

**Computer Science 162, Fall 2014**  
**David Culler**  
**University of California, Berkeley**  
**Midterm 2**  
**November 14, 2014**

Name	
SID	
Login	
TA Name	
Section Time	

---

This is a closed book exam with one 2-sided page of notes permitted. It is intended to be a 80 minute exam. You have 80 minutes to complete it. The number at the beginning of each question indicates the points for that question. Write all of your answers directly on this paper. Make your answers as concise as possible. If there is something in a question that you believe is open to interpretation, please raise your hand to request clarification. When told to open the exam, put your login on every page and check that you have them all. (Final page is for reference.)

By my signature below, I swear that this exam is my own work. I have not obtained answers or partial answers from anyone. Furthermore, if I am taking the exam early, I promise not to discuss it with anyone prior to completion of the regular exam, and otherwise I have not discussed it with anyone who took the early alternate exam.

X\_\_\_\_\_

**Grade Table (for instructor use only)**

Question	Points	Score
1	25	
2	25	
3	25	
4	25	
Total:	100	

---

1. (25 points) **Operating Systems Concepts**

- (a) (20 points) Choose **either** true or false for the below questions. You do not need to provide justifications.
- i. (2 points) A thread that wants to signal other threads upon completion of a critical section should do so with a Condition Variable after releasing the lock.  
 True  
 False
  - ii. (2 points) By using an atomic read-modify-write instruction, a user thread can acquire a lock without entering the operating system.  
 True  
 False
  - iii. (2 points) With paged virtual address translation threads only share state when they are within the same process.  
 True  
 False
  - iv. (2 points) With paged virtual memory, an process can be started with none of its code or data pages being resident in memory.  
 True  
 False
  - v. (2 points) The size of page tables is always much smaller than the size of physical memory.  
 True  
 False
  - vi. (2 points) The offset within the page of a data element is the same in virtual memory page and physical memory page, regardless of whether a single level or multi-level page table is used.  
 True  
 False
  - vii. (2 points) With multi-level page tables, the TLB holds translations for each of the level.  
 True  
 False
  - viii. (2 points) In a Unix-style system user access rights are checked on file read and file write operations.  
 True  
 False
  - ix. (2 points) Compared to FIFO, disk scheduling using shortest-seek-time first reduces average seek time but also reduces fairness.  
 True  
 False
  - x. (2 points) If all the operations in a transaction are durably recorded in the log, the transaction can be applied to the disk store, before the final commit marker is placed in the log.  
 True  
 False

- (b) (5 points) Which of the following is true about File System operation? Select **all** the choices that apply.
- The directory structure can be searched on Open and need not be examined on Read or Write.
  - The user FILE \* object contains information about how the file is stored on disk.
  - SCAN has shorter average seek distance than C-SCAN because it services requests while the head is moving in either direction.
  - If the transaction log is stored on disk, requests must be written in order, as with FIFO scheduling.
  - File Index structures are optimized to handle small and large files in the same manner.



Registers

r0	0x0000 0000
r1	0x0000 0000
r2	0x1104 5110
r3	0x0000 0000

PC	0x1104 4110
----	-------------

PTBR	0x0001 0020
------	-------------

Disassembly of 1<sup>st</sup> Three Instructions

0x1104 4110: load r1, (r2)  
 0x1104 4114: add r2, r2, 4096  
 0x1104 4118: load r3, (r2)

TLB contents


Physical Memory

	0x1001 0000
0x8001 0030	0x1001 0110
0x8001 0050	0x1001 0114
0x0001 0040	0x1001 0118

	0x1002 0000
0x8001 0010	0x1002 0110
0x8001 0040	0x1002 0114
0x8001 8840	0x1002 0118

	0x1003 0000
0x8140 0000	0x1003 0110
0x0240 1000	0x1003 0114
0x8341 0000	0x1003 0118

	0x1004 0000
0x8001 0060	0x1004 0110
0x8001 0050	0x1004 0114
	0x1004 0118

	0x1005 0000
	0x1005 0110
0x8001 0050	0x1005 0114
0x0240 1000	0x1005 0118

	0x1006 0000
0x0000 1000	0x1006 0110
0x0000 1001	0x1006 0114
	0x1006 0118

Figure 1: Figure for question 2.

3. (25 points) **File Systems**

Consider the following file system code drawn from Pintos. You may assume all the code is run on intel x86 **32** bit architecture.

```
#define BLOCK_SIZE 4096

/* Block device that contains the file system. */
struct block *fs_device;

/* Reads sector SECTOR from BLOCK into BUFFER */
void block_read (struct block *block, block_sector_t sector, void *buffer);

/* Write sector SECTOR to BLOCK from BUFFER, mark block as not free */
void block_write (struct block *block, uint32_t sector, const void *buffer)

/* Returns one free block */
long get_free_block();

/* Marks given block as free */
void free_block(long bnum);

/* On-disk inode.
   Must be exactly BLOCK_SIZE bytes long. */
struct inode_disk
{
    uint32_t start;                /* First data sector. */
    long length;                  /* File size in bytes. */
    unsigned int magic;           /* Magic number. */
    uint32_t unused[1021];        /* Not used. */
};

/* In-memory inode. */
struct inode
{
    struct list_elem elem;        /* Element in inode list. */
    uint32_t sector;             /* Sector number of disk location. */
    int open_cnt;                /* Number of openers. */
    bool removed;                /* True if deleted, false otherwise. */
    int deny_write_cnt;          /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;       /* Inode content. */
};
```

- (a) (4 points) **In fifteen words or less**, what can you say about how files are allocated on disk in this file system.

*Solutions longer than 15 words will receive no credit*

- (b) (2 points) Recall that in the unix file system inodes had both direct and indirect block pointers. We decide in our file system we want our inodes to always be in one of two modes: either the maximum amount of direct block pointers or the maximum amount of singly indirect block pointers, but never mixed. Please modify the struct definitions below to allow that behavior. We have added the mode field for you. You may delete fields

**Make the minimum number of modifications necessary for proper functionality, extra modifications will incur a loss of points**

```

/* On-disk inode.
   Must be exactly BLOCK_SIZE bytes long. */
struct inode_disk
{
    uint32_t mode;                /* direct or indirect mode */
    -----;
    -----;
    -----;

    long length;                 /* File size in bytes. */
    unsigned int magic;          /* Magic number. */
    uint32_t unused[1021];       /* Not used. */
};

/* In-memory inode. */
struct inode
{
    struct list_elem elem;       /* Element in inode list. */
    uint32_t sector;             /* Sector number of disk location. */
    int open_cnt;                /* Number of openers. */
    bool removed;                /* True if deleted, false otherwise. */
    int deny_write_cnt;          /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;      /* Inode content. */
    -----;
    -----;
    -----;
    -----;
};

```

- (c) (2 points) What is the maximum theoretical file size in this file system now?

- (d) (5 points) Implement the function `fblock_to_dblock`. We have given you a skeleton. You need to complete the error checking and the operational parts.

```
/* Converts file_block_num (a block offset into the file) to an absolute disk_block_num to
   returns -1 in the case of error.
```

```
*/
```

```
long
```

```
fblock_to_dblock (struct inode *inode, int file_block_num)
```

```
{
```

```
    long bnum;
```

```
    if ( inode->data.mode ) {
```

```
        if ( _____ &&
```

```
            _____ ) {
```

```
            bnum = _____;
```

```
        } else {
```

```
            return -1;
```

```
        }
```

```
    } else {
```

```
        if ( _____ &&
```

```
            _____ ) {
```

```
            _____;
```

```
            _____;
```

```
            _____;
```

```
            block_read(_____);
```

```
            bnum = _____;
```

```
        } else {
```

```
            return -1;
```

```
        }
```

```
    }
```

```
    return bnum;
```

```
}
```



- (e) (4 points) Now implement `inode_block_read` to support the above implementation. You may assume `fblock_to_dblock` is implemented correctly.

```
/* Tries to read BLOCK_SIZE bytes from INODE into BUFFER from file_block_num
   Returns the number of bytes actually read, which may be less
   than BLOCK_SIZE if end of file or and error is reached.
   err is filled when an error occurs
*/
long
inode_block_read (struct inode *inode, void *buffer_, int file_block_num, int* err)
{
    long bytes_read = BLOCK_SIZE;
    long sector_idx = fblock_to_dblock(inode, file_block_num);

    if (_____ ) {
        _____;

        return 0;
    }

    if (_____ ) {
        bytes_read = _____;
    }
    block_read (fs_device, sector_idx, buffer);
    return bytes_read;
}
```

- (f) (4 points) Consider the below implementation of `inode_block_write`.

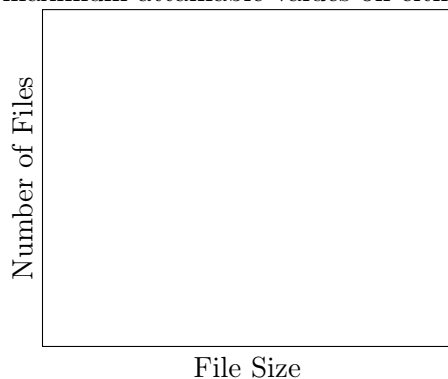
```
/* Writes BLOCK_SIZE bytes from BUFFER into INODE, starting at file_block_num*BLOCK_SIZE
all inputs are assumed to be valid and in bounds. Zero pads if file_block_num is greater
than file size.
*/
```

```
void inode_write_block (struct inode *inode, const void *buffer_, int file_block_num, int en
{

    long bnum = fblock_to_dblock(inode, file_block_num);
    if (file_block_num > inode->data.length/BLOCK_SIZE) {
        void* zeros = calloc(BLOCK_SIZE,1);
        for (int i = inode->data.length/BLOCK_SIZE + 1; i < file_block_num; ++i){
            block_write(fs, fblock_to_dblock(inode, i), zeros);
        }
    }
    block_write(fs_device, bnum, buffer);
    inode_recalc_length(inode); // updates length field if necessary.
}
```

Assuming all inputs to `inode_write_block` are valid and in bounds, **In forty words or less**, state whether the above implementation is correct (no explanation necessary), or precisely describe the steps necessary to fix the above implementation.

- (g) (4 points) Given that our superblock has a fixed size inode region  $2MB$  in size Draw a histogram on the graph below to reflect the usage pattern that this file system is optimized for. Remember to label the values on each axis so we can interpret your graph properly. Particularly consider the maximum attainable values on either axis.



4. (25 points) **Syscall Implementation**

This question requires you to implement the `dup2()` syscall.

According to the man page, `dup2(int oldfd, int newfd)` makes `newfd` become the copy of `oldfd`, closing `newfd` first if necessary. After a successful return the old and new file descriptors may be used interchangeably. If `oldfd` is not a valid file descriptor, the call fails and `newfd` is not closed. If `oldfd` is a valid file descriptor and `newfd` has the same value as `oldfd`, then `dup2()` does nothing. On success, these system calls return the new descriptor. On error, `-1` is returned.

- (a) (3 points) Below is a simple **Linux** C program which uses the `dup2()` syscall to redirect `stdin` so that executed process greps the word "cs162" from the file "cs162.txt". Fill in the blanks. Assume calls to `open` `close` and `execvp` succeed.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    char *grep_args[] = {"grep", "cs162", NULL};
    int in = open("cs162.txt", O_RDONLY);

    // replace standard input with input file

    dup2(_____);

    close(in);                // close unused file descriptors
    execvp("grep", grep_args); // execute grep
}
```

- (b) (5 points) Suppose you have to implement `dup2` in **Pintos**. The following code provides a framework for the **Pintos** implementation of `dup2`. You may assume `fs_lock` is properly initialized. Read it and answer the questions following.

```
struct thread
{
    ... /* Other members as in your pintos projects */

    /* Owned by syscall.c. */
    struct list fds;           /* List of open file descriptors. */
    int next_handle;         /* Look at the sys_open and sys_close code
                               in the appendix to figure out its purpose */
};

/* A file descriptor, for binding a file handle to a file object. */
struct file_descriptor
```

```
{
    struct list_elem elem;    /* List element */
    struct file *file;       /* File object */
    int handle;              /* File handle */
};

/* Returns the file descriptor associated with the given handle.
   Returns NULL if HANDLE is not associated with an
   open file. */

static struct file_descriptor *lookup_fd (int handle) {
    struct thread *cur = thread_current ();
    struct list_elem *e;

    for (e = list_begin (&cur->fds); e != list_end (&cur->fds);
         e = list_next (e))
    {
        struct file_descriptor *fd;
        fd = list_entry (e, struct file_descriptor, elem);
        if (fd->handle == handle)
            return fd;
    }
    return NULL;
}

static struct lock fs_lock;
```

For reference on how these structures can be used to implement the syscalls `open` and `close`, refer to the appendix.

- i. (3 points) What is the purpose of the lock in the functions `sys_open` and `sys_close` (given in the appendix?)

- ii. (2 points) What is `next_handle` member in `struct thread`?

- (c) (15 points) Given this setup, implement the `dup2` system call using the skeleton below. Assume that the global syscall table calls the required functions, and copies the returned value to the `eax` register. You do not have to set `errno`. Make sure your code has no memory leaks. You are NOT allowed to write more lines of code than the blanks given. Assume calls to `malloc` succeed, and user programs are single-threaded. ( Do not worry about modifying or bound-checking `next_handle` in this system call ).

```
static int sys_dup2(int oldfd, int newfd)
{
    /* Variable initialization
       nfd : new file_descriptor
       ofd : old file_decriptor
    */
    struct file_descriptor *nfd, *ofd;

    if (newfd == oldfd) return newfd;    /* handle special case */

    ofd = _____;

    if (ofd == NULL ) {
        _____;
    }

    nfd = _____;

    if (nfd == NULL ) {
        _____;
        _____;
        _____;
    } else {
        _____;
        _____;
        _____;
    }

    _____;

    return newfd;
}
```

- (d) (2 points) Consider the user program in part(a) running in Pintos as well as Linux. Why does this program work as expected (i.e grep has its input redirected from cs162.txt) in Linux but not in Pintos ? (Assume that `execvp` is replaced by `exec` in Pintos, and the arguments to `exec` are valid)

```
/******Reference Syscall Implementation *****/

static int sys_open (const char *ufile) {
    // Copy userspace pointer to kernel space
    char *kfile = copy_in_string (ufile);
    struct file_descriptor *fd;
    int handle = -1;

    fd = malloc (sizeof *fd);
    if (fd != NULL) {
        lock_acquire (&fs_lock);
        fd->file = filesys_open (kfile);
        if (fd->file != NULL)
            {
                struct thread *cur = thread_current ();
                handle = fd->handle = cur->next_handle++;
                list_push_front (&cur->fds, &fd->elem);
            }
        else
            free (fd);
        lock_release (&fs_lock);
    }
    palloc_free_page (kfile);
    return handle;
}

/* Close system call. */
static int
sys_close (int handle){
    struct file_descriptor *fd = lookup_fd (handle);
    lock_acquire (&fs_lock);
    file_close (fd->file);
    lock_release (&fs_lock);
    list_remove (&fd->elem);
    free (fd);
    return 0;
}

/****** String Processing *****/
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

**This page has intentionally been left blank**

**DO NOT WRITE ANSWERS ON THIS PAGE**