Computer Science 162
David E. Culler
University of California, Berkeley
Midterm Exam
October 10, 2019

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

This is a closed book exam with one 2-sided page of notes permitted. It is intended to be a 90 minute exam. You have 110 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 Student ID on every page and check that you have them all. The 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:	1	2	3	4	5	6	7	Total
Points:	1	20	20	19	20	20	0	100
Score:								

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

2. (20 points) **Operating System Concepts**

Choose **either** True or False for the questions below. You do not need to provide justifications.

(a) (1 point) Dual-mode operation allows the kernel to access memory of user processes, but prevents user processes from accessing memory of the kernel.

○ True

○ False

(b) (1 point) A thread can be a part of multiple processes.

○ True

○ False

(c) (1 point) In Unix-based operating systems, a process can read and write a file without opening it.

○ True

○ False

(d) (1 point) In Unix-based operating systems, a process can exit without first closing the files it has been reading/writing.

○ True

○ False

(e) (1 point) Switching between threads belonging to different processes typically has higher overhead than switching between threads belonging to the same process.

○ True

○ False

(f) (1 point) A user program may enforce a critical section by disabling interrupts.

○ True

○ False

(g) (1 point) A TCP socket (i.e., socket of type `SOCK_STREAM`) provides reliable, bidirectional communication between processes on different hosts.

○ True

○ False

(h) (1 point) A pipe (i.e., output of a single call to `pipe()`) provides reliable, bidirectional communication between processes on the same host.

○ True

○ False

(i) (1 point) An operating system provides a specific set of services to user programs, regardless of the programming language they are written in, through syscalls.

○ True

○ False

(j) (1 point) An appropriate way for the operating system and user programs to cooperate is to allow programs to read variables maintained by the kernel.

○ True

○ False

Choose **either** True or False for the questions below and provide an explanation.

(k) (2 points) A kernel interrupt handler is a thread.

○ True
○ False

(l) (2 points) Once a file is `open`ed, its entire contents can always be read into memory with a single call to `read`, assuming the file's contents fit in memory.

○ True
○ False

(m) (2 points) List the three most important attributes of a thread that the operating system must maintain when the thread is not running.

(n) (2 points) What is a process?

(o) (2 points) List what causes the transition from user mode to kernel mode.

What causes the transition from kernel mode to user mode?

3. (20 points) **Processes, Threads and Concurrency**

(a) (1 point) The scheduler in an operating system (check the **single** best answer):

○ May run any thread
○ Runs the highest priority ready thread
○ Selects the next ready thread to run according to a policy
○ Selects the thread that just completed I/O

(b) (2 points) The Base and Bound register approach provides which of the following (check **all** that apply):

□ Memory protection between processes
□ Memory sharing between processes
□ Expandable heap and stack
□ Protection of the kernel from user processes

(c) (2 points) The paged virtual address approach provides which of the following (check **all** that apply):

□ Memory protection between processes
□ Memory sharing between processes
□ Expandable heap and stack
□ Protection of the kernel from user processes

(d) (2 points) In your Project 1, Task 2, the Pintos `exec` syscall creates a child process and starts it running. What is the roughly equivalent sequence of syscalls (provided through the C system library, i.e., `unistd.h`) needed to achieve this functionality?

(e) (2 points) A common approach to ensuring that a resource is released only after all users of it are done is to include a reference count. For example, the object representing a parent process may need to persist until all its children have exited.

i. Why do we need to acquire a lock, or do an equivalent synchronization operation, before performing operations on this reference count?

ii. Describe another situation that occurs in systems implementation, other than a parent/child wait structure, where a reference count is useful.

(f) (2 points) A call to `pthread_mutex_lock` typically takes longer to return when there is contention for the mutex, than when the mutex is uncontended. Explain your answer.

○ True
○ False

(g) (2 points) For kernel threads, one can use atomic memory operations, like `test&set`, to implement a lock in a manner that does not spin-wait. Explain your answer.

○ True
○ False

(h) (2 points) For this question, assume calls to the `pthread` library always succeed.

```
void* worker(void* arg) {
    printf("%d", (int) arg);
    pthread_exit(NULL);
}
int main(int argc, char** argv) {
    pthread_t tid[4];
    int i;
    for (i = 0; i != 4; i++) {
        pthread_create(&tid[i], NULL, worker, (void*) (i + 1));
    }
    for (i = 0; i != 4; i++) {
        pthread_join(tid[i], NULL);
    }
    printf("%d", 5);
}
```

Which of these statements about the above program are true? Check **all** that apply:

□ When the program is run, its output *might* be `21345`.
□ When the program is run, its output *might* be `1235`.
□ The four worker threads *might* exit in the same order that they were created.
□ The four worker threads *might* exit in a different order than they were created.
□ When the `printf` statement in `main` is reached, all four worker threads are *guaranteed* to have called `pthread_exit`.
□ When the `printf` statement in `main` is reached, the operating system is *guaranteed* to have fully reclaimed the resources associated with all four worker threads.

(i) (2 points) Consider the following program:

```
void luffy() {
    sema_down(&loguetown);
    sema_up(&marineford);
    sema_down(&skypeia);
    printf("L: I found the One Piece!");
}

void blackbeard() {
    sema_down(&skypeia);
    sema_up(&skypeia);
    sema_down(&loguetown);
    printf("B: The One Piece is mine!");
}

void akainu() {
    sema_down(&marineford);
    sema_down(&loguetown);
    sema_up(&skypeia);
    printf("A: I’ve secured the One Piece.");
}

void main() {
    thread_create(luffy);
    thread_create(blackbeard);
    thread_create(akainu);
    exit(0);
}
```

i. Assume all semaphores are initialized to a value of one. Which of these lines could be printed? (check **all** that apply)

□ A: I’ve secured the One Piece.

□ B: The One Piece is mine!

□ L: I found the One Piece!

□ (nothing is printed)

ii. Is it possible for multiple of these lines to be printed in a single run of the program? If Yes, explain which and how. If No, explain why not.

○ Yes

○ No

(j) (3 points) Implement `strcpy` from the C standard library. You may assume that `src` and `dest` point to allocated memory buffers large enough to store the string in `src`, including its null terminator (but not necessarily any larger), and that the two buffers do not overlap. You may use `strlen`, but not `memcpy`.

```
char* strcpy(char* dest, const char* src) {
    ______________________________________________
    ______________________________________________
    ______________________________________________
    ______________________________________________
    ______________________________________________
    ______________________________________________
    ______________________________________________
    ______________________________________________
    return dest;
}
```

Explain why `strcpy` can be hazardous to use within the operating system.

4. (19 points) **Scheduling, Performance, and Deadlock**

(a) (10 points) Comparing schedulers

We are going to compare the behavior of four schedulers on a workload consisting of 4 tasks:

- Task A arrives at tick 0 and uses the CPU for 4 ticks.
- Task B arrives at tick 1 and uses the CPU for 7 ticks.
- Task C arrives at tick 2 and uses the CPU for 1 tick, issues I/O that takes 4 ticks, and then uses the CPU for 1 tick.
- Task D arrives at tick 3 and does 1 tick CPU, 1 tick I/O, 1 CPU, 1 tick I/O, and 1 tick CPU.

This workload is illustrated below as if each executes on its own dedicated CPU.

	0	1	2	3	4	5	6	7
A	a1	a2	a3	a4				
B		b1	b2	b3	b4	b5	b6	b7
C			c1	-	-	-	-	c6
D				d1	-	d3	-	d5

You are to schedule these onto a single CPU. When one does I/O, another ready task can run on the CPU. When a task completes it I/O ticks, it is ready to run on the CPU. If two tasks finish I/O in the same tick, the one that started I/O first is treated as arriving first.

Consider four schedulers: first-come-first-serve (FCFS), preemptive round-robin with a quanta of 2 ticks (RR-2), shortest-remaining-time-first with quanta of 2 where only remaining CPU time is considered, and completely fair scheduler (CFS) with a quanta of 2 where only the CPU time consumed is considered.

Below is a scratch area for you to keep track of what each scheduler runs when. We have filled in the FCFS scheduler for you. Note that D runs as soon as C blocks for I/O, and it gets to run again before C's longer I/O finishes, after its I/O completes. **We will not grade the grid**, but it may help you work out the questions.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
FCFS	a1	a2	a3	a4	b1	b2	b3	b4	b5	b6	b7	c1	d1	-	d3	-	c6	d5

Now, answer the following questions.

i. (1 point) All three preemptive schedulers completely mask the waits for I/O with CPU activity, resulting in 100% utilization.

○ True

○ False

ii. (1 point) In all the schedules, B starts running before C.

○ True

○ False

iii. (1 point) In all three preemptive schedulers, B is the last to finish.

○ True

○ False

iv. (1 point) A has a smaller wait time (number of ticks where it is ready to run, but waiting) in SRTF than it has in the other three schedulers.

○ True

○ False

v. (2 points) Which has the lowest average response time? (Response time of a task is defined as the number of ticks from its arrival to its completion.)

○ FCFS ○ RR ○ SRTF ○ CFS

vi. (2 points) Which has the smallest range of response times?

○ FCFS ○ RR ○ SRTF ○ CFS

vii. (2 points) Which has the smallest range of wait times?

○ FCFS ○ RR ○ SRTF ○ CFS

(b) (2 points) What is the difference between an open system and a closed system?

Give an example that behaves as a closed system.

(c) (2 points) List the four conditions required for deadlock to happen.

(d) (3 points) Each of the following three functions runs in a separate thread. Assume that the three threads are all created and ready to run.

```
mutex_t morph = INITIAL_FREE;
mutex_t levelup = INITIAL_FREE;
void Pikachu() {
    lock(&morph);
    lock(&levelup);
    store some electricity
    unlock(&levelup);
    unlock(&morph);
}
```

```
void Celebi() {
    lock(&morph);
    lock(&levelup);
    guard the ilex forest
    unlock(&morph);
    unlock(&levelup);
}
void Mew() {
    lock(&levelup);
    lock(&morph);
    become invisible
    unlock(&levelup);
    unlock(&morph);
}
```

i. Describe an execution sequence that would result in deadlock.

ii. Does preemptive scheduling solve the problem?

iii. Describe how you would fix the code so that deadlock would not occur.

(e) (2 points) If a thread reads a 10MB file with a transfer rate of 5MB/s with 100ms **total** overhead to start and complete the read, for how many 50ms quanta is the thread blocked?

If another thread was computing at a rate of 100 GFLOP/s, how many floating point operations could it get done in this time?

5. (20 points) **Operating System Implementation**

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

   (a) (2 points) List two types of IPC that are implemented in Pintos as part of Project 1.

   (b) (2 points) Kristoff 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. Can Kristoff's `pipe` system call be used for IPC in Pintos? If it can be used, explain how; if not, explain why not.

   (c) (16 points) Anna suggests providing IPC via a global key-value store. Each process can `PUT` a *single* key-value pair, if the key does not already exist, and can `GET` a value by looking it up by key. When a process exits, its key-value pair is removed from the key-value store.

   Keys and values are strings, whose lengths can be up to `MAX_KEY_LEN` and `MAX_VAL_LEN`, respectively. You may assume these constants are reasonably small (e.g., less than 162). The empty string may **not** be used as a key.

   You may wish to look at all parts of this question first, before starting to answer it.

   i. The group is trying to determine what new data structures they must add to Pintos to support this functionality. Help them finish their design by filling in the blank lines below in their new `struct kv`.

```
struct kv {

    char key[ ______________________________ ];

    char value[ ______________________________ ];

    struct list_elem elem;
};
```

ii. Determine how to extend `struct thread` to support Anna'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. Determine what additional global variables must be added 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. */

______________________________________________

______________________________________________

______________________________________________

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

    /* Perform any necessary initialization work. */

    ______________________________________________

    ______________________________________________

    ______________________________________________
}
```

As part of implementing Project 1, Elsa 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`.

iv. Anna decided on the following syscall interface:

```
/*
 * kv_get writes the value associated with KEY to VALUE, as a null-
 * terminated string. VALUE must point to a buffer large enough to
 * hold a maximum-length value. Returns 0 on success and -1 on failure.
 */
int kv_get(const char* key, char* value);

/*
 * kv_put maps KEY to VALUE. Both must be null-terminated strings.
 * Returns 0 on success and -1 on failure.
 */
int kv_put(const char* key, const char* value);
```

Implement the system call handler for these 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_KV_GET:

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________
        f->eax = syscall_kv_get((char*) args[1], (char*) args[2]);
        break;

    case SYS_KV_PUT:

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________
        f->eax = syscall_kv_put((char*) args[1], (char*) args[2]);
        break;

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

Elsa implemented the following helper function that you may find useful:

```
static struct kv* kv_find(const char* key) {
    struct list_elem* e;
    for (e = list_begin(&kv_list); e != list_end(&kv_list);
            e = list_next(e)) {
        struct kv* kvpair = list_entry(e, struct kv, elem);
        if (strcmp(kvpair->key, key) == 0) {
            return kvpair;
        }
    }
    return NULL;
}
```

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

```
int syscall_kv_get(const char* key, char* value) {
    struct thread* t = thread_current();

    ____________________________________________________________

    ____________________________________________________________
    if (strlen(key) == 0 || strlen(key) > MAX_KEY_LEN) {
        return -1;
    }

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
    struct kv* found = kv_find(key);
    if (found == NULL) {

        ________________________________________________________

        ________________________________________________________
    }
    strlcpy(value, found->value, sizeof(found->value));

    ____________________________________________________________
    return 0;
}
```

vi. Implement `syscall_kv_put`. You may not need to use all of the blank lines.

```
int syscall_kv_put(const char* key, const char* value) {
    struct thread* t = thread_current();

    ____________________________________________________________

    ____________________________________________________________
    if (strlen(key) == 0 || strlen(key) > MAX_KEY_LEN) {
        return -1;
    }

    ____________________________________________________________

    if (_______________________________________________) {

        _______________________________________________________
        return -1;
    }

    ____________________________________________________________

    struct kv* kvpair = ___________________________________;
    strlcpy(kvpair->key, key, sizeof(kvpair->key));
    strlcpy(kvpair->value, value, sizeof(kvpair->value));

    ____________________________________________________________

    ____________________________________________________________

    list_push_front(&kv_list, &kvpair->elem);

    ____________________________________________________________
    return 0;
}
```

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

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

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

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

6. (20 points) **Remote Procedure Calls**

Neo wants to extend an existing RPC server by adding support for a new operation:

```
uint32_t *read_int_list(char *path)
```

This operation reads the sequence of integers stored in the file `path` and returns them to the caller.

The server uses a protocol in which the first byte sent from the client to the server identifies which procedure to invoke. `read_int_list` will be uniquely identified by the value `162`.

The arguments for `read_int_list` are serialized as follows:

- A 4-byte integer representing the length of the `path` string, including the null terminator
- The contents of `path`, including the null terminator

The return value for `read_int_list` is serialized as follows:

- A byte representing the success or failure of the procedure call. `0` indicates success, while `-1` indicates failure
- If the procedure call was successful:
  - A 4-byte integer representing the number of integers read from `path`
  - The sequence of integers, each 4 bytes in length, contained in the file

You may find the following functions useful for serialization and deserialization:

- `uint32_t htonl(uint32_t host_long)` converts the unsigned integer `host_long` from host byte order to network byte order.
- `uint32_t ntohl(uint32_t net_long)` converts the unsigned integer `net_long` from network byte order to host byte order.

Neo already has an implementation of the client stub for `write_int_list`, shown below. For simplicity, you may assume that `read` and `write` always process the requested number of bytes.

```
int read_int_list_client_stub(struct addrinfo *server, char *path,
                              uint32_t **dest) {
    int sock_fd = socket(server->ai_family, server->ai_socktype,
                         server->ai_family);
    if (sock_fd == -1) {
        return -1;
    }
    if (connect(sock_fd, server->ai_addr, server->ai_addrlen) == -1) {
        return -1;
    }

    /* Send arguments to server */
    uint8_t op_code = 162;
    write(sock_fd, &op_code, sizeof(op_code));
    uint32_t path_len = htonl(strlen(path) + 1);
    write(sock_fd, &path_len, sizeof(path_len));
    write(sock_fd, path, strlen(path) + 1);

    /* Receive result from server */
    uint8_t status;
    read(sock_fd, &status, sizeof(status));
    if (status != 0) {
        close(sock_fd);
        return -1;
    }
    uint32_t num_ints;
    read(sock_fd, &num_ints, sizeof(num_ints));
    num_ints = ntohl(num_ints);
    *dest = malloc(num_ints * sizeof(uint32_t));
    read(sock_fd, *dest, num_ints * sizeof(uint32_t));
    for (uint32_t i = 0; i < len; i++) {
        (*dest)[i] = ntohl((*dest)[i]);
    }
    close(sock_fd);
    return num_ints;
}
```

(a) (5 points) Help Neo complete the implementation of the `read_int_list` function. This is executed on the server to read a sequence of integers from a specified file. It returns the number of integers read, or -1 upon error. Upon success, the pointer referred to by `dest` points to a copy of the integer sequence in memory.

Fill in the missing pieces of C code below. You may or may not need all lines. For simplicity, you may assume that `fread` and `fwrite` always succeed.

```
int *read_int_list(char *path, uint32_t **dest) {

    FILE *f = ________________________________________

    if (______________________________________) {
        return -1;
    }
    uint32_t num_ints;
    fread(&num_ints, sizeof(num_ints), 1, f);
    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________

    ______________________________________________
    return (int) num_ints;
}
```

(b) (10 points) Next, help Neo implement the server stub for the `read_int_list` operation. It reads the serialized arguments from the client, invokes `read_int_list`, and sends the serialized result back to the client. Fill in the missing pieces of C code below. You may or may not need all lines, and you may make the simplifying assumptions that **`read` and `write` always process the requested number of bytes** and that the client request is well-formed.

```
void read_int_list_server_stub(int sock_fd) {
    /* Receive arguments from client */
    uint32_t path_len;

    ________________________________________________________

    ________________________________________________________

    ________________________________________________________

    ________________________________________________________

    ________________________________________________________

    uint8_t result;
    uint32_t *ints;
    int num_ints = read_int_list(path, &ints);
    if (num_ints < 0) {

        ____________________________________________________

        ____________________________________________________

        ____________________________________________________

        return;
    }
    for (int i = 0; i < num_ints; i++) {
        ints[i] = htonl(ints[i]);
    }
    result = 0;

    ________________________________________________________

    ________________________________________________________

    ________________________________________________________

    ________________________________________________________

    ________________________________________________________

    ________________________________________________________

}
```

(c) (5 points) Finally, Neo wants to write the server code to handle client connections and dispatch to the proper server stub. He decides to handle each client connection in a separate child process. Fill in the missing pieces of server code in the `main` function. Assume you do not need to handle zombie processes. You may or may not need all blank lines.

```
const int8_t invalid_op_error = -1;

void serve_client(int client_socket) {
    uint8_t operation;
    read(client_socket, &operation, sizeof(operation));

    switch (operation) {
        / * Code to dispatch to other procedures not shown */

        case 162:
            read_int_list_server_stub(client_socket);
            break;
        default:
            write(client_socket, &invalid_op_error,
                  sizeof(invalid_op_error));
            break;
    }
    close(client_socket);
}
```

Fill in code on the next page ⟶

```
int main(int argc, char **argv) {
    struct addrinfo *server = setup_address(argv[1]);
    if (server == NULL) {
        return 1;
    }
    int server_socket = socket(server->ai_family, server->ai_socktype,
                               server->ai_protocol);
    if (server_socket == -1) {
        return 1;
    }
    if (bind(server_socket, server->ai_addr, server->ai_addrlen) == -1) {
        return 1;
    }
    if (listen(server_socket, 50) == -1) {
        return 1;
    }
    while (1) {
        int connection_socket = accept(server_socket, NULL, NULL);
        if (connection_socket == -1) {
            return 1;
        }
        pid_t pid = fork();

        if (___________________________________) {

            __________________________________________________________

            serve_client(connection_socket);

            __________________________________________________________

            __________________________________________________________

        } else if (___________________________________) {

            __________________________________________________________

            __________________________________________________________

            __________________________________________________________

            __________________________________________________________

        } else {
            return 1;
        }
    }
    return 0;
}
```

7. (0 points) If there's anything you'd like to tell the course staff (e.g., feedback about the class or exam, suspicious activity during the exam, new logo suggestions, etc.) you can write it on this page.

This page is intentionally left blank.

Do not write any answers on this page.

You may use it for *ungraded* scratch space.

```
/******************************* Threads *******************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int 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[]);
/*************************** 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);
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 *********************************/
void list_init(struct list *list);
struct list_elem *list_head(struct list *list)
struct list_elem *list_tail(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);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
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);
```