University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2023 Kubiatowicz

Midterm I

February 16th, 2023

CS162: Operating Systems and Systems Programming

Your Name:	
SID AND Autograder Login (e.g. student042):	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 1 page of notes (both sides). You have 110 minutes to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Make your answers as concise as possible. On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	20	
2	16	
3	16	
4	20	
5	28	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [20 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT*.

Problem 1a[2pts]: One process can have two different file descriptors (in the process's file descriptor table) that point to the same open file description structure in the kernel.

□ True □ False

Explain:

Problem 1b[2pts]: The up/down functions of a semaphore initialized with value 1 will always exhibit identical behavior to the release/acquire functions of a lock. (Assume that the lock enforces that a thread cannot release the lock unless it has already acquired it.)

□ True □ False

Explain:

Problem 1c[2pts]: The following code will print "**7**", assuming that fork() never fails:

```
void main(int argc, char **argv) {
    int count = 0;
    pid_t main_process_pid = getpid();
    for (int i = 0; i < 3; i++) {
        if (fork() == 0) count++;
        else wait(NULL);
    }
    if (getpid() == main_process_pid)
        printf("%d\n", count);
}
```

□ True □ False

Explain:

Problem 1d[2pts]: A thread cannot be blocked on multiple condition variables simultaneously.

□ True □ False

Explain:

Problem 1e[2pts]: Threads within the same process *can* share data (since they live in the same address space), but threads in different processes *cannot* share data.

□ True □ False

Explain:

Problem 1f[2pts]: There are situations where disabling interrupts *must* be used as opposed to other synchronization primitives.

□ True □ False

Explain:

Problem 1g[2pts]: The use of sockets with TCP/IP is limited to providing a single unique connection between each physical client and physical server.

□ True □ False

Explain:

Problem 1h[2pts]: System calls are achieved by library code (for instance, in `libc`) that first changes the processor mode from user mode to kernel mode (using one of the "switch processor mode" instructions), then makes an explicit call to a handler function in the kernel.

□ True □ False

Explain:

Problem 1i[2pts]: Suppose you write a multithreaded program that generates 1000 different threads to all read data within a shared array. To make sure that there are no race conditions between these threads, you must implement a synchronization method.

□ True □ False

Explain:

Problem 1j[2pts]: In Pintos, a child process's state (running status, exit code, etc) can never be freed until its parent process terminates.

□ True □ False

Explain:

Problem 2: Multiple Choice [16pts]

Problem 2a[2pts]: Select all true statements about processes (*choose all that apply*):

A: ☐ When a new process is created it must be initialized with a new virtual address space.

B: ☐ The PCB must be located in user memory for the user program to access the virtual address to physical address translation scheme (e.g., the base/bound or page directory).

C: ☐ When a parent process exits, all child processes must also immediately exit.

D: ☐ Immediately after fork completes, if a parent process and its new child process perform a read at the same virtual address they may end up reading from the same physical memory.

E: ☐ None of the above.

Problem 2b[2pts]:

Select all true statements about monitors (*choose all that apply*):

A: ☐ A monitor consists of a lock and zero or more conditional variables.

B: ☐ The `cond_wait()` function (internally) releases the monitor lock and puts the thread to sleep until the `cond_signal()` or `cond_broadcast()` function has been called.

C: ☐ In a system with Mesa semantics, there is no guarantee that a signaled condition will still be true when the signaled (and awoken) thread gets around to checking the condition.

D: ☐ Monitors cannot be implemented with only semaphores.

E: ☐ None of the above.

Problem 2c[2pts]: Select all of the following that are true of the stack of a PintOS user program (*choose all that apply*):

A: ☐ When pushing argv to the user stack, the values of the arguments are pushed before the pointers to those values.

B: ☐ During context switching, all FPU and thread registers are saved to the user stack.

C: ☐ The user stack pointer starts at the virtual address PHYS_BASE and is decremented to make space for arguments, local variables, etc.

D: ☐ The user stack and the kernel stack for a given process are located on the same page in physical memory.

E: ☐ A user program can successfully execute the assembly instruction mov 0xc0000008, esp (ignore any issues that may occur in following instructions).

Problem 2d[2pts]: In Pintos, every user-level thread has both a user-level stack and a kernel-level stack. The kernel stack is sometimes called a "kernel thread" because it manipulated by the scheduler when multiplexing the CPU. What is true about this arrangement (*choose all that apply*):

A: ☐ The physical memory for the kernel stack must be at a lower address than the physical memory for the user stack so that the stacks can be considered together (treating a system call like a procedure call into the kernel).

B: ☐ The kernel gains safety because it does not have to rely on the correctness of the user's stack pointer register or validity of user's stack memory for correct behavior.

C: ☐ While the thread is executing within the kernel, it has access to an arbitrarily large stack, allowing deeply recursive handling of the scheduling algorithm.

D: ☐ Threads which run exclusively within the kernel (and have no associated user-level stack) can be scheduled by the same scheduler as regular (user) threads.

E: ☐ When the user-thread makes a system call, the thread can be blocked at any time (and at any call depth within the kernel) by saving current state on the kernel stack and/or TCB, putting the kernel stack on a wait queue, and restoring state from another kernel stack linked into the ready queue.

Problem 2e[2pts]: What are some things that can cause a transfer from user mode to kernel mode? (*choose all that apply)*:

A: ☐ User code divides by zero.

B: ☐ The user executes **`fib(0x20000000)`** with a recursive implementation of **`fib().`** Here, `fib(n)` computes the n^{th} Fibonacci number.

C: ☐ A packet is received from the network

D: ☐ The application uses malloc() to allocate memory from the heap.

E: ☐ The timer goes off.

Problem 2f[2pts]: Kernel mode differs from User mode in the following ways (*choose all that apply)*:

A: ☐ The CPL (current processor level) is 0 in kernel mode and 3 in user mode for Intel processors

B: ☐ In Kernel mode, additional instructions become available, such as those that modify page table registers and those that enable/disable interrupts.

C: ☐ Specialized instructions for security-related operations (such as for cryptographic signatures) are only available from Kernel mode.

D: ☐ Control for I/O devices (such as the timer, or disk controllers) are only available from kernel mode

E: ☐ Pages marked as Kernel-mode in the PTEs are only available in kernel mode.

Problem 2g[2pts]: Consider the following *pseudocode* implementation of a `lock_acquire()`.

```
lock_acquire() {
    interrupt_disable();
    if (value == BUSY) {
         put thread on wait queue;
         Go to sleep();
    } else {
         value = BUSY;
    }
   interrupt_enable();
}
```

Which of the following are TRUE? Assume we are running on a uniprocessor/single-core machine. (*choose all that apply):*

A: ☐ It is possible to build a system with two *independent* (separate) locks even though there is only one global interrupt_disable() bit.

B: ☐ The wait queue being referenced here must keep threads in FIFO order in order to provide a correct locking implementation.

C: ☐ For this implementation to be correct, `sleep()` should trigger the scheduler which will reenable interrupts as part of running the next thread.

D: ☐ It is possible for a lock built this way to be exploited by user code.

E: ☐ None of the above.

Problem 2h[2pts]: Which of the following are true about semaphores (*choose all that apply):*

A: ☐ Semaphores can be initialized to any 32-bit values in the range -2^{31} to $2^{31}-1$

B: ☐ `Semaphore.V()` increments the value of the semaphore and wakes a sleeping thread if the value of the semaphore is > 0

C: ☐ Semaphores can be implemented with Monitors (using one condition variable per Semaphore).

D: ☐ The interface for `Semaphore.P()` is specified in a way that prevents its implementation from busywaiting, even for a brief period of time.

E: ☐ The pure semaphore interface does not allow querying for the current value of the semaphore.

Problem 3: Cat-Dog Lock [16pts]

A cat-dog lock is a sort of generalized reader-writer lock: in a reader-writer lock there can be any number of readers or a single writer (but not both readers and writers at the same time), while in a cat-dog lock there can be any number of cats or any number of dogs (but not both a cat and a dog at the same time). Assume that we are going to implement this lock at user level utilizing pthread monitors (i.e pthread mutexes and condition variables). Note that the assumption here is that we will put threads to sleep when they attempt to acquire the lock as a Cat when it is already acquired by one or more Dogs and vice-versa.

You must implement the behavior using condition variable(s). Assume that the system provides MESA semantics. Points will be deducted for any spin-waiting behavior.

Some snippets from POSIX Thread manual pages showing function signatures are shown at end of this exam. They may or may not be useful.

Our first take at this lock is going to utilize the following structure and enumeration type:

```
/* The basic structure of a cat-dog lock */
struct cdlock {
    pthread_mutex_t lock;
    pthread_cond_t wait_var;

    // Simple state variable
    int state;      // (<0) => CATS, 0 => FREE, (>0) => DOGS
};

/* Enumeration to indicate type of requested lock */
enum cdlock_type { CDLOCK_FREE, CDLOCK_CAT, CDLOCK_DOG };

/* interface functions: return 0 on success, error code on failure */
int cdlock_init(struct cdlock *lock);
int cdlock_lock(struct cdlock *lock, enum cdlock_type type);
int cdlock_unlock(struct cdlock *lock);
```

Note that the lock requestor specifies the type of lock that they want at the time that they make the request:

```
/* Request a Cat lock */
if (cdlock_lock(mylock, CDLOCK_CAT) {
    printf("Lock request failed!");
    exit(1);
}
/* . . . Code using lock . . . */

/* Release your lock */
cdlock_unlock(mylock);
```

Problem 3a[3pts]: Complete the following sketch for the initialization function. Note that initialization should return zero on success and a non-zero error code on failure (e.g. return the failure code, if you encounter one, from the various synchronization functions). *Hint: the state of the lock is more than just "acquired" or "free".* This must be done in five (5) or less lines (can be done in 4).

```
/* Initialize the CD lock.
 * Args: pointer to a cdlock
 * Returns: 0 (success)
 *          non-zero (errno code from synchronization functions)
 */
int cdlock_init(struct cdlock *lock) {
    int errcode; // For error values









}
```

Problem 3b[4pts]: Complete the following sketch for the lock function. Think carefully about the state of the lock; when you should wait, when you can grab the lock. Also think about what is required to handle unlock later. Return a failure code from underlying pthread functions if they occur. *Hint: accumulate count of compatible types.* This must be done in nine (9) or less lines (can be done in 7).

```
/* Grab a CD lock.
 * Args: (pointer to a cdlock, enum lock type)
 * Returns: 0 (lock acquired)
 *          non-zero (errno code from synchronization functions)
 */
int cdlock_lock(struct cdlock *lock, enum cdlock_type type) {
    int errcode; // For error values

    // Get direction for accumulating lock holders
    int dir = (type == CDLOCK_CAT)?-1:1;















}
```

Problem 3c[4pts]: Complete the following sketch for the unlock function. Be sure to return an error code from the underlying synchronization functions if they occur. This must be done in nine (9) or less lines (can be done in 7).

```
/* Release a CD lock.
 * Args: pointer to a cdlock
 * Returns: 0 (lock acquired)
 *          non-zero (errno code from synchronization functions)
 */
int cdlock_unlock(struct cdlock *lock) {
    int errcode; // For error values











}
```

Problem 3d[2pts]: Consider a group of "nearly" simultaneous arrivals (i.e. they arrive in a period much quicker than the time for any one thread that has successfully acquired the Cdlock to get around to performing `cdlock_unlock()`). Assume that they enter the `cdlock_lock()` routine in this order:

C_1, D_1, C_2, D_2, C_3, C_4, C_5, D_3, D_4, C_6, C_7, C_8, D_5

How will they be grouped? (Place braces, namely "{}" around requests that will hold the lock simultaneously). This simple lock implementation (with a single state variable) is subject to starvation. Explain. *Note that we are asking for 2 separate things in this problem—the grouping and the explanation about starvation.*

Problem 3e[3pts]: Suppose that we want to enforce fairness, such that Cat and Dog requests are divided into phases based on arrival time into the `cdlock_lock()` routine. Thus, for instance, an arrival stream of Cats and Dogs such as this:

$$C_1, C_2, D_1, D_2, D_3, D_4, C_3, D_5, C_4, C_5$$

will get granted in groups such as this:

$$\{C_1, C_2\}, \{D_1, D_2, D_3, D_4\}, \{C_3\}, \{D_5\}, \{C_4, C_5\}$$

To do this, we will enhance our cdlock structure like this:

```
/* The basic structure of a cat-dog lock */
#define MAX_SIMU_GROUPS 20
struct cdlock {
    pthread_mutex_t lock;
    pthread_cond_t wait_var;

    int head,tail;
    int state[MAX_SIMU_GROUPS]; // (<0) => CATS, 0 => FREE, (>0) => DOGS
};
```

In 3 sentences or less, explain how this change will allow the CD lock to enforce fairness. *Hint: Explain (at a high level, without code) how this would change* `cdLock_Lock():`.

Problem 4: Short Answer Potpourri [20 pts]

For the following questions, provide a concise answer of NO MORE THAN 2 SENTENCES per sub-question (or per question mark).

Problem 4a[3pts]: How does the OS prevent malicious user programs from executing arbitrary code in kernel mode or from accessing other programs' memory? Name 3 mechanisms (limit 1 sentence per mechanism):

Problem 4b[4pts]: Explain the key difference between the low-level and high-level file APIs in C as discussed in lecture. Give an example in which the low-level API would be faster than the high-level API and give an example in which the high-level API would be faster than the low-level API (limit 1 sentence per example):

Problem 4c[3pts]: What are some of the hardware enforced differences between kernel mode and user mode? Name three:

For problems **4d, 4e,** and **4f**, consider the following sketch of the code for a network server. This server communicates with clients running on other machines using the socket abstraction. To handle requests from clients, the server follows four steps:

```
       server {
(1):      int lsock = socket(…)
(2):      bind(lsock,…)
(3):      listen(lsock,…)
          while (true) {
(4):         int conn = accept(lsock,…);
             If (conn < 0) break;
(5):         handle_request(conn);
          }
       }
```

Problem 4d[2pts]: The `accept()` system call in step (4) returns an integer. What does this integer represent (be explicit) and why does the code include an infinite while loop to keep executing `accept()` over and over?

Problem 4e[2pts]: At step (5), above, the server can either directly handle the incoming request, or it can create a new thread or process to handle the connection. Assuming that the server has only a single core with no hyperthreading, explain why it might make sense to create concurrency (i.e. either a thread or a process):

Problem 4f[2pts]: Assuming the concurrency option of (**Problem 4e**) has been chosen, provide one advantage for why the server should create a new **process** to handle the request and one advantage for why the server should create a new **thread** to handle the new request instead. [Single sentence per advantage]:

Problem 4g[4pts]: In the following code, add global variables/locks and/or synchronization functions so that it is guaranteed to print:

B
A
C

You do not need to use every line, but you may only put one statement per line (no comma expressions) and only declare global variables or synchronization functions. Assume that any necessary include files have been included.

```
struct semaphore sema1;
struct semaphore sema2;

void* A(void* aux) {

    ________________________________________
    printf("A\n");

    ________________________________________
    return(NULL);
}

void* B(void* aux) {

    ________________________________________
    printf("B\n");

    ________________________________________
    return(NULL);
}

void* C(void* aux) {

    ________________________________________
    printf("C\n");

    ________________________________________
    return(NULL);
}

int main() {

    ________________________________________

    ________________________________________
    pthread_t tid;
    pthread_create(&tid, NULL, A, NULL);
    pthread_create(&tid, NULL, B, NULL);
    pthread_create(&tid, NULL, C, NULL);
    pthread_exit(NULL);
}
```

Problem 5: Futex Implementation[28pts]

In lecture, we introduced the Linux `futex()` system call, short for "fast userspace mutex". In this problem, you will implement a portion of a simplified version of the `futex()`.

The function signature for our simplified futex syscall is as follows:

```
int futex(int *uaddr, int futex_op, int val) {
  /* uaddr: a pointer to an int in user space.
     futex_op: a code indicating which futex operation to perform
     (e.g. FUTEX_WAIT or FUTEX_WAKE).
     val: a value that is compared against *uaddr.
  */
}
```

Normally, this system call is buried in `libc` and used to implement various locking primitives such as locks, semaphores, and monitors. While the full version of futex() defines a number of futex_op values, we will focus on only two of them:

1. For `FUTEX_WAIT`, if `*uaddr == val`, the calling thread will go to sleep. The loading of the value stored at uaddr, the comparison of that value with val, and the blocking of the thread will happen atomically. After waking up, the calling thread will **return 0**. If, instead, `*uaddr != val`, the function will return immediately with a **return value of -1**.
2. For `FUTEX_WAKE`, this function will wake up to val waiting threads. You can assume in this problem that val will always be <= the actual number of waiting threads.

Consequently, futex() provides a unique sleep queue in the kernel associated with each uaddr.

Problem 5a[2pts]: In lecture, we showed one possible implementation of a user-level lock using the `futex()` system call as a primitive. It looked like this:

<pre>acquire(int *thelock) {
   while (test&set(thelock)) {
      futex(thelock, FUTEX_WAIT, 1);
   }
}</pre>
	<pre>release(int *thelock) {
	   *thelock = 0;
	   futex(thelock, FUTEX_WAKE, 1);
	}</pre>

In two sentences or less, explain why this implementation does no *busy waiting*, despite the presence of a `while()` loop in `acquire()`:

Problem 5b[2pts]: In two sentences or less, explain (1) why this implementation *does not* allow an *uncontested* lock (one primarily used by one thread at a time with occasional collisions) to be acquired and released quickly entirely at user level (without entering the kernel) and (2) how we suggested fixing this problem in lecture:

Problem 5c[7pts]: Since `futex()` asssociates a sleep queue with each unique integer address (first argument), we can build semaphores very simply: a semaphore is literally a pointer to a memory location holding an integer. Complete the following implementations for `sema_down()` and `sema_up()` using `futex()` and compare-and-swap (`CAS`). Recall the following behavior for `CAS`:

```
/* pseudocode for behavior of the atomic compare and swap instruction */
bool CAS(int *addr, int expr1, int expr2) {
   if (*addr == expr1) {
      *addr = expr2;
      return true;
   } else {
      return false;
   }
}
```

Only one expression per blank will be accepted (no semicolons or comma expressions). Keep in mind that an expression of the form `(a = b)` does two things in "C". (1) It assigns the value of `b` to the variable `a` and (2) it returns the value of `b`. Thus, `(a = b) == c` is a valid expression.

Hint: don't forget that the implementation must work (be atomic) with multiple simultaneous assertions of `sema_down()` and `sema_up()`. Further note that most interesting transitions happen around 0.

```
sema_down(int *the_sema) {
   int curvalue;
   do {

      while (__________________________________________________) {


         futex(the_sema, ___________________, _____________________);
      }

   } while (!CAS(the_sema, ___________________, ____________________));
}

sema_up(int *the_sema) {
   int curvalue;
   do {

      ________________________________________;


   } while (!CAS(the_sema, ___________________, ____________________));


   if (___________________________________________) {


      futex(the_sema, ___________________, _____________________);
   }
}
```

Problem 5d[8pts]: Inside the kernel, the `futex()` implementation must provide a wait queue for each unique address (`uaddr`) passed to `futex()`. We represent each wait queue as follows:

```
typedef struct futex {
   struct condition cond;  /* queue of waiting threads */
   struct lock lock;
   int *uaddr;
   struct list_elem elem;
} futex_t;
```

Fill in the blanks to produce the helper function that gets a futex object associated with a given uaddr. As before, only one expression per blank, no semicolons or comma expressions:

```
struct list futex_list;       // Global list used by the kernel
struct lock futex_list_lock;
/* Called when kernel boots up. */
void init() {
   list_init(&futex_list);
   lock_init(&futex_list_lock);
}

/* Our target helper function */
futex_t *get_futex(int* uaddr) {
   struct list_elem* e;
   futex_t* f;
   bool found = false;

   ______________________________________________________________;

   for (e = ___________________________; ______________________________;
        e = list_next(e)) {

      __________________________________________________________;

      if (______________________________________________________) {
         found = true;
         break;
      }
   }
   if (!found) {
      f = (futex_t*)malloc(sizeof(futex_t));

      __________________________________________________________;

      __________________________________________________________;

      __________________________________________________________;
      list_push_front(&futex_list, &f->elem);
   }
   ______________________________________________________________;
   return f;
}
```

Problem 5e[9pts]: Finally, implement the futex() system call. Don't forget that it is called from user space an must be properly validated. As before, provide only one expression per blank, no semicolons or comma expressions:

```
// Checks to see if pointer is within the memory address space.
// Terminates the current program if not; no need to return
void validate_pointer(char* pointer);

// This function called by syscall handler after user calls futex()
int syscall_futex(int* uaddr, int futex_op, int val) {

    ____________________________________________________________;

    futex_t* f = ___________________________________________;
    if (futex_op == FUTEX_WAIT) {
        enum intr_level old_level;
        old_level = intr_disable();

        if (______________________________________________________) {

            ____________________________________________________;

            ____________________________________________________;

            ____________________________________________________;
            intr_set_level(old_level);
            return 0;
        } else {

            ____________________________________________________;
            return -1;
        }
    } else if (futex_op == FUTEX_WAKE) {
        int num_woken = 0;

        ________________________________________________________;
        while (num_woken < val) {

            ____________________________________________________;
            num_woken++;
        }

        ________________________________________________________;
        return 0;
    } else {
        // Remaining futex_ops elided for this problem
    }
}
```

Function Reference Sheet

Feel free to remove this sheet during the exam

```
/* Process */
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[]);

/* pthreads */
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);
void pthread_exit(void *retval);

/* pthread Semaphore interface */
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem); /* The p() or down() operation */
int sem_post(sem_t *sem); /* The v() or up() operation */

/* pthread Lock/mutex operations */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* pthread Condition Variable */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
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);

/* Pintos locks */
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

/* Pintos semaphore interface */
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);

/* Pintos condition variables */
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_broadcase(struct condition *cond, struct lock *lock);
```

```
/* Pintos Readers/Writers Locks */
void rw_lock_init(struct rw_lock*);
void rw_lock_acquire(struct rw_lock*, bool reader);
void rw_lock_release(struct rw_lock*, bool reader);

/* Pintos List */
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 list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);

/* Strings */
char *strcpy(char *dest, char *src);
char *strdup(char *src);

/* Interrupt enable/disable */
enum intr_level {};
enum intr_level intr_get_level(void)
enum intr_level intr_set_level(enum intr_level level)
enum intr_level intr_enable(void)
enum intr_level intr_disable(void)

/* High-Level IO */
FILE *fopen(const char *pathname, const char *mode);
int fclose(FILE *stream);
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);
fprintf(FILE * restrict stream, const char * restrict format, ...);

/* Low-Level IO */
int open(const char *pathname, int flags);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);

/* Socket */
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

[Scratch Page: Do Not Put Answers Here]

[Scratch Page: Do Not Put Answers Here]