1 point) The Internet Protocol (IP) delivers packets from a *process* on one host to a *process* on another host.

○ True

○ False

(d) (1 point) The Internet Protocol (IP) guarantees that packets sent from one host to another will not be reordered in the network.

○ True

○ False

(e) (1 point) The Internet Protocol (IP) guarantees that packets sent from one host to another will not be duplicated.

○ True

○ False

(f) (1 point) The Transmission Control Protocol (TCP) uses timeouts to detect if a packet is lost.

○ True

○ False

(g) (1 point) In a Transmission Control Protocol (TCP) connection, a host can remove data from the send buffer as soon as it sends a packet containing that data to the receiving host.

○ True

○ False

(h) (1 point) In a Transmission Control Protocol (TCP) connection, a host can remove data from the receive buffer as soon as it is transferred into a process' address space via a `read` system call.

○ True

○ False

(i) (1 point) In a Transmission Control Protocol (TCP) connection, the amount of "in-flight" data from one host to another is influenced by the advertised window size of the receiving host.

○ True
○ False

(j) (1 point) The purpose of congestion control is to prevent one host from sending data so fast as to overwhelm the receiving host.

○ True
○ False

(k) (1 point) Two-Phase Commit (2PC) allows multiple nodes in a distributed system to perform an action simultaneously.

○ True
○ False

(l) (1 point) With quorum consensus, the master (directory) node of a directory-based key-value store with replication can acknowledge a client's `put` request without first waiting for acknowledgments from all replicas.

○ True
○ False

(m) (1 point) In a linearizable key-value store, a `get` that is processed concurrently with a `put` to the same key may see the either the old value (i.e., the value before the `put` is processed) or the new value (i.e., the value after the `put` is processed).

○ True
○ False

(n) (1 point) The Chord key-value store relies on a master node that is aware of all of the storage nodes in the system.

○ True
○ False

(o) (1 point) In a hardware virtualization setup, while a process belonging to the Guest OS is executing on the processor, the page table used for address translation is the one maintained by the Guest OS. (Assume that the processor does *not* supported nested paging.)

○ True
○ False

3. (26 points) **Short Answer**

   Each student received 11 questions, taken from the ones below as follows: one of a or b, c, one of d or e, one of f or g, one of h or i, j, k, one of l or m, n, o, and p.

   (a) (2 points) What system call must an RPC server successfully issue, for the operating system to complete the TCP three-way handshake for incoming RPC requests?

   (b) (2 points) What does the client stub do in a Remote Procedure Call?

   (c) (2 points) Suppose that transferring data from a hard-disk drive has an average startup cost of 10 ms (including controller time, seek time, and rotation time) and can transfer data read sequentially on disk at 50 MB/s. At what transfer size is the half-power bandwidth achieved? Show your work.

   (d) (2 points) The concept of *caching* arises in a variety of systems. Give two examples of caching *in software systems* that we studied in class.

   (e) (2 points) The concept of *transparency* arises in a variety of systems. Give two examples of transparency *in software systems* that we studied in class.

(f) (3 points) In a journaling file system based on Redo Logging (the logging scheme that we discussed in class), why is it important to wait until the corresponding COMMIT is written to the log *before* performing an operation?

(g) (3 points) What property of the individual log entries allows the journaling file system discussed in class to never have to undo any operations during recovery?

(h) (3 points) Consider a journaling file system based on Redo Logging (the logging scheme that we discussed in class). Suppose that the system restarts after a crash. Upon inspecting its log, it finds a transaction in the log, including the final COMMIT entry of the transaction. What should the system do to process this transaction?

(i) (3 points) Consider a journaling file system based on Redo Logging (the logging scheme that we discussed in class). Suppose that the system restarts after a crash. Upon inspecting its log, it finds a transaction in the log, but the final COMMIT entry of the transaction is not present in the log. What should the system do to process this transaction?

(j) (2 points) In Two-Phase Commit, why must the coordinator write its final decision (COMMIT or ABORT) to its local log *before* sending it to the worker nodes?

(k) (2 points) A student in CS 162 writes the following x86 assembly code that runs in Pintos in a kernel thread in **kernel mode**:

```
movl %eax, (%ebx)
movl (%ebx), %ecx
```

To her surprise, the registers `%eax` and `%ecx` contain *different* values after this code executes. Assume that no thread writes to the stack of any other thread, and that the memory address stored in `%ebx` does not belong to any thread's stack. Which of the following are possible reasons why this may have happened?

□ In between the two instructions, the kernel context-switched to a different thread, which modified the value stored in the register `%ebx`.

□ In between the two instructions, the kernel context-switched to a different thread, which read the value in memory at the address stored in `%ebx`.

□ In between the two instructions, the kernel context-switched to a different thread, which wrote to memory at the address stored in `%ebx`.

□ `%ebx` contains a memory address that maps to an I/O device instead of physical memory.

(l) (2 points) Suppose that a Guest OS is running on a Host OS using a traditional hardware virtualization setup. Explain how control is transferred from the Guest OS' system call handler to the guest process (i.e., the process running on the Guest OS) once the Guest OS finishes handling a system call.

(m) (2 points) Suppose that a Guest OS is running on a Host OS using a traditional hardware virtualization setup. Explain how control is transferred from the guest process (i.e., the process running on the Guest OS) to the Guest OS system call handler when the guest process issues a system call.

Choose the best answer for each of the questions below, and **explain your answer in the box provided**.

(n) (2 points) Consider two periodic tasks, S and T. Each task consists of a series of CPU bursts; each task yields to the scheduler between CPU bursts but (for simplicity) has zero I/O time between CPU bursts. **This is exactly the setup from Problem 3 on the Scheduling Lab.** Is this an open system or a closed system? Explain.

□ Open system □ Closed system

(o) (3 points) If a user `write`s to a file in the Network File System (NFS) and then `close`s it, then the user's modifications are *guaranteed* to be visible to any user who `read`s the file *immediately* after the original user `close`d it. True or false? Explain.

□ True □ False

(p) (3 points) If a user `write`s to a file in the Andrew File System (AFS) and then `close`s it, then the user's modifications are *guaranteed* to be visible to any user who `read`s the file *immediately* after the original user `close`d it. True or false? Explain.

□ True □ False

4. (8 points) **Operating System Abstractions**

Each student received 4 questions out of the ones below, selected randomly. For each question below, select **all** of the choices that apply. You should assume:

- Calls to `open`, `fopen`, `fork`, `pthread_create`, `malloc`, and `realloc` always succeed.
- Calls to `read`, `write`, `dup`, and `dup2` succeed **if a valid file descriptor is provided**.
- The necessary header files from the C standard library are `#included`.
- Before each program is run, `file.txt` is an empty file.
- All threads *eventually* make progress. Make no other assumptions about the scheduler.

(a) (2 points) Which of the following could be the contents of `file.txt` after all processes of the program below terminate?

```
int main(int argc, char** argv) {
    int fd = open("file.txt", O_WRONLY);
    if (fork() == 0) {
        write(fd, "a", 1);
    } else {
        write(fd, "b", 1);
    }
}
```

□ (empty) □ a □ b □ aa □ ab □ ba □ bb □ aab □ aba
□ baa □ abb □ bab □ bba □ aabb □ abab □ baab □ baba
□ bbaa

(b) (2 points) Which of the following could be the contents of `file.txt` after all processes of the program below terminate?

```
int main(int argc, char** argv) {
    int fd = open("file.txt", O_WRONLY);
    if (fork() == 0) {
        write(fd, "a", 1);
    }
    write(fd, "b", 1);
}
```

□ (empty) □ a □ b □ aa □ ab □ ba □ bb □ aab □ aba
□ baa □ abb □ bab □ bba □ aabb □ abab □ baab □ baba
□ bbaa

(c) (2 points) Which of the following could be the contents of `file.txt` after all processes of the program below terminate?

```
int main(int argc, char** argv) {
    int fd = open("file.txt", O_WRONLY);
    if (fork() == 0) {
        write(fd, "a", 1);
        close(fd);
    } else {
        write(fd, "b", 1);
        close(fd);
    }
}
```

□ (empty) □ a □ b □ aa □ ab □ ba □ bb □ aab □ aba
□ baa □ abb □ bab □ bba □ aabb □ abab □ baab □ baba
□ bbaa

(d) (2 points) Which of the following could be the contents of `file.txt` after all processes of the program below terminate?

```
int main(int argc, char** argv) {
    int fd = open("file.txt", O_WRONLY);
    if (fork() == 0) {
        dup2(fd, fd + 1);
        write(fd + 1, "a", 1);
    }
    write(fd + 1, "b", 1);
}
```

□ (empty) □ a □ b □ aa □ ab □ ba □ bb □ aab □ aba
□ baa □ abb □ bab □ bba □ aabb □ abab □ baab □ baba
□ bbaa

(e) (2 points) Which of the following could be the contents of `file.txt` after the program below terminates?

```
int fd;
void* helper(void* arg) {
    write(fd, "a", 1);
}
int main(int argc, char** argv) {
    fd = open("file.txt", O_WRONLY);
    pthread_t thread;
    pthread_create(&thread, NULL, helper, NULL);
    pthread_join(thread, NULL);
    write(fd, "b", 1);
}
```

□ (empty) □ a □ b □ aa □ ab □ ba □ bb

(f) (2 points) Which of the following could be the contents of `file.txt` after the program below terminates?

```
int fd;
void* helper(void* arg) {
    write(fd, "b", 1);
}
int main(int argc, char** argv) {
    fd = open("file.txt", O_WRONLY);
    write(fd, "a", 1);
    pthread_t thread;
    pthread_create(&thread, NULL, helper, NULL);
}
```

□ (empty) □ a □ b □ aa □ ab □ ba □ bb

(g) (2 points) Which of the following could be the contents of `file.txt` after the program below terminates?

```
int fd;
void* helper(void* arg) {
    write(fd, "a", 1);
}
int main(int argc, char** argv) {
    fd = open("file.txt", O_WRONLY);
    pthread_t thread;
    pthread_create(&thread, NULL, helper, NULL);
    write(fd, "b", 1);
    pthread_join(thread, NULL);
}
```

□ (empty) □ `a` □ `b` □ `aa` □ `ab` □ `ba` □ `bb`

5. (8 points) **Pintos**

Choose **either** True or False for the questions below. You do not need to provide justifications. **Each student received 5 questions, chosen randomly from the ones below.**

(a) (1 point) In Pintos, a *user program* may obtain a pointer to the TCB by calling `thread_current` and then access its fields directly.

○ True
○ False

(b) (1 point) In Pintos, interrupt handlers execute in their own stack, separate from the stack of any kernel thread.

○ True
○ False

(c) (1 point) It is inherently unsafe to call `sema_up` on a semaphore in external interrupt context.

○ True
○ False

(d) (1 point) The idle thread puts the processor to sleep with interrupts disabled.

○ True
○ False

(e) (1 point) Once a file is created, its length can never decrease unless it is removed and created again. (Assume that Project 3 is implemented.)

○ True
○ False

(f) (1 point) The kernel image (containing kernel code, globals, etc.) exists as a file in the Pintos file system.

○ True
○ False

(g) (1 point) Pintos is a microkernel.

○ True
○ False

(h) (1 point) In the default Pintos file system (before Project 3), free disk blocks are allocated using a free list (i.e., an on-disk linked list).

○ True
○ False

Fill in the blanks correctly for the questions below. You do not need to provide justifications. **Each student received one of Part i and j. Every student received Part k.**

(i) (1 point) What is the purpose of the "magic" field in `struct thread` in Pintos?

(j) (1 point) Why is it important for `sizeof(struct thread)` to not grow too large in Pintos?

(k) (2 points) Suppose the scheduling timer expires while interrupts are disabled. When will the `timer_interrupt` function (i.e., the external interrupt handler) be invoked next?

6. (16 points) **Priority Donation**

Priority donation in Pintos can be formalized as a directed graph. Each thread in the system corresponds to a vertex in the graph. An edge from t_1 to t_2, denoted (t_1, t_2), exists in the graph if t_1 is blocked on acquiring a mutex that t_2 holds. Together with each thread t's base priority, t.priority, the graph provides sufficient information to compute each thread's *effective priority*.

(a) (1 point) What is the maximum number of outgoing edges from any particular vertex in the graph? Explain your answer.

(b) (1 point) It is safe to assume that the above graph has no cycles, because a cycle would indicate a particular type of scheduling bug in the kernel. What type of bug would a cycle indicate? (Hint: We spent a full lecture in class studying this type of bug.)

(c) (3 points) For any thread t (a vertex in the graph), let $ep(t)$ denote the *effective priority* of t. Explain how to calculate $ep(t)$. **It is acceptable to write your answer as a formula or explain it in words; we do not expect you to write out pseudocode.** *Hint: Because the graph has no cycles, feel free to use a recursive definition, where $ep(t)$ is defined using the effective priorities of t's ancestors.*

(d) (2 points) Suppose that a particular thread changes its priority by calling `thread_set_priority`. Could other threads' effective priorities immediately change as a result of this call? Explain.

Suppose that we implement a new synchronization primitive in Pintos, called an n-mutex. An n-mutex can be held simultaneously by up to n threads. **You should assume that n is a constant known at compile-time, and that $n > 1$.** A thread blocks on acquiring an n-mutex if and only if the n-mutex is already held by n threads. As before, an edge from t_1 to t_2 exists in the graph if t_1 is blocked on acquiring a mutex and t_2 is a holder of that mutex.

(e) (1 point) With n-mutexes in the system, what is the maximum number of outgoing edges from any particular vertex in the graph? Explain your answer. (This is a repeat of Part a, with n-mutexes considered this time.)

(f) (4 points) Unlike the case of regular mutexes, a cycle in the graph does *not* necessarily imply a scheduling bug. Give an example of a lock acquisition pattern involving n-mutexes that would result in a cycle but does *not* indicate the scheduling bug in Part b.

(g) (4 points) Explain how to compute the effective priority of each thread when the graph contains cycles. **Explaining your algorithm in words is fine; we do not expect you to write out pseudocode.** *Hint: If it is useful, you may use your algorithm/formula from Part c as a subroutine.*

7. (18 points) **Multicolor Lock**

Consider a new synchronization primitive called a *multicolor lock*. A thread can acquire a multicolor lock as either blue or gold. When there are multiple threads waiting to acquire the lock, the lock gives preference to blue acquisitions over gold acquisitions. Note that blue waiters do not preempt the lock if it is currently held by a gold holder; they must wait for the current lock holder to release the lock, before they can acquire it. The colors blue and gold are represented as the boolean values `true` and `false`, respectively.

In this question, you will implement a multicolor lock that supports this behavior. Your implementation should run in a Linux user program, using the `pthread` library for synchronization. You may make a reasonable assumption on the number of waiters—for example, that it does not exceed `UINT32_MAX`. **You should make sure to use synchronization primitives properly. For example, a thread should only release a lock if it currently holds that lock.** You may not need to use all of the blank lines. It is also fine if your solution has more lines than the number of blank lines provided.

(a) (3 points) First, implement the `mc_lock_t` struct.

```
typedef struct {

    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________

} mc_lock_t;
```

(b) (3 points) Next, implement the `mc_lock_init` function, which initializes a new `mc_lock_t` structure.

```
void mc_lock_init(mc_lock_t* mc_lock) {

    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________

}
```

(c) (6 points) Now, implement the `mc_lock_acquire` function.

```
void mc_lock_acquire(mc_lock_t* mc_lock, bool color) {

}
```

(d) (6 points) Finally, implement the `mc_lock_release` function.

```
void mc_lock_release(mc_lock_t* mc_lock) {

}
```

8. (14 points) **Scheduling and Performance**

Consider a sequence B_i of CPU bursts, where (for simplicity) each CPU burst has a fixed length M. The first burst, B_0, arrives at time $t = 0$ (i.e., $\text{ArrivalTime}(B_0) = 0$). For $i \geq 1$, the arrival time of B_i is given by $\text{ArrivalTime}(B_i) = \text{ArrivalTime}(B_{i-1}) + X_i$, where the X_i are i.i.d. exponentially distributed random variables with parameter λ. To allow the CPU bursts to execute concurrently, we model each CPU burst as executing in its own task.

This is exactly the same setup as Problem 2 in the Scheduling Lab from Project 2. As in the Scheduling Lab, assume that the system has one CPU. In the scheduling lab, you ran a simulation in which you measured latency as a function of the arrival rate.

(a) (1 point) If the arrival rate λ is increased beyond a certain threshold, then the queuing time grows indefinitely over time, never converging to a fixed value. Denote this threshold as λ^*. Write λ^* in terms of the variables given in the problem.

(b) (1 point) What squared coefficient of variance C describes the service time distribution (CPU burst distribution)?

(c) (4 points) Write a closed-form expression for the expectation of response time in steady state, which we will denote as R. Your expression for R should be in terms of the arrival rate λ and burst length M. Assume that the scheduler is FCFS. Steady state means that you can assume that there are very large number of CPU bursts, ignoring the time to fill the pipeline at the beginning and flush it at the end. Show your work (you do not have to show intermediate equations, but explain how you arrived at your final answer).

(d) (3 points) Suppose that you run a simulation in Python, as you did in the Scheduling Lab. Because your simulation uses a finite number of CPU bursts, your simulation includes the effects of filling the pipeline at the beginning and flushing it at the end. Given that your simulation includes these “fill-and-flush” effects, will the response time you measure in your simulation be (1) less than, (2) greater than, or (3) approximately equal to what you calculated in Part c? Explain.

□ less than □ greater than □ approximately equal to

(e) (3 points) Suppose that, instead of using a FCFS scheduler, we instead use a SRTF scheduler whose time quantum is much less than M. Would using an SRTF scheduler instead of FCFS change your answer to Part c (fixed-length CPU bursts)? State whether R will increase, decrease, or stay the same, and explain your answer.

□ increase □ decrease □ stay the same

(f) (2 points) Explain why SRTF is difficult to implement in a real operating system, and list an easier-to-implement approximation to SRTF that we studied in class.

9. (13 points) **Inode Design**

Consider a file system for Pintos based on Berkeley FFS, with 10 direct pointers, one indirect pointer, one doubly indirect pointer, and one triply indirect pointer, in that order, in each inode. Rather than having each direct pointer point to a single block, however, each direct block pointer is replaced by a reference to an *extent*—a set of sequential blocks on disk. This applies not only to direct pointer directly in the inode, but also to direct pointers stored in indirect blocks. For example, an indirect block pointer still points to a single block, but the direct pointers within that block now refer to extents.

(a) (2 points) List one advantage of this inode structure compared to Berkeley FFS.

(b) (2 points) List one disadvantage of this inode structure compared to Berkeley FFS.

(c) (2 points) List one similarity between this inode structure and Windows NTFS.

(d) (7 points) Complete `struct inode_disk` for this design. Do *not* assume any useful file data is stored in the `magic` or `unused` fields.

Ensure that `sizeof(struct inode_disk) == BLOCK_SECTOR_SIZE`. Note that `off_t` is defined as follows: `typedef int32_t off_t;`.

If you need to define any new `struct`s or `union`s, do so on the lines above the `struct inode_disk` definition (there is at least one good solution that does not require any additional `struct`s or `union`s, but there are also good solutions that do). You may not need to use all of the blank lines. It is also fine if your solution has more lines than the number of blank lines provided.

```
____________________________________________________________

____________________________________________________________

____________________________________________________________

____________________________________________________________

____________________________________________________________

____________________________________________________________

struct inode_disk {
    off_t length;
    uint32_t isdir;

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    unsigned magic;
    uint8_t unused[______________________________________________];
};
```

10. (20 points) **Distributed Key-Value Stores**

Recall that a distributed key-value store supports two operations:

- `GET`: allows a client to obtain the value corresponding to a key
- `PUT`: allows a client to insert a new key-value pair. If a key-value pair already exists with the same key, it is overwritten by the new one.

(a) Bob, a student in CS 162, sets up a distributed key-value store consisting of a master node and k worker nodes, indexed 0 to $k-1$. When a key-value pair is inserted in the system, the client sends the key-value pair to the master node. The master node hashes the key to obtain an integer h, and then the master node forwards the key-value pair to the worker node whose index is $h \mod k$. For example, if the hash of the key is $h = 131$ and the are $k = 3$ workers, then that key-value pair will be stored on the worker node whose index is 2.

i. (1 point) If one of the k worker nodes fails permanently, what fraction of key-value pairs (on average) will be lost? (You may assume that the hash function is "ideal"—it behaves as if the hash of each key is chosen uniformly at random.)

ii. (1 point) If one additional worker node is added to the system, what fraction of key-value pairs must be moved to a different server in this setup? (You may assume that the hash function is "ideal"—it behaves as if the hash of each key is chosen uniformly at random.)

iii. (2 points) How might we reduce the fraction of key-value pairs that must be moved when a new server is added to the system?

iv. (3 points) An alternative design is to not have a master node at all. Instead, the client hashes the key locally, and sends the key-value pair directly to the appropriate worker. List an advantage and a disadvantage of this idea compared to the approach described above.

v. (2 points) Can we use Two-Phase Commit (2PC) on `GET` and/or `PUT` operations to improve the consistency guarantees of this distributed key-value store? Either explain how to use 2PC to improve this system, or explain why 2PC is not applicable.

(b) Alice, a different student in CS 162, sets up a *separate* distributed key-value store from the previous part. She chooses to replicate the server—instead of having a single server, she chooses three servers in geographically separate locations to host the key-value pairs. In her design, each key-value pair is replicated on all three servers, so that each client can read data from any of the servers. To ensure that all clients see a consistent view of the files, a write only succeeds if *all* three servers acknowledge the operation.

i. (2 points) How might replicating the server, as stated above, affect the availability of read operations?

ii. (2 points) How might replicating the server, as stated above, affect the availability of write operations?

iii. (3 points) Is it possible to replicate the server in a way that does not require writes to wait for acknowledgments from all three servers, without compromising consistency? Either explain how to replicate the server in such a way, or explain why it is not possible.

(c) (2 points) List one way in which Alice's approach (replicating the server) is preferable to Bob's approach (hash keys to different servers).

(d) (2 points) List one way in which Bob's approach (hash keys to different servers) is preferable to Alice's approach (replicating the server).

11. (26 points) **Shared Page**

On Quiz 2, you helped Bobby, William, Jonathan, and Kevin implement an IPC mechanism based on single, shared pipe. Unfortunately, they find this mechanism difficult to use. Instead, they would like to implement IPC based on **shared memory**.

Bobby suggests implementing the following system call:

```
bool write_remote(pid_t pid, uint8_t* address, uint8_t value);
```

It writes the `value` to the specified `address` in the address space of the process with the given `pid`. It returns `true` on success and `false` on failure.

(a) (2 points) Explain why Bobby's suggested IPC mechanism is *not* well-designed.

To address the shortcomings in Bobby's suggestion, William suggests that a process should be able to allocate a *shared page*. When the process spawns a child process, the child process inherits the shared page, allowing the parent and child to communicate using memory on the shared page.

(b) (1 point) On Linux, what system call(s) can be used to achieve William's suggested IPC functionality?

(b) ____________

To implement William's suggestion in Pintos, Jonathan suggests the following system call interface:

```
/*
 * Allocates a page at the specified virtual address. If this process spawns
 * new ones with exec(), they inherit this page, mapped to the same virtual
 * address, unless a page is already mapped at this virtual address in the
 * child process. Returns the virtual address of the new page on success, and
 * NULL on failure.
 * If a process already has a shared page, either because it made a successful
 * call to shared_page or because it inherited one from its parent, subsequent
 * calls to shared_page should fail.
 */
void* shared_page(void* address);
```

(c) (1 point) If a process maps a shared page at an `address` of its choice, and then issues an `exec` system call to spawn a child process, how might it communicate to the child process at which address to find the shared page?

(d) (2 points) In Pintos, the `pagedir_set_page` function might fail, if there is not enough memory to set the entry in the page table. Why might setting/creating an entry in the page table require allocating memory? Be specific about how this relates to the structure of the page table.

(e) (20 points) In the remainder of this question, you will implement the `shared_page` system call in Pintos. For simplicity, you may make the following assumptions:

- Each process has at most one shared page. If a process already has a shared page, either because it made a successful call to `shared_page` or because it inherited one from its parent, subsequent calls to `shared_page` should fail.
- Calls to `malloc`, `palloc_get_page`, and `pagedir_set_page` always succeed (i.e., the system does not run out of memory). This is designed to reduce how much error-handling code you need to write. You should still deallocate memory as appropriate (e.g., `free` what you `malloc`).

If a page is shared among multiple processes, and one of the processes exits, the shared page should remain accessible to the other processes. The page should be deallocated when the last process with access to the page exits.

You may wish to look at all parts of this question first, before starting to answer it. **If you believe that no code is needed for one or more parts of this question, it is fine to just leave those sections of the code blank. If you wish, you may write "NO CODE" in the appropriate code blocks for clarity.** You may not need to use all of the blank lines. It is also fine if your solution has more lines than the number of blank lines provided.

Recall that, in the course of completing Project 1, Bobby already implemented the following useful functions for the group:

```
void validate_user_buffer(void* pointer, size_t length);
void validate_user_string(const char* string);
```

These functions check if the provided buffer (or string) exists entirely within valid user-accessible memory. If not, they terminate the calling process with exit code `-1`.

i. Extend the system call handler to support the new `shared_page` system call. You may make calls to `validate_user_buffer` and/or `validate_user_string` in the system call handler.

```
static void syscall_handler (struct intr_frame* f) {
    uint32_t* args = ((uint32_t*) f->esp);
    validate_user_buffer(args, sizeof(uint32_t));

    switch (args[0]) {
        /* Pre-existing cases (not shown) */
        ...

    case SYS_SHARED_PAGE:

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________
        f->eax = (uint32_t) syscall_shared_page((void*) args[1]);
        break;

        /* Additional pre-existing cases (not shown) */
        ...
    }
}
```

ii. Determine how to extend `struct thread` to support the `shared_page` system call. If you need to define any new `struct`s or `union`s, do so on the lines above the `struct thread` definition (there is at least one good solution that does not require any additional `struct`s or `union`s, but there are also good solutions that do).

```
struct thread {
    /* Pre-existing members (not shown) */
    ...

    /* Add additional members on the lines below. */







    unsigned magic;
};
```

iii. Implement the system call handler for the `shared_page` system call.

```
void* syscall_shared_page(void* address) {
    struct thread* current = thread_current();
    if (pg_ofs(address) != 0) {
        return NULL;
    }
    if (pagedir_get_page(current->pagedir, address) != NULL) {
        return NULL;
    }

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
    return address;
}
```

iv. Implement `inherit_shared_page`. If the `parent` has a shared page, either because it allocated one with a `shared_page` system call or because it inherited one from its parent, then the `child` process should inherit a shared page from the `parent`. Otherwise, the `inherit_shared_page` function should do nothing.

```
void inherit_shared_page(struct thread* parent, struct thread* child) {

}
```

v. In their implementation of the `exec` system call, the group uses a semaphore to ensure that the parent process waits until the child process is fully loaded before returning fro `exec`. At which point(s) during the `exec` system call is it correct to call `inherit_shared_page`? Check **all** that apply.

□ In the parent process before calling `thread_create`.

□ In the parent process after `thread_create` returns but before calling `sema_down`.

□ In the parent process after `sema_down` returns but before `exec` returns.

□ In the child process before calling `load`.

□ In the child process after calling `load` but before calling `sema_up`.

□ In the child process after `sema_up` returns.

vi. Modify `process_exit` as necessary to support the `shared_page` system call. *Hint: Remember that `pagedir_destroy` frees all pages mapped in the page table.*

```
void process_exit(void) {
    struct thread* cur = thread_current();

    /* Existing code for Project 1 (not shown). */
    ...

    if (____________________________________________________________) {

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________

        ________________________________________________________________
    }

    /* This code is in the Pintos starter code. */
    uint32_t* pd = cur->pagedir;
    if (pd != NULL) {
        cur->pagedir = NULL;
        pagedir_activate(NULL);
        pagedir_destroy(pd);
    }
}
```

12. (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) What was your favorite topic in CS 162?

(c) (0 points) What was your least favorite part of CS 162?

(d) (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.

```
/******************************* 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);
```