Computer Science 162
Sam Kumar
University of California, Berkeley
Quiz 2
July 20, 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	Total
Points:	1	10	25	20	24	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. (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) A successful call to `pthread_create` *always* issues a system call.

√ True
◯ False

(b) (1 point) A successful call to `pthread_mutex_lock` *always* issues a system call.

◯ True
√ False

(c) (1 point) It is possible to implement a mutex in userspace for kernel-provided threads (e.g., `pthreads`) that never issues a system call and never busy-waits.

◯ True
√ False

(d) (1 point) It is impossible for the kernel to switch to another thread while interrupts are disabled.

◯ True
√ False

(e) (1 point) It is possible to implement a lock using only semaphores (with no other calls to disable interrupts, acquire locks, use condition variables, or issue atomic read-modify-write instructions).

√ True
◯ False

(f) (1 point) It is possible to implement a condition variable using only semaphores (with no other calls to disable interrupts, acquire locks, use condition variables, or issue atomic read-modify-write instructions).

√ True
◯ False

(g) (1 point) If two threads belonging to the same process concurrently issue a `read` system call without proper synchronization in user space, kernel data structures may become corrupted.

◯ True
√ False

(h) (1 point) SRTF is an optimal solution to hard real-time scheduling on a single-core machine.

◯ True
√ False

(i) (1 point) It is possible to implement scheduling algorithms like MLFQ using a priority scheduler, by dynamically adjusting the priorities of threads.

√ True
◯ False

(j) (1 point) In Pintos, the `struct thread` representing a user process is *always* stored on the same page as the thread's user stack.

◯ True
√ False

(k) (1 point) In Pintos, the `struct thread` representing a user process is *always* stored on the same page as the thread's kernel stack.

√ True
◯ False

(l) (1 point) In general, a web server that spawns a new process to handle each request is well-conditioned.

◯ True
√ False

(m) (1 point) Modern operating systems, like Linux, use Banker's Algorithm to avoid deadlock.

◯ True
√ False

(n) (1 point) In Stride Scheduling, a job with a lower stride is given more CPU time than a job with a higher stride, on average.

√ True
◯ False

(o) (1 point) Priority donation solves the priority inversion problem by allowing a high priority thread to take a lock away from a low priority thread before the low priority thread releases the lock.

◯ True
√ False

3. (25 points) **Short Answer**

Answer the following questions. **Each student received 7 questions, taken from the ones below as follows: one of a or b, one of c or d, one of e or f, one of h or i, one of j or k, and g and l.**

(a) (4 points) The CS 162 TAs set up a search engine service, similar to Google. They measured that, on average, the system is able to process 100 requests per second, and that the average latency for each request is 162 ms. On average, how many requests are being processed by the system at any given instant? Explain.

Solution: Little's Law gives that there are about 16.2 requests in the system, on average. We computed this by multiplying 100 requests per second by 162 ms.

(b) (4 points) List three techniques to reduce context switch overhead in a highly concurrent system.

Solution: Here are some possibilities:

- Use fewer kernel threads (e.g., event-driven programming, or bounded thread pools).
- Reduce the number of mode switches (e.g., user-level scheduling).
- Use a scheduling algorithm that switches threads less (e.g., FIFO instead of RR).

Each bullet point encompasses multiple possible answers. For example, listing bounded thread pools, event-driven programming, and user-mode threads would receive full credit.

(c) (3 points) How can the kernel synchronize access to data shared by a thread and an interrupt handler?

Solution: Disable interrupts when accessing the data from a thread.

(d) (3 points) How can a user program synchronize access to data shared by a thread and a signal handler?

Solution: Block (disable) signals when accessing the data from a thread.

(e) (4 points) Is there ever a situation where busy-waiting is more efficient than putting a thread to sleep and waking it up with an interrupt? List such a situation, or explain why

none exists.

Solution: If the time the thread needs to sleep is less than the time it takes to put the thread to sleep and wake it up, then busy-waiting is more efficient. One such situation is a when a thread needs to enter a short critical section that is currently occupied by a thread currently executing on another core.

(f) (4 points) Why might finer-grained locking lead to improved performance?

Solution: Finer-grained locking makes locks less contended, resulting in less time waiting for a lock. Additionally, on a multi-core system, a finer-grained locking allows for more parallelism.

(g) (4 points) Consider the following code to acquire a spin-lock:

```
void wait_until_two_then_increment(int* var) {
    while (!compare_and_swap(var, 2, 3)) {}
}
```

Modify the the `wait_until_two_then_increment` function to busy-wait more efficiently. Write your new implementation of the function below. It should continue to busy-wait; do not suspend the calling thread. Hint: `compare_and_swap` is an atomic read-modify-write instruction. `compare_and_swap(var, 2, 3)` atomically: (1) returns the value of `*var == 2` and (2) if it returns `true`, it sets `*var = 3`.

Solution: Before issuing an atomic read-modify-write instruction, first check the value of `var` using a regular read-memory instruction, and only issue `compare_and_swap` if the read gave the desired value of 2. For example:

```
void wait_until_two_then_increment(int* var) {
    do {
        while (var != 2) {}
    } while (!compare_and_swap(var, 2, 3));
}
```

(h) (3 points) In what sense is EDF optimal?

Solution: In a hard real-time scheduling, EDF will schedule jobs on a single core in such a way that all jobs meet their deadlines, assuming it is possible to do so.

(i) (3 points) In what sense is SRTF optimal?

Solution: SRTF will produce a schedule with the best possible average response time, among pre-emptive schedulers.

(j) (3 points) Explain how MLFQ maintains responsiveness of interactive tasks.

Solution: Interactive tasks tend to have short CPU bursts that expire before the time quantum, so they will remain in a high-priority queue. In contrast, compute-bound jobs will sink to a low-priority queue because their time quanta will expire.

(k) (3 points) Under which metric does FCFS outperform round robin?

Solution: It optimizes for throughput. minimizes context switch overhead and optimizes cache behavior by running each job to completion.

(l) (4 points) For each of the three scheduling metrics we discussed in class—throughput, timeliness, and fairness—explain how Linux CFS caters to each one.

Solution:

- Fairness: Linux CFS schedules threads so that their virtual runtimes increase together and remain approximately equal.
- Timeliness: Linux CFS maintains target latency, adjusting the time quantum according to the number of threads so that all threads are serviced within the target latency.
- Throughput: Linux CFS ensures that the time quantum does not fall below the minimum granularity, to prevent context switch overhead from degrading throughput.

4. (20 points) **Deadlock and Scheduling**

Consider the following C program.

```
pthread_mutex_t x, y, z;
void* a(void* arg) {
    pthread_mutex_lock(&y);
    pthread_mutex_lock(&z);
    pthread_mutex_unlock(&y);
    pthread_mutex_unlock(&z);
    return NULL;
}
void* b(void* arg) {
    pthread_mutex_lock(&z);
    pthread_mutex_lock(&x);
    pthread_mutex_unlock(&z);
    pthread_mutex_unlock(&x);
    return NULL;
}
void* c(void* arg) {
    pthread_mutex_lock(&x);
    pthread_mutex_lock(&y);
    pthread_mutex_unlock(&x);
    pthread_mutex_unlock(&y);
    return NULL;
}
int main(int argc, char** argv) {
    pthread_mutex_init(&x, NULL);
    pthread_mutex_init(&y, NULL);
    pthread_mutex_init(&z, NULL);
    pthread_t a_thread, b_thread, c_thread;
    pthread_create(&a_thread, NULL, a, NULL);
    pthread_create(&b_thread, NULL, b, NULL);
    pthread_create(&c_thread, NULL, c, NULL);
    pthread_join(&a_thread, NULL);
    pthread_join(&b_thread, NULL);
    pthread_join(&c_thread, NULL);
    return 0;
}
```

(a) (3 points) Provide a valid execution order that results in deadlock. By execution order, we mean the sequence in which threads acquire and release locks. Provide your answer as a list of statements of the form "`THREAD` calls `lock_acquire(LOCK)`" or "`THREAD` calls `lock_release(LOCK)`" (where `THREAD` is one of `a`, `b`, or `c` and `LOCK` is one of `x`, `y`, or `z`). **Include calls to `lock_acquire` that block.**

Solution: `a` calls `lock_acquire(&y)`, `b` calls `lock_acquire(&z)`, `c` calls `lock_acquire(&x)`, `a` calls `lock_acquire(&z)`, `b` calls `lock_acquire(&x)`, `c` calls `lock_acquire(&y)`

(b) (4 points) In the execution order you provided in the previous part, what is the first *unsafe* state and where does it occur in the sequence?

Solution: The first unsafe state in the above execution order is the state where `a` holds `y`, `b` holds `z`, and `c` holds `x`. It occurs after the first three statements.

(c) (3 points) Is it possible to resolve deadlock by changing the order in which each thread *unlocks* the mutexes? Either provide such an ordering, or explain why one does not exist.

Solution: No, because the deadlock occurs before any thread unlocks any mutexes.

(d) (3 points) Suppose the underlying operating system schedules threads for execution using a FCFS scheduler. Is the program prone to deadlock for this particular scheduler? Either provide an execution order that results in deadlock, or explain why it is not possible.

Solution: No. A FCFS scheduler runs each thread to completion. Therefore, there is no contention for the locks (all lock operations do not block).

(e) (3 points) Would using a preemptive thread scheduler eliminate the possibility of deadlock?

Solution: No, it would not eliminate the possibility of deadlock. Although a preemptive scheduler would preempt the CPU from a thread, it will not preempt locks from threads that hold them.

(f) (4 points) A student in CS 162 attempts to fix the above code, so that it is deadlock-free regardless of the scheduler, by rewriting the function `c` as follows:

```
void* c(void* arg) {
    pthread_mutex_lock(&y);
    pthread_mutex_lock(&x);
    pthread_mutex_unlock(&x);
    pthread_mutex_unlock(&y);
    return NULL;
}
```

Is the resulting code deadlock-free? If not, provide an execution order that results in deadlock. If so, explain why the program is deadlock-free.

Solution: Yes, the resulting code is deadlock-free because it follows a consistent lock ordering (`y`, `z`, `x`), eliminating circular wait.

5. (24 points) **Single Shared Pipe**

Bobby, William, Jonathan, and Kevin are in a group for their CS 162 project, and they have just finished Project 1. Bobby, being an eager student, wants to extend Pintos by adding additional support for inter-process communication (IPC).

(a) (3 points) William suggests adding a `pipe` system call to Pintos. It would work the same way as in Linux; a pipe is created in the kernel, and the calling process obtains read and write file descriptors to access the pipe. What else must be implemented in Pintos to use a pipe for communication between two *different* processes?

Solution: No, it cannot be used for IPC in Pintos. For the `pipe` syscall to be useful, we must allow each child process to inherit file descriptors from its parent in Pintos. Otherwise, there is no way for two different processes to obtain file descriptors for the *same* pipe.

(b) (3 points) Jonathan suggests implementing sockets in Pintos. Sockets would use the same APIs as in Linux. Would using sockets for communication between two *different* processes require the same change you identified in the previous part? Explain why or why not.

Solution: No. A connection can be established without the child process inheriting file descriptors, by calling `bind`, `listen`, and `accept` in one process and `connect` in the other.

(c) (18 points) Kevin suggests modifying the pipe abstraction so that it can be used for IPC in Pintos. Instead of there being a `pipe` system call to create a pipe, there exists a single global pipe shared by all processes in the system. Any process can write a byte into the pipe by issuing a `pipe_write` system call, and any process can read a byte from the pipe by issuing a `pipe_read` system call. The interface to the system calls is as follows:

```
/* Writes the byte BYTE to the pipe. */
void pipe_write(uint8_t byte);

/* Reads a byte from the pipe and returns it. */
uint8_t pipe_read();
```

The pipe has the same semantics as a pipe in Linux. In particular:

- If the pipe is full, writes block until space is available to complete the write.
- If the pipe is empty, reads block until data is written to the pipe.
- No data should be read from the pipe twice; once one process reads data from the pipe, that data is no longer in the pipe and cannot be read by other processes.

In principle, the capacity of the pipe could be any nonnegative number of bytes. **For the purpose of this question, the capacity of the pipe should be 162.**

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.**

As part of implementing Project 1, Bobby implemented the following useful functions:

```
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. Implement the system call handler for the `pipe_write` and `pipe_read` syscalls below. You may make calls to `validate_user_buffer` and/or `validate_user_string`. You may not need all of the blank lines.

```
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_PIPE_WRITE:

        validate_user_buffer(&args[1], sizeof(uint32_t));

        syscall_pipe_write((uint8_t) args[1]);
        break;

    case SYS_PIPE_READ:

        f->eax = (uint32_t) syscall_pipe_read();
        break;

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

ii. Determine how to extend `struct thread` to support Kevin's IPC functionality. You may not need to use all of the blank lines.

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

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

    _____

    _____

    _____

    _____

    _____

    _____

    _____

    _____

    unsigned magic;
};
```

iii. Add any necessary global variables to `syscall.c`, and extend `syscall_init` to do any necessary initialization work. You may not need to use all of the blank lines.

```
/* This is used by the group's Project 1 implementation. */
static struct lock fs_lock;

/* Add additional global variables on the lines below. */

static char pipe_buffer[162];

static int pipe_read_idx;

static int pipe_write_idx;

static bool pipe_empty;

static struct lock pipe_lock;

static struct condition no_longer_empty;

static struct condition no_longer_full;

void syscall_init(void) {
    intr_register_int(0x30, 3, INTR_ON, syscall_handler, "syscall");
    lock_init(&fs_lock);

    /* Perform any necessary initialization work. */

    pipe_read_idx = 0;

    pipe_write_idx = 0;

    pipe_empty = true;

    lock_init(&pipe_lock);

    cond_init(&no_longer_empty);

    cond_init(&no_longer_full);


}
```

iv. Implement `syscall_pipe_write`. You may not need to use all of the blank lines.

```
void syscall_pipe_write(uint8_t byte) {
    lock_acquire(&pipe_lock);
    while (pipe_write_idx == pipe_read_idx && !pipe_empty) {
        cond_wait(&no_longer_full, &pipe_lock);
    }
    pipe_buffer[pipe_write_idx] = byte;
    pipe_write_idx = (pipe_write_idx + 1) % 162;
    pipe_empty = false;
    cond_signal(&no_longer_empty);
    lock_release(&pipe_lock);
}
```

v. Implement `syscall_pipe_read`. You may not need to use all of the blank lines.

```
uint8_t syscall_pipe_read() {
    lock_acquire(&pipe_lock);
    while (pipe_read_idx == pipe_write_idx && pipe_empty) {
        cond_wait(&no_longer_empty, &pipe_lock);
    }
    uint8_t rv = pipe_buffer[pipe_read_idx];
    pipe_read_idx = (pipe_read_idx + 1) % 162;
    pipe_empty = (pipe_read_idx == pipe_write_idx);
    cond_signal(&no_longer_full);
```

```
    lock_release(&pipe_lock);

    return rv;
}
```

vi. Add code to the beginning of `process_exit`, if necessary for your design.

```
void process_exit(void) {

    /* Pre-existing code (not shown) */
    ...
}
```

Commentary:
The solution given above uses condition variables. This is in line with the textbook's philosophy that one should prefer locks and condition variables to semaphores. However, it uses an extra `bool` variable to keep track of whether the bounded buffer is empty or full, in the case that the two indices are equal (since the indices will be equal in both the empty case and the full case). One alternative is to allocate one extra byte in the array, and call a "full buffer" one where the write index is just before the read index. This prevents all elements of the array from being used simultaneously, but is potentially less error prone than adding another variable.
A particularly elegant solution is to use the "two-semaphore" approach we discussed in lecture (see Lecture 9 slides). If one uses this approach, then one doesn't even need the extra boolean, as one could rely entirely on the semaphores to prevent writes when the buffer is full or reads when the buffer is empty.

6. (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);
```