Computer Science 162
Sam Kumar
University of California, Berkeley
Quiz 3
August 3, 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 quiz, you may **not** communicate with other people regarding the quiz 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 80 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 quiz, 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	Total
Points:	1	15	20	18	11	15	0	80
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. (15 points) **Operating System Concepts**

Choose **either** True or False for the questions below. You do not need to provide justifications. **Each student received 15 questions, chosen randomly from the ones below.**

(a) (1 point) Segmentation (without paging) is prone to internal fragmentation.

√ True
◯ False

(b) (1 point) Segmentation (without paging) allows different processes to share parts of their address space.

√ True
◯ False

(c) (1 point) Paging is prone to internal fragmentation.

√ True
◯ False

(d) (1 point) For a fully associative cache using a Least-Recently-Used (LRU) eviction policy, increasing the cache size may decrease the hit rate.

◯ True
√ False

(e) (1 point) For a fully associative cache using a First-In-First-Out (FIFO) eviction policy, increasing the cache size may decrease the hit rate.

√ True
◯ False

(f) (1 point) It is possible for a single memory access to result in a TLB hit and a page fault.

√ True
◯ False

(g) (1 point) It is possible for a single memory access to result in a TLB miss and a cache hit.

√ True
◯ False

(h) (1 point) Invalid PTEs may be cached in the TLB.

◯ True
√ False

(i) (1 point) On every page fault, the OS either brings in a page from disk to memory or terminates the process that issued the memory access.

◯ True
√ False

(j) (1 point) While the OS brings a page in from disk to memory in response to a page fault, the thread that caused the page fault is blocked, allowing the OS to schedule another thread in its place.

√ True
◯ False

(k) (1 point) The clock algorithm will evict exactly the same pages as LRU, without the need to manipulate a linked list on every access.

◯ True

√ False

(l) (1 point) While a process is executing on the CPU, the page table for its address space must be held entirely in physical memory.

◯ True

√ False

(m) (1 point) A user program may interact directly with certain I/O devices using port-mapped I/O (i.e., issuing `in` and `out` instructions directly).

◯ True

√ False

(n) (1 point) The CPU may only issue commands to a device using memory-mapped I/O if the device is attached directly to the memory bus, without any intervening buses, bridges, or bus adapters.

◯ True

√ False

(o) (1 point) The commodity storage drives with largest storage capacity are Hard-Disk Drives (HDDs).

◯ True

√ False

(p) (1 point) Hard-Disk Drives typically provide similar performance on random access patterns as they do on sequential scans.

◯ True

√ False

(q) (1 point) The operating system is responsible for remapping pages on a Solid-State Drive (SSD) for the purpose of wear leveling.

◯ True

√ False

(r) (1 point) In the FAT file system, file metadata is stored directly in directory entries.

√ True

◯ False

(s) (1 point) In Berkeley FFS, file metadata is stored directly in the directory entries.

◯ True

√ False

(t) (1 point) In Berkeley FFS, a highly fragmented file may span multiple inodes in the inode array.

◯ True

√ False

(u) (1 point) In Windows NTFS, a highly fragmented file may span multiple entries in the Master File Table.

√ True
◯ False

(v) (1 point) A soft link in one file system may refer to a file in a different file system.

√ True
◯ False

(w) (1 point) One file system may contain a hard link to a file in a different file system.

◯ True
√ False

(x) (1 point) Unix-like operating systems check a file's permissions on `open` system calls, but not on `read` and `write` system calls.

√ True
◯ False

(y) (1 point) In Unix-like operating systems, the data passed to `write` may not be visible to subsequent `read` syscalls to the same file, if the written data is in the buffer cache and has not yet reached the storage device.

◯ True
√ False

3. (20 points) **File Systems**

Consider system with the following properties:

- The file system is Berkeley FFS with 10 direct pointers, and 1 each of indirect, doubly indirect, and triply indirect pointers.
- The block size is 512 bytes.
- Each disk address is 4 bytes.
- Assume that the contents of each directory fit within a single disk block.
- The buffer cache is initially empty, but is large enough to simultaneously contain all disk blocks accessed in this question.
- The buffer cache is not flushed until all disk I/Os for this question are complete.

Suppose that a process `open`s the file `/cs162/file.txt` and `read`s the first 5633 bytes of the file.

(a) (2 points) Where does the OS look up the inode number of the root directory?

Solution: The root directory is at a known inumber specific to the file system. (Alternatively, saying that the OS looks up this information at a superblock at a known disk address is also acceptable.)

(b) (4 points) How many disk reads are issued by the `open` system call? For each disk read, explain its purpose.

Solution: 5 total disk reads. Read root inode, read `cs162` directory entry, read `cs162` inode, read `file.txt` directory entry, read `file.txt` inode.

(c) (4 points) How many disk reads are issued by the `read` system call(s)? Assume the inode for `file.txt` is already in the buffer cache. Explain how you calculated your answer.

Solution: 13 disk reads. $5633 = 512 * 11 + 1$, so 12 data blocks must be read. There are only 10 direct pointers in the inode, so the indirect block must be accessed, adding one additional disk access.

(d) (4 points) Suppose that the file system is stored on a Hard Disk Drive (HDD). List two techniques used by Berkeley FFS to reduce the seek time when performing the above operations.

Solution: Berkeley FFS organizes the file system into block groups, so that blocks that are likely to be accessed together are nearby (for example, the inode for `file.txt` will be near the file's data). Additionally, Berkeley FFS uses first-fit block allocation so that the data blocks for `file.txt` are likely to be contiguous on disk.

(e) (3 points) Of the four components of a file system (Directory Structure, Index, File Data Blocks, Free Blocks), which components must the operating system access when the above process issues the `open` system call? (You should count hits in the buffer cache as accesses to the component to which the block belongs.)

- √ Directory Structure
- √ Index
- ☐ File Data Blocks
- ☐ Free Blocks
- ☐ None of These

Each student received **one** of the following two questions, selected randomly.

(f) (3 points) Of the four components of a file system (Directory Structure, Index, File Data Blocks, Free Blocks), which components must the operating system access when the above process issues the `read` system call(s)? (You should count hits in the buffer cache as accesses to the component to which the block belongs.)

- ☐ Directory Structure
- √ Index
- √ File Data Blocks
- ☐ Free Blocks
- ☐ None of These

(g) (3 points) Suppose that, after reading the file, the process seeks to a known offset in the file by issuing an `lseek` system call with the `SEEK_SET` flag. Of the four components of a file system (Directory Structure, Index, Data Blocks, Free Blocks), which components must the operating system access when the above process issues the `lseek` system call? (You should count hits in the buffer cache as accesses to the component to which the block belongs.)

- ☐ Directory Structure
- ☐ Index
- ☐ File Data Blocks
- ☐ Free Blocks
- √ None of These

4. (18 points) **Demand Paging**

Each student received either Part a or Part b, not both.

(a) (3 points) Under what workload characteristics does LRU perform comparably to MIN, the optimal page replacement algorithm?

Solution: LRU performs comparably to MIN for workloads exhibiting temporal locality. Under these assumptions, pages that have not been accessed in a long time (the least recently used pages in memory) won't be used for a very long time in the future, meaning that they are good candidates for eviction.

(b) (3 points) Are there any workload characteristics for which FIFO would outperform LRU? Explain.

Solution: For workloads that exhibit very poor temporal locality, FIFO may outperform LRU. This is because LRU is only an approximation to MIN (the optimal algorithm) under the assumption of temporal locality.

(c) (4 points) Before implementing a virtual memory system, you measure that accessing a word that is resident in physical memory takes about 10 ns (with the hardware cache and TLB taken into account). You measure the time it takes the OS to service an access to an on-disk page (including page fault overhead, fetching the page from disk, and re-executing the instruction) to be about 10 ms. What is the highest page fault rate we can tolerate to prevent the average memory access time from exceeding 20 ns? Explain how you calculated your answer.

Solution: 0.0001%. We get the equation $10 + 10000000r = 20$, and solving for r gives us the answer.

(d) (3 points) Suppose that, after implementing a virtual memory system, you find that it is thrashing on certain workloads. What changes might you make to the operating system to mitigate the problem?

Solution: Run fewer processes at a time. For example, suspend some processes for an extended period of time so that the working sets of others fit in memory simultaneously, allowing them to make progress.

For the remainder of this question, consider a virtual memory system based on the Second-Chance List algorithm discussed in class, with the following change. When a page that is not resident in memory is accessed, it is moved to the *back* of the active list, instead of the front of the active list. Accesses to a page on the second-chance list work just as explained in class—the accessed page is moved to the *front* of the active list. In both cases, the item previously at the back of the active list is moved to the the second-chance list, and the least-recently-accessed item in the second-chance list is evicted.

We will refer to this modified algorithm as "Modified Second-Chance List" (MSCL), and the original unmodified version we covered in class as simply "Second-Chance List" (SCL).

(e) (2 points) Suppose that an on-disk page was just accessed, and that the OS has just finished bringing it into physical memory, according to MSCL. Should the OS mark the PTE referencing this page valid or invalid? Explain.

Solution: Valid, because the page is on the active list. (If we marked it as invalid, then we would just fault again when attempting to restart the program.)

(f) (3 points) Consider a workload where memory accesses are i.i.d. according to a Zipf distribution (i.e., the page accessed by each each memory access follows a Zipf distribution, and furthermore each access is independent of all others). Explain why MSCL might outperform SCL for this workload.

Solution: With the Zipf distribution, there are a few pages that are very popular, but a significant fraction of accesses are to pages that are not popular. Because the accesses are independent (no temporal locality), recently accessed unpopular pages will not be accessed again for a very long time. By moving pages from disk to the *back* of the active list, MSCL ensures that such pages leave the active list quickly. The same will happen for popular pages, except that popular pages will likely be accessed again while they are on the second chance list, and therefore they will be moved to the *front* of the active list and spend a long time there. Thus, MSCL will try to fill the active list with the most popular pages in the Zipf distribution.

(g) (3 points) Under what workload characteristics might SCL outperform MSCL?

Solution: SCL might outperform MSCL for workloads that exhibit temporal locality—a page that was just accessed is likely to be accessed again in the near future, even after the next page fault.

Commentary:
One could potentially obtain more flexibility by splitting the second-chance list into two pieces. If a page near the front of the second-chance list is accessed, it is moved to the front of the active list, but if a page near the back of the second-chance list is accessed, it is moved to the back of the active list. The threshold between the front and back of the second-chance list would be a configurable parameter. This would decouple the threshold of what is considered a "frequent" access from the length of the second-chance list.

5. (11 points) **Early Alarm**

Bobby, William, Jonathan, and Kevin are in a group for their CS 162 project, and they have just finished Project 2. William would like to extend the Alarm Clock functionality from Project 2 by implementing `void wakeup_early(void)`, a function that wakes up *all* threads currently blocked due to a call to `timer_sleep`, even if they have not slept for their full duration. **The `wakeup_early` function must behave correctly whether it is called by a thread or an interrupt handler.**

In their Project 2 implementation, the students declare and initialize the following global variable, intended to be a list of sleeping threads.

```
static struct list sleepers;
```

Additionally, they added the fields `int64_t awaketime;` and `struct list_elem sleepelem;` to `struct thread`, and they implemented the `timer_sleep` function as follows:

```
void timer_sleep(int64_t ticks) {
  int64_t start = timer_ticks();
  ASSERT(intr_get_level() == INTR_ON);
  struct thread* current = thread_current();
  enum intr_level old_level = intr_disable();
  current->awaketime = start + ticks;
  list_insert(&sleepers, &current->sleepelem);
  thread_block();
  intr_set_level(old_level);
}
```

Finally, as part of Project 2, the group modified the `timer_interrupt` function to awaken all sleeping threads whose `awaketime` is less than or equal to the current number of timer ticks. The code for this is not shown.

(a) (3 points) Jonathan suggests that, instead of introducing the new field `sleepelem`, the group could have just reused the existing `elem` field in `struct thread`. In particular, the `list_insert` call above could be replaced with `list_insert(&sleepers, &current->elem)`. Is Jonathan correct? Explain.

Solution: Yes, Jonathan is correct. Although `elem` is used for the waiters list in a semaphore and for the ready list, the thread being blocked will never be part of any of those lists while it is sleeping or executing the relevant code in `time_sleep`. Therefore, it is safe to reuse `elem` for the list of sleeping threads.

(b) (8 points) Now, implement the `void wakeup_early(void)` function that the group desires. **Your implementation should (1) work correctly with priority scheduling, and (2) work correctly in both thread context and external interrupt context**. If necessary, you may assume that the `priority` field of `struct thread` contains the thread's priority, including any priority donations.

Solution:

```
void wakeup_early(void) {
    enum intr_level old_level = intr_disable();
    while (!list_empty(&sleepers)) {
        struct list_entry* curr = list_pop_front(&sleepers);
        struct thread* t = list_entry(curr, struct thread, sleepelem);
        thread_unblock(t);
    }
    intr_set_level(old_level);
    if (intr_context()) {
        intr_yield_on_return();
    } else {
        thread_yield();
    }
}
```

Commentary:

One could potentially make the above solution more efficient by only yielding if one of the awakened threads has higher priority than the current thread. This was not required, however.

6. (15 points) **Address Translation**

There are ten versions of this problem. Each student received only one version. This document contains only one version.

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 (equal size at each level)
- 4-entry, fully associative TLB, unified for both instructions and data
- 32-bit page table entry (PTE) size

(a) (5 points) For the virtual address of the first instruction, `0x1084 0104`, fill in the value of each line. 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: 0x0000 0042 (or 0b0001 0000 10)

Byte offset of PTE in Root PT page: 0x0000 0108 (or 0b0001 0000 1000)

Index of entry in L2 Page Table: 0x0000 0040 (or 0b00 0100 0000)

Byte offset of PTE within L2 PT page: 0x0000 0100 (or 0b0001 0000 0000)

Code page offset: 0x0000 0104

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 page table base register (`cr3`), which contains the physical frame number of the root page table. The TLB is initially empty (i.e., all TLB entries are initially invalid). 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. For this problem, you may ignore all other flags in each page table entry.

Registers

cr3 (PTBR)	0x0001 0020
eax	0x1004 1108
ebx	0x1044 2108
ecx	0x1044 2100
edx	0x1044 2104

Instructions

```
0x1084 0104 movl (%ebx), %edx
0x1084 0106 movl (%ecx), %eax
```

TLB Contents

Valid	Tag	Frame
1	0x10840	0x10040
1	0x10442	0x10030
0		
0		

Physical Memory

Phys. Addr.	Contents
0x1001 0000	
...	
0x1001 0100	0x1044 0108
0x1001 0104	0x1084 2100
0x1001 0108	0x8001 0030
0x1002 0000	
...	
0x1002 0100	0x8001 0050
0x1002 0104	0x8001 0010
0x1002 0108	0x8001 0050
0x1003 0000	
...	
0x1003 0100	0x1084 2104
0x1003 0104	0x1084 2104
0x1003 0108	0x8001 0050
0x1004 0000	
...	
0x1004 0100	0x1084 0104
0x1004 0104	0x088b 1a8b
0x1004 0108	0x8001 0050
0x1005 0000	
...	
0x1005 0100	0x8001 0040
0x1005 0104	0x1004 1108
0x1005 0108	0x1004 2100

(b) (10 points) **You are to step through the instructions whose addresses and disassembly are shown on the previous page.** In the space provided below, write down the operation, address, and value associated with every memory access associated with the instructions, by filling in the blank cells in the table. For reads, the "Value" field should contain the data read from physical memory; for writes, it should contain the data written to physical memory. **Only include accesses to physical memory in the table; do not include changes to register values or updates to the TLB as separate rows in the table.** If you wish, you may update the state of the memory, registers, and TLB by writing on the figure on the previous page. **You may not need all of the rows provided in the table.** If a page fault occurs, then you should indicate in the "Comment" column that a page fault occurred and stop executing the instructions, leaving the remaining lines blank. Otherwise, filling out the "Comment" field is optional, and serves only to explain your reasoning to help us award partial credit.

Operation	Address	Value	TLB Hit?	Comment
Read root PT	0x1002 0108	0x8001 0050	No	Read valid PTE for top-level PT
Read L2 PT	0x1005 0100	0x8001 0040	No	Read valid PTE for code page
Read memory	0x1004 0104	0x1a8b	No	Fetch first instruction
Read root PT	0x1002 0104	0x8001 0010	No	Read valid PTE for top-level PT
Read L2 PT	0x1001 0108	0x8001 0030	No	Read valid PTE for data page
Read memory	0x1003 0108	0x8001 0050	No	Execute first instruction
Read memory	0x1004 0106	0x088b	Yes	Fetch second instruction
Read memory	0x1003 0100	0x1084 2104	Yes	Execute second instruction

 Student ID: _____

7. (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.

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