CS 162 Fall 2022 Solutions

MIDTERM 2

INSTRUCTIONS

Please do not open this exam until instructed to do so.

Do not discuss exam questions for at least 24 hours after the exam ends, as some students may be taking the exam at a different time.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

Preliminaries

This is a proctored, closed-book exam. You are allowed 2 pages of notes (both sides). You may not use a calculator. You have 70 minutes to complete as much of the exam as possible. Make sure to read all the questions first, as some are substantially more time-consuming.

If there is something about the questions you believe is open to interpretation, please ask us about it.

We will overlook minor syntax errors when grading coding questions. **There is a reference sheet at the end of the exam that you may find helpful.**

(a) Name

(b) Student ID

(c) Please read the following honor code: "I understand this is a closed-book exam. I promise the answers I give on this exam are my own. I understand that I am allowed to use two 8.5x11, double-sided, handwritten cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or the internet in constructing my answers."

Write your full name below to acknowledge that you've read and agreed to this statement.

1. (10.0 points) True/False

Please explain your answer in **TWO SENTENCES OR LESS**. Longer explanations may get no credit. Answers without an explanation **GET NO CREDIT**.

(a) (2.0 points)

i. If the OS scheduler receives 3 threads at tick 0, each with a runtime of 2 ticks, then the average thread completion time will be greater when using a FCFS (first-come first-served) scheduler than when using a RR (round-robin) scheduler with quantum = 1 tick. (Note: The completion time for a thread is the number of ticks between when the thread requests to be run and when the thread finishes running.)

○ True.

● False.

ii. Explain.

False
FCFS avg time = (2 + 4 + ... + 2n) / n = (2 * (1 + ... + n)) / n = (2 * n * (n + 1) / 2) / n = n + 1 = 4 (for 3 threads)
RR avg time = ((n + 1) + (n + 2) + ... + (n + n)) / n = (n^2 + (1 + ... + n)) / n = n + (n + 1) / 2 = (3/2)n + 1/2 = 5 (for 3 threads)

(b) (2.0 points)

i. Circular wait implies deadlock.

○ True.

● False.

ii. Explain.

False
Also require the OS to have mutual exclusion, no preemption, and hold and wait.

(c) (2.0 points)

i. MLFQs are typically implemented such that IO-bound jobs are clustered in low priority queues while CPU-bound jobs are clustered in high priority queues.

○ True.

● False.

ii. Explain.

False
Opposite is true (IO-bound jobs are placed in high priority queues since they yield their time to the CPU).

(d) (2.0 points)

i. Suppose physical memory can store N pages and our demand paging system uses a LRU replacement policy. Then if a process's resident set contains N + 1 pages accessed sequentially forever (i.e., $P_1, P_2, ..., P_N, P_{N+1}, P_1, ...$), the hit rate for the demand paging system will tend to 1 / (N + 1).

○ True.

● False.

ii. Explain.

False
The hit rate will be 0, since each access will always evict the next accessed page (e.g., accessing P_{N+1} evicts P_1, P_1 evicts P_2, etc.)

(e) (2.0 points)

i. Suppose a system has only 2 threads: a high priority thread (H) and a low priority thread (L). If H is waiting on a lock held by L, then the scheduler needs to implement priority donation to prevent starvation of H.

○ True.

● False.

ii. Explain.

False
Starvation only happens if there is a middle priority thread preventing low priority thread from being run

2. (10.0 points) Multiple Choice

Choose ALL that apply.

(a) (2.0 points)

i. Select all true statements about basic scheduling protocols.

- [x] If an OS uses a MLFQ (multilevel feedback queue) scheduler, a thread may use certain syscalls to manipulate which queue it is placed on.
- [ ] The convoy effect occurs in SJF (shortest job first) because longer threads can get stuck behind a build-up of shorter threads.
- [x] A lottery scheduler can approximate priority scheduling.
- [x] If all jobs in a queue are the same length, FCFS is an optimal scheduling protocol among non-preemptive schedulers.
- [ ] None of the above

A: Thread can use IO syscalls to yield CPU and get pushed up

B: The convoy effect is the reverse (shorter threads get stuck behind longer threads)

C: Giving more tickets to high priority jobs increases the probability that the scheduler runs high priority jobs

D: All jobs have the same length, so all non-preemptive schedulers will have the same average completion time, wait time, response time, etc.

(b) (2.0 points)

i. Select all true statements about paging.

- [x] Systems that use paging can allow for two processes to access shared physical memory.
- [ ] When using paging, contiguous virtual addresses must map to contiguous physical addresses.
- [ ] If a virtual address has 12 bits and we use a page table with 4 entries, the offset in the virtual address has 8 bits.
- [ ] If a virtual address space has 2^M pages, and m $< 2^M$ pages have been mapped, a single level page table will use O(m) memory.
- [ ] None of the above

A: Both process's page tables can contain a mapping to the same physical page

B: If two successive virtual addresses cross a page boundary, both can be located in non-successive pages in physical memory

C: The VPN uses log2(4) = 2 bits, so the offset uses 12 - 2 = 10 bits.

D: A single level page table must contain entries for the entire virtual address space (using O(2^M) memory)

(c) (2.0 points)

i. Select all true statements about caching.

- ☐ Translating an address with a two-level page table requires at least two TLB lookups.
- ☐ Demand paging is the use of a hardware cache to avoid needing to read pages from main memory.
- ■ We can include PIDs in TLB entries to make them stay valid after context switching.
- ☐ A miss in the TLB is called a page fault.
- ☐ None of the above

A: A TLB caches the VPN -> PPN translation itself (only one lookup needed)

B: Demand paging uses main memory to avoid reading pages from disk

C: Adding the PID prevents the new process from accidentally reading an incorrect translation leftover from the previous process

D: A page fault is when a process tries to access a page located in disk

(d) (2.0 points)

i. Select all true statements about demand paging.

- ■ The clock algorithm is an approximation of MIN.
- ☐ The more pages we allocate for the second chance list, the more the algorithm approximates FIFO.
- ■ The EAT (expected access time) may stay the same if page hit rate and page hit time are both increased.
- ☐ The LRU replacement policy is susceptible to Belady's anomaly.
- ☐ None of the above

A: The clock algorithm approximates LRU, which approximates MIN

B: The more pages allocated for the second chance list, the more the algorithm approximates LRU

C: EAT = hit time + (1 - hit rate) * miss penalty (increasing both can result in the same EAT)

D: LRU has the stack property (so it is immune to Belady's anomaly)

(e) (2.0 points)

i. Select all true statements about threads in PintOS.

- [x] An exited thread will free its contents only after a context switch from that thread.
- [ ] When a kernel thread stack overflows, it will allocate more space to dynamically fit extra data.
- [x] Semaphores contain an internal queue of threads waiting on the semaphore.
- [ ] A user thread created with the `pthread_create` syscall will share the same kernel thread as its process's main thread.
- [ ] None of the above.

A: The contents of the thread are free in `thread_switch_tail`

B: The kernel stack is limited to one page (overflow will overwrite the TCB)

C: Threads added to `sema.waiters` if they block in `sema_down`

D: `pthread_create` creates a new kernel thread to manage the new user thread in a 1-1 correspondence

3. (18.0 points) Short Answer

(a) (4.0 pt) Given:

- threads T1, T2, T3
- resources R1, R2, R3
- the current allocations of each resource to each thread
- the current requests for each resource from each thread

Provide a valid ordering of threads that will allow each request to be completed. Assume that if a request can be completed, the thread relinquishes its currently allocated resources. If not all requests can be completed, provide an ordering of threads that maximizes the number of requests that can be completed AND write "DEADLOCK".

Available	R1	R2	R3
	0	2	1

Allocations	R1	R2	R3
T1	1	2	2
T2	5	0	1
T3	10	3	6

Requests	R1	R2	R3
T1	3	0	2
T2	0	1	1
T3	1	3	3

T2 T1 T3
At each step, the thread's resources must be less than or equal to the amount available on the system. After the thread is run, its allocation is freed and added to the amount available on the system.
Run T2:
Available: R1: 5, R2: 2, R3: 2
Run T1:
Available: R1: 6, R2: 4, R3: 3
Run T3:
Available: R1: 16, R2: 7, R3: 9

(b) (4.0 pt) Assume that you have correctly implemented a strict priority scheduler that includes priority donation for locks. Prove that, if Thread A tries to acquire a lock held by Thread B, then Thread B's effective priority must be less than or equal to Thread A's effective priority. If this is not true, provide a counterexample. (Maximum 3 sentences.)

Thread B can either be on the ready queue or on a waiting queue (i.e. it is blocked on acquiring some lock). Assume for the sake of contradiction that Thread B has higher effective priority than Thread A.
Case 1: Thread B is on the ready queue. By definition of strict priority scheduling, B should be running instead of A. This contradicts the fact that A is running.
Case 2: Thread B is blocked on acquiring some lock. This means that there is a "dependency chain" of the form [Thread B] -> [Thread C] -> ... [Ready Thread], where Thread B donates priority to Thread C, Thread C donates priority to the next thread in the chain, and so on, all the way until the "Ready Thread", which is a thread on the ready queue. Note that "Ready Thread" need not be the holder of the lock that B wishes to acquire; it is certainly possible that the holder of the lock that B wants to acquire is itself blocked on another lock. However, if we traverse the full dependency chain, we must arrive at a thread that is not blocked on any other thread. By priority donation, the "Ready Thread" will have effective priority equal to that of B, which is greater than the effective priority of A. By strict priority scheduling, the "Ready Thread" must be running instead of A. Contradiction.

(c) (10.0 points)

For this question, suppose we have a single-level page table, 22-bit virtual addresses, 18-bit physical addresses and 4 KiB pages. Show your work for all subparts. Assume the 18-bit physical address is sufficient to address each byte of physical memory.

i. (1.0 pt) How many physical pages does this sytem have?

$2^{18}/2^{12} = 2^6$ **physical pages.**

ii. (1.0 pt) How many entries does the page table have?

$2^{22}/2^{12} = 2^{10}$ **virtual pages** $\Rightarrow$ 2^{10} **page table entries.**

iii. (2.0 pt) Suppose this system has a TLB with a hit rate of 80% and an access time of 20ns and a memory cache with a hit rate of 10% and an access time of 60ns. Main memory accesses take 200ns, and cache and TLB lookups do not overlap.

What is the average time it takes to access memory at a given virtual address? Show your work. (Hint: Remember that the page table is stored in memory as well.)

The average cache lookup time is C = 60ns + 0.9 * 200ns = 240ns. The AMAT is 20ns + 0.2 * C + C = 308ns.

iv. (6.0 pt) Suppose that we keep the same number of virtual pages, but use a two-level page table (assume that tables at both levels each have the same number of entries). Ignoring caching, describe the three memory accesses that would need to be made to access the virtual address `0x2ff57a`.

To save you some work, the binary representation of `0x2ff57a` is `0b10 1111 1111 0101 0111 1010`.

NOTE: You do not need to provide the exact addresses that are accessed, but discuss conceptually how you would determine these addresses based on the information you have and what additional information you would need to use. Be specific about how you would use the provided virtual address.

The first step would be to split up the given address into virtual page number and offset bits. Since pages are 2^12 bytes, we need 12 offset bits. The remaining 10 bits of the virtual address are split equal between the two levels of the page table (5 bits for VPN 1, 5 bits for VPN 2).
Memory access 1: We would need to look up the physical page number for the virtual page number `0b10111` (first 5 bits) in the top level page table, whose location is stored at the PTBR. This will tell us the physical address of the second level page table, A2.
Memory access 2: We would then need to look up the physical page number for the virtual page number `0b11111` (second 5 bits) in the second level page table, whose location is at A2. This would give us the physical page that our virtual page corresponds to.
Memory access 3: We now use the offset (`0b010101111010`) to index into the physical page found earlier and read the desired byte in memory.

4. (6.0 points) Scheduling Protocols

Suppose tasks A, B, and C arrive at the OS scheduler at the same time. **Show the scheduling order of tasks for FCFS, Stride, and Lottery schedulers for 8 time slices. Break ties alphabetically. Assume the runtime for each task is infinite.** Provide each ordering as a comma-separated list of 8 task names, one for each time slice; for example, the ordering for a round robin scheduler with quantum = 1 time slice would be `A, B, C, A, B, C, A, B`.

For the stride scheduler, use the following strides for each task:

Task A	Task B	Task C
50	40	25

For the lottery scheduler, the system assigns the following lottery tickets to each task:

Task A	Task B	Task C
[0, 1, 2]	[3, 4, 5, 6]	[7]

Assume the system uses a "random" number generator to select lottery tickets at each time slice, which you can assume outputs the following values (mod 8): **[0, 2, 6, 3, 7, 5, 5, 2]**. Specifically, at time slice 0 it outputs 0, at time slice 1 it outputs 2, at time slice 2 it outputs 6, etc.

(a) (2.0 pt) FCFS

`A, A, A, A, A, A, A, A`

(b) (2.0 pt) Stride

`A, B, C, C, B, A, C, C`

(c) (2.0 pt) Lottery

`A, A, B, B, C, B, B, A`

5. (10.0 points) Walter's Algorithm

Walter White and Jesse are manufacturing rock candy for the highly competitive confections market! Each rock candy color needs different amounts of two reusable resources, flasks and ovens, over the course of its recipe.

Unfortunately, Jesse already started cooking multiple colors simultaneously, so each color now has an initial allocation of each resource that cannot be deallocated, stored in the `struct color` below.

```
struct color {
    int max_flasks;    // max flasks needed over entire recipe
    int max_ovens;     // max ovens needed over entire recipe
    int init_flasks;   // initial allocation of flasks
    int init_ovens;    // initial allocation of ovens
    bool completed;    // color has completed cooking (initialized to false)
}
```

(a) (5.0 points) Part 1: Coding Walter's Algorithm

Mr. White needs you to see if it's possible to fix Jesse's mistake. Complete the function below that returns a bool for whether it is possible to run all color recipes to completion. Assume that you may only cook one color at a time, and that a color's ENTIRE recipe must be completed without interrupts.

```
NUM_COLORS; // total number of colors to manufacture

/**
 * returns true if there is a way to run all color recipes.
 * COLOR_INFO_ARR: array of NUM_COLORS initial struct color* elements
 * INIT_TOTAL_FLASKS: number of initially unallocated flasks
 * INIT_TOTAL_OVENS: number of initially unallocated ovens
**/
bool walters_algorithm(struct color** color_info_arr,
                       int init_total_flasks,
                       int init_total_ovens) {
    int num_completed = 0;
    int available_flasks = init_total_flasks;
    int available_ovens = init_total_ovens;
    while (num_completed < NUM_COLORS) {
        bool updated = false;
        struct color* curr;
        for (int i = 0; i < NUM_COLORS; i++) {
            curr = color_info_arr[i];
            if (!curr->completed) {
                if (______[A]______ <= available_flasks &&
                    ______[B]______ <= available_ovens) {
                    updated = true;
                    num_completed++;
                    curr->completed = true;
                    available_flasks += ______[C]______;
                    available_ovens += ______[D]______;
                }
            }
        }
        if (______[E]______) {
            return false;
        }
    }
    return true;
}
```

i. **(1.0 pt)** [A]

```
curr->max_flasks - curr->init_flasks
```

ii. **(1.0 pt)** [B]

```
curr->max_ovens - curr->init_ovens
```

iii. **(1.0 pt)** [C]

```
curr->init_flasks
```

iv. **(1.0 pt)** [D]

```
curr->init_ovens
```

v. **(1.0 pt)** [E]

```
!updated
```

(b) (5.0 points) Part 2: Breaking Walter's Algorithm

Mr. White makes an important observation! Each color's recipe has two separate stages: Stage 1 (stirring) only requires flasks, and Stage 2 (baking) only requires ovens. So, Mr. White relaxes his requirement that we must run each color to completion before starting another color. Now, each color can acquire its maximum flasks needed (Stage 1), relinquish its flasks and allow another color to cook, and then acquire its maximum ovens needed (Stage 2) and finish cooking.

Given this relaxed assumption, using exactly 2 colors with the maximum resources needed given below, **provide an initial allocation of flasks and ovens that would result in `walters_algorithm` returning an incorrect value.**

```
struct color c1;
c1.init_flasks = ______[A]______;
c1.init_ovens = ______[B]______;
c1.max_flasks = 5;
c1.max_oven = 5;

struct color c2;
c2.init_flasks = ______[C]______;
c2.init_ovens = ______[D]______;
c2.max_flasks = 5;
c2.max_oven = 5;

init_total_flasks = 0;
init_total_ovens = 0;
```

i. (5.0 pt) [A]; [B]; [C]; [D]; (input solutions to each blank in order and separated by semicolons)

```
// The colors can be swapped but must satisfy
// the following properties to receive full credit.
struct color c1;
c1.init_flasks = 5; // any value >= 5 works
c1.init_ovens = 0; // any value < 5 works

struct color c2;
c2.init_flasks = 0; // any value < 5 works
c2.init_ovens = 5; // any value >= 5 works
```

6. (6.0 points) Deadlock Detection for Pintos Locks

Given the following implementations of `lock_acquire` and `lock_release`, fill in the blanks to complete `will_deadlock`, which checks if acquiring lock `to_acquire` would lead to a circular wait. The relevant fields of `struct thread` and `struct lock` have been included below.

```
struct thread {
    ...
    struct list locks_held;     /* Locks held by thread. */
    struct lock* lock_waiting;  /* Lock that thread is waiting on (NULL if none). */
    ...
}

struct lock {
    bool held;
    struct thread* holder;      /* Thread holding lock (for debugging). */
    struct list waiters;        /* Threads that are in the wait queue */
    struct list_elem elem;      /* For storing in TCB list of locks held */
};

void lock_acquire(struct lock* lock) {
    enum intr_level old_level = intr_disable();
    struct thread* t = thread_current();
    t->lock_waiting = lock;
    while (lock->held) {
        struct thread* holder = lock->holder;
        list_push_back(&lock->waiters, &t->elem);
        thread_block();
    }
    lock->held = true;
    lock->holder = t;
    list_push_back(&t->locks_held, &lock->elem);
    intr_set_level(old_level);
}

void lock_release(struct lock* to_acquire) {
    enum intr_level old_level = intr_disable();
    struct thread* t = thread_current();
    lock->holder = NULL;
    lock->held = false;
    list_remove(&lock->elem);
    if (!list_empty(&lock->waiters)) {
        struct thread* next = list_entry(list_pop_front(&lock->waiters), struct thread, elem);
        thread_unblock(next);
    }
    intr_set_level(old_level);
}


bool will_deadlock(struct lock* to_acquire) {
    enum intr_level old_level = intr_disable();
    struct thread* t = thread_current();
    struct lock* curr = to_acquire;

    // Recursively check that the lock being acquired
    // isn't held by the current thread.
```

```
    while (curr != NULL && _____[A]_____) {
        struct list_elem *e;
        for (e = list_begin(&t->locks_held);
                e != list_end(&t->locks_held); e = list_next(e)) {
            struct lock* held_lock = list_entry(e, struct lock, elem);
            if (_____[B]_____) {
                _____[C]_____;
                return true;
            }
        }
        curr = _____[D]_____;
    }
    _____[E]_____;
    return false;
}
```

(a) (1.0 pt) [A]

```
curr->holder != NULL
```

(b) (1.0 pt) [B]

```
held_lock == curr
```

(c) (1.0 pt) [C]

```
intr_set_level(old_level)
```

(d) (1.0 pt) [D]

```
curr->holder->lock_waiting
```

(e) (1.0 pt) [E]

```
intr_set_level(old_level)
```

7. (15.0 points) TLB Implementation

In this problem, you will be implementing a fully-associative TLB with an LRU eviction policy.

NOTE: You should use AT MOST the number of lines specified for each section. Extra lines will not be graded.

```
#define NUM_ENTRIES 128

typedef struct tlb {
    tlb_entry_t tlb_entries[NUM_ENTRIES];
} tlb_t;

typedef struct tlb_entry {
    int vpn;
    int ppn;
    bool valid;
    int last_used;
} tlb_entry_t;

// Returns the number of ticks since startup.
// Hint: You may want to use this to implement LRU.
int get_ticks();

// Initializes the relevant fields of the TLB.
void tlb_init(tlb_t* tlb) {
    for (int i = 0; i < NUM_ENTRIES; i++) {
        tlb_entry_t* tlb_entry = &tlb->tlb_entries[i];
        __________[A]x3__________;
    }
}


// Returns the physical page number corresponding to the
// given virtual page number. Return -1 if the appropriate entry
// does not exist.
int tlb_lookup(tlb_t* tlb, int vpn) {
    for (int i = 0; i < NUM_ENTRIES; i++) {
        tlb_entry_t* tlb_entry = &tlb->tlb_entries[i];
        if (__________[B]__________) {
            __________[C]x3__________
        }
    }
    __________[D]__________
}

// Insert the given VPN-to-PPN translation into the TLB,
// evicting entries as necessary.
void tlb_insert(tlb_t* tlb, int vpn, int ppn) {
    tlb_entry_t* lru_entry = NULL;
    for (int i = 0; i < NUM_ENTRIES; i++) {
        tlb_entry_t* tlb_entry = &tlb->tlb_entries[i];

        // Check if the current entry is unused.
        if (__________[E]__________) {
            __________[F]x5__________
            return;
        }
```

```
        // Update `lru_entry` appropriately.
        ___________[G]x6___________
    }
    // Evict the LRU TLB entry.
    ___________[J]x5___________
    return;
}
```

(a) (1.0 pt) [A] (max 3 lines)

```
tlb_entry->valid = false;
```

(b) (1.0 pt) [B]

```
tlb_entry->valid && tlb_entry->vpn == vpn
```

(c) (2.0 pt) [C] (max 3 lines)

```
tlb_entry->last_used = get_ticks();
return tlb_entry->ppn;
```

(d) (1.0 pt) [D]

```
return NULL;
```

(e) (1.0 pt) [E]

```
!tlb_entry.valid
```

(f) (3.0 pt) [F] (max 5 lines)

```
tlb_entry->valid = true;
tlb_entry->vpn = vpn;
tlb_entry->ppn = ppn;
tlb_entry->last_used = get_ticks();
```

(g) (3.0 pt) [G] (max 6 lines)

```
if (lru_entry == NULL)
    lru_entry = tlb_entry;
else if (lru_entry->last_used > tlb_entry->last_used)
    lru_entry = tlb_entry;
```

(h) (3.0 pt) [H] (max 5 lines)

```
lru_entry->vpn = vpn;
lru_entry->ppn = ppn;
lru_entry->last_used = get_ticks();
```

8. Reference Sheet

```
/* Semaphore */
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);

/* Lock */
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* Condition Variable */
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *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);

/* 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[]);

/* Thread */
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);

/* 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);

/* 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);

/* Pintos Interrupts */
enum intr_level {
  INTR_OFF, /* Interrupts disabled. */
  INTR_ON   /* Interrupts enabled. */
};
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);
```