CS 162 Fall 2023

Operating Systems and Systems Programming

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 75 minutes to complete as much of the exam as possible. This exam is out of 75 points. 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. However, answers that are illegible will receive NO CREDIT.

(a) Full 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. (18 points) True/False

Please explain your answer in TWO SENTENCES OR LESS. Longer explanations will GET NO CREDIT. Answers without an explanation or that just rephrase the question will GET NO CREDIT.

(a) (2 points)

i. A thread can starve even when you use a preemptive scheduler.

- (●) True.
- ( ) False.

ii. Explain.

Although preemption means schedulers can run certain tasks sooner, it does not always mean that there is fair distribution of resources. For example, preemptive priority schedulers lead to the starvation of lower-priority tasks.

(b) (2 points)

i. Any scheduler that minimizes context-switching overhead will necessarily minimize average completion time.

- ( ) True.
- (●) False.

ii. Explain.

First-come first-serve minimizing the number of context switches needed, but it can perform poorly with respect to average completion time (e.g. due to the convoy effect).

(c) (2 points)

i. For the same tasks, a First Come First Serve (FCFS) scheduler will always have a strictly worse average completion time compared to a Shortest Time to Completion First (STCF) scheduler.

◯ True.

● False.

ii. Explain.

If all of the tasks take the same amount of time to be executed, STCF becomes FCFS, so their average completion time would be the same.

(d) (2 points)

i. To fix priority inversion, it suffices for each thread to donate their priority only to the thread that owns the resource it is trying to acquire.

◯ True.

● False.

ii. Explain.

The high priority thread might also have to recursively donate their priorty to the other threads that the thread that owns the resource it wants to acquire is waiting for. Otherwise, there could still be some priority inversion if the other thread is also forced to wait on another thread.

(e) (2 points)

i. In the Linux O(1) scheduler, user tasks preempt real-time tasks in order to promote interactivity.

○ True.

● False.

ii. Explain.

The Linux O(1) scheduler gives user threads lower priorities than kernel and real-time tasks so that critical kernel functions are addressed first, so no this preemption would not occur.

(f) (2 points)

i. Base and bound allows two different processes to share the same code segment.

○ True.

● False.

ii. Explain.

With base and bound each process has separate address spaces and cannot access each other's address space.

(g) (2 points)

i. Simple segmentation-based address translation systems can suffer from external fragmentation.

● True.

○ False.

ii. Explain.

True. This is because segments cannot be rearranged for free but are of arbitrary sizes. Thus, there can be arbitrarily sized gaps between segments which might not fit a single larger segment.

(h) (2 points)

i. To represent the same virtual address space, a two level page table will take up the same amount of space as a single level page table.

Clarification: *All entries in the page tables are filled.*

○ True.

● False.

ii. Explain.

The full two level page table takes up more space than the single level page table.
All the entries in the second level page tables correspond to the entries in the single level page table. However, the two level page table additionally contains the top level table, causing it to take up more space.

(i) (2 points)

i. An eviction policy that evicts entries uniformly at random performs better than LRU for a workflow that repeatedly loops over the elements of an array that is too large to fit in cache (e.g. workload were you access `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[0]`, `a[1]`, `a[2]`, `a[3]`, ...).

● True.

○ False.

ii. Explain.

LRU is bad for looping workloads, as it ends up initiating a page fault for every page access after the first loop. Random actually does much better since it won't predictably evict all the pages we need on each loop.

2. (12 points) Multiple Select

(a) (2 pt) Select the following statements about schedulers that are true.

- ☐ In the Linux CFS, threads with higher nice values accumulate more physical CPU time per physical CPU cycle than threads with lower nice values.
- ☐ A lottery scheduler with at least 1 ticket per thread will guarantee that all threads eventually run to completion.
- ■ In a stride scheduler, threads with a high stride are scheduled less frequently than threads with a low stride.
- ■ For a real-time scheduler where predictability is critical, we should optimize worst-case performance over best-case performance.
- ☐ None of the above

A: They accumulate more *virtual* CPU time. B: Deadlock is still possible. C: Higher stride means a thread will have to wait for longer before being scheduled again. D: If we optimize for best case performance instead, we lose an upper bound on time spent waiting.

(b) (2 pt) Select the following factors that can lead to starvation in a multi-threaded system.

- ■ Circular waiting of resources.
- ☐ The use of a stride scheduler.
- ☐ Giving out unequal amounts of CPU to different threads.
- ■ The use of a strict priority scheduler with two priorities, 0 (low) and 1 (high).
- ☐ None of the above.

This is one of the necessary but not sufficient conditions of a deadlock, which is a stronger condition of starvation. A stride scheduler deterministically ensures that every task will run. One form of this is the Linux Completely Fair Scheduler, which helps prevent starvation. Any strict priority scheduler will suffer from starvation for lower priority threads.

(c) (2 pt) Select the following statements about base and bound that are true.

- ■ The base and bound values are stored in registers.
- ■ A process is allowed to access memory at addresses greater than the base, and less than the base plus bound.
- ☐ Base and bound restricts what memory addresses can be executed in kernel mode.
- ☐ Base and bound does not experience internal fragmentation.
- ☐ None of the above.

A: True: The base and bound are stored in registers. B: True: This is the range of valid addresses for a base and bound. C: False: Kernel mode executes without the base and bound registers. D: False: Base and bound experiences internal fragmentation, since there may be gaps inside the allocated region where it isn't fully used up.

(d) **(2 pt)** Select the following address translation schemes that can support sharing memory between arbitrary processes.

- ☐ Base and Bound.
- ☐ Base and Bound with Relocation.
- ■ Segmentation.
- ■ Paging.
- ☐ None of the above.

(e) **(2 pt)** Select the following changes that could reduce the total number of cache misses for your program.

- ■ Increasing cache size.
- ■ Increasing cache associativity.
- ■ Decreasing cache associativity.
- ■ Rearranging the layout of data structures.
- ☐ None of the above.

A: If your application is getting a lot of cache misses, it is likely because your cache is too small and an increased cache size will reduce the number of misses. B: If your application is getting a lot of conflict misses (accessing successive addresses that map to the same cache line) then you can reduce misses by increasing associativity (e.g. from direct mapped to 2-way set associative). C: If you have a fully associative cache of size N and you repeatedly cycle through N+1 addresses, every access will be a miss. By going to a direct mapped cache, you reduce it to 2 misses for every N+1 accesses. D: By rearranging the in-memory layout of data structures, you can avoid misses in a cache which has conflict misses.

(f) **(2 pt)** Select the following statements about virtually- and physically-addressed caches that are true.

- ☐ A physically-addressed cache maps PPNs to VPNs.
- ☐ A physically-addressed cache needs to be invalidated on process switch.
- ■ A virtually-addressed cache does not need to be manually invalidated on process switch if it stores extra metadata.
- ☐ A virtually-addressed cache maps from virtual addresses to physical addresses.
- ☐ None of the above.

A: Physically-addressed caches map from physical addresses to data. B: Physically-addressed caches do not need to be invalidated as they are after address translation. C: A virtually-addressed cache can store a process ID and on every access can compare if the process ID matches the currently running process. D: A virtually-addressed cache maps from virtual addresses to data.

3. (13 points) Short Answer

(a) (2 pt) A client wants to request and receive a file from an HTTP server. How many sockets each do the client and server need to achieve this, and what is each one used for? *Limit your response to 2 sentences.*

- **2 for server: 1 for listening for new connections, 1 for client connection send/receive**
- **1 for client: used to connect to server**

(b) (2 pt) Identify one advantage segmentation has over paging.

Sample answers: 1. logically organizes the program since segments are split according to logical portions, 2. Suffers less from internal fragmentation.

(c) (2 pt) List the 4 approaches to deal with deadlock in systems.

Clarification: It suffices to just list the methods – no descriptions are necessary.

i. **Prevention (write correct code)**
ii. **Recovery (rollback when it happens)**
iii. **Avoidance (delay resource requests to maintain safe state)**
iv. **Denial (Ostrich algorithm; just reboot system if it happens)**

(d) (3 points)

In the following questions, we will be asking about different design decisions and implementation details of `sema_down` in PintOS.

i. **(1 pt)** How do we ensure that there are no race conditions when accessing the appropriate data structures inside of the implementation of `sema_down`? *Limit 2 sentences.*

In both `sema_down` and `sema_up`, we disable interrupts before performing any further operations.

ii. **(2 pt)** Suppose that a thread calls `sema_down` when the value of the semaphore is already 0. How do we prevent this thread from being scheduled while it is waiting? *Limit 2 sentences.*

We set the status of the thread to `THREAD_BLOCKED` and do not put it on the ready queue.

(e) **(4 pt)** Assume a machine uses paged segmentation and has a page size of 1024 bytes, physical memory of 1 MiB, 16-bit virtual addresses, and records 4 bits of metadata for each page table entry. Further assume that each program has exactly four segments of equal size across the full address space and that every virtual address is accessible. How large is a page table for a segment in bytes? *Show your work.*

num bits in virtual address = num bits for segment + num bits for VPN + num bits offset
num bits for VPN = num bits in virtual address - num bits for segment - num bits offset = 16 - 2 - 10 = 4 bits
num page table entries per segment = num virtual pages per segment = 2^(# bits for VPN) = 16 entries
PTE size in bits = num bits PPN + num bits metadata = log_2($2^{20/2}$10) + 4 = 14
PTE size in bytes = PTE size in bits / 8 = 7/4 bytes (valid to round to 2)
page table size per segment in bytes= PTE size in bytes * # page table entries per segment = 7/4 * 16 = 28 bytes (2 * 16 = 32 bytes also valid)

4. (8 points) Short Answer Coding

(a) (4 points)

Suppose we have a group of tasks that we want to schedule, where each task has a known completion time stored in the `duration` field. After each task finishes, we call `next_task_to_run` to find the next task to start running. Complete the implementation `next_task_to_run` below to use Shortest Job First (SJF). Assume that `ready_list` is sorted in ascending order by the time at which the task was added to the queue.

```
struct task {
    ...
    struct list_elem elem;
    int duration;
}

struct list ready_list;

/* Chooses the next task to run, or returns NULL if the queue is empty. */
struct task* next_task_to_run() {
    struct task* chosen = NULL;
    struct list_elem* e;

    for (e = list_begin(&ready_list); e != list_end(&ready_list); e = list_next(e)) {
        struct task* current = list_entry(e, struct task, elem);
        if (_______[A]_______) {
            _______[B]_______
        }
    }

    return chosen;
}
```

Clarification: For part (a) only, you do not need to remove the element from the list. You will need to do so in part (b).

i. (2 pt) [A] (max 1 line)

```
chosen == NULL || current->duration < chosen->duration
```

ii. (1 pt) [B] (max 1 line)

```
chosen = current;
```

iii. (1 pt) Say you want to use First Come First Served (FCFS) instead. Re-implement `next_task_to_run` using 4 lines or fewer.

```
if (!list_empty(&ready_list))
    return list_entry(list_pop_front(&ready_list), struct task, elem);
return NULL;
```

(b) (4 points) Banker's Drives

We have a system where threads can access a shared set of drives through this API:

- `void request_more_drives(int num_drives)`: blocks until the OS allocates `num_drives` more drives to this thread
- `void free_all_drives()`: releases all drives held by this thread back to the OS

We have a deadlock avoidance procedure powered by Banker's algorithm (not shown) that checks calls to `request_more_drives` and returns whether the request would place the system in either a safe or unsafe state. We want to test this procedure by finishing the following example test case:

- The system starts with 12 drives total.
- 0 drives are initially allocated to thread A.
- 4 drives are then initially allocated to thread B.
- `bool rand_bool()` randomly returns true or false each time it is called. Consider the system *before* the functions `thread_A` and `thread_B` have started running.

Fill in the blanks in `thread_A` and `thread_B` such that the deadlock avoidance procedure would return "unsafe" given the first request in `thread_A`, but the system would not always deadlock if run. There may be multiple correct solutions.

```
// Will run in thread A
void thread_A() {
    request_more_drives(3);
    request_more_drives(______[A]_______);
    free_all_drives();
}

// Will run in thread B
void thread_B() {
    if (_______[B]_________) {
        __________[C]____________;
    }
    _________[D]__________;
}
```

i. (1 pt) [A] (max 1 expression)

```
some int > 5 and <= 9
```

ii. (1 pt) [B] (max 1 expression)

```
rand_bool()
```

iii. (1 pt) [C] (max 1 line)

```
request_more_drives(i);
where i > 5 and i < 9
```

iv. (1 pt) [D] (max 1 line)

```
free_all_drives();
```

5. (11 points) Paging

In this problem, we will construct a simple simulation in C to see how address translation works with a multi-level page table. Here's the specification of our system:

```
32-bit virtual address    ->  [ Level 1 (8 bits) | Level 2 (8 bits) | Offset (16 bits)   ]
32-bit physical address   ->  [ Physical Page Number (16 bits)      | Offset (16 bits)   ]
32-bit page table entry   ->  [ Physical Page Number (16 bits)      | Metadata (16 bits) ]
```

The PTEs are stored in big-endian form in memory (i.e. the MSB is first byte in memory).

Complete the program below with respect to the memory translation scheme given. Note that in this problem, the size of the `unsigned` data type is 4 bytes.

Note: It will be helpful to recall that, in C, `&` and `|` can be used for bitwise AND and OR respectively, and that `x >> n`, and `x << n` can be used to compute a bitshift of value `x` right or left by `n` bits respectively. You can also define hexademical numeric literals by prefixing the number with `0x` (e.g. 15 becomes `0xF`).

```
#include <stdio.h>
#include <stdlib.h>

typedef struct physical_memory {
    ...
} physical_memory_t;

typedef struct pte {
    short ppn;
    short metadata_bits;
} pte_t;

// Initialize physical memory with the multi-level page table.
void init(physical_memory_t* mem);

// Return the PTE at the corresponding address.
pte_t get_pte(physical_memory_t mem, unsigned addr);

// Get the base table pointer for the current process.
unsigned get_base();

int main() {
    physical_memory_t mem;
    init(&mem);
    unsigned base_table_pointer = get_base();
    unsigned vaddr = 0x3271798;
    unsigned offset = ______[A]______;
    unsigned first_level_index = ______[B]______;
    unsigned second_level_index = ______[C]______;
    pte_t first_level_pte = ______[D]______;
    unsigned second_level_pt_addr = ______[E]______;
    pte_t second_level_pte = ______[F]______;
    unsigned paddr = ______[G]______;
    printf("VADDR -> PADDR: 0x%08x -> 0x%08x", vaddr, paddr);
    return 0;
}
```

(a) (1 pt) [A] (max 1 expression)

```
vaddr & 0xFFFF or 0x1798 or 6040
```

(b) (1 pt) [B] (max 1 expression)

```
vaddr >> 24 or 0x3 or 3
```

(c) (1 pt) [C] (max 1 expression)

```
(vaddr >> 16) & 0xFF or 0x27 or 39
```

(d) (1 pt) [D] (max 1 expression)

```
get_pte(mem, base_table_pointer + first_level_index * sizeof(pte_t))
```

(e) (1 pt) [E] (max 1 expression)

```
first_level_pte.ppn << 16
```

(f) (1 pt) [F] (max 1 expression)

```
get_pte(mem, second_level_pt_addr + second_level_index * sizeof(pte_t))
```

(g) (1 pt) [G] (max 1 expression)

```
(second_level_pte.ppn << 16) | offset
```

Assume the memory translation scheme given above. Suppose that the base table pointer for the current user level process is 0x200000. Translate the following virtual addresses to physical addresses (in hex), using the memory contents given on the reference sheet. Make sure to truncate leading zeros in your answer.

(h) **i. (1 pt)** 0x31047

0x1047

ii. (1 pt) 0x8006047

0x6047

iii. (1 pt) 0x2026047

0x306047

iv. (1 pt) 0x3060162

0x67200162

6. (13 points) Pintos Multi-Level Feedback Queue

In this problem, we will implement a basic multilevel feedback queue for Pintos. Assume that we want to promote the current thread's priority by 1 level if it has yielded, and demote it by 1 level if has not, subject to the constraint that priorities should remain between 0 and 63 (inclusive).

You may assume that there is a `yielded` flag that indicates whether the most recently run thread yielded the CPU, which is appropriately modified by `schedule` and `thread_yield`.

```
#define MIN_PRIORITY 0
#define MAX_PRIORITY 63
#define MIN(X, Y) ((X) < (Y)) ? (X) : (Y)
#define MAX(X, Y) ((X) > (Y)) ? (X) : (Y)

struct thread {
    /* ... other fields omitted ... */
    struct list_elem elem;
    int priority;
    bool yielded;
    enum thread_status status;
};

/* Idle thread, scheduled when there are no other available threads. */
static struct thread* idle_thread;
/* Lists of ready threads, indexed by their priority. */
static struct list ready_queues[64];

/* Return the highest priority thread available, resolving ties according to FIFO.
If there are no threads that are ready, return the idle thread instead. */
struct thread* next_thread_to_run(void) {
    for (_______[A]_______) {
        ________[Bx4]________
    }
    _______[C]_______
}

/* Pushes the given thread into the appropriate ready queue. Responsible for promoting
or demoting the priority of this thread, depending on whether it yielded or not. */
static void thread_enqueue(struct thread* t) {
    if (yielded) {
        _______[D]_______
    } else {
        _______[E]_______
    }
    _______[F]_______
}
```

Clarification: The code for `thread_enqueue` should read `if (t->yielded)`.

(a) (1 pt) [A] (max 1 line)

```
int i = MAX_PRIORITY; i >= MIN_PRIORITY; i--
```

(b) (2 pt) [B] (max 4 lines)

```
if (!list_empty(&ready_queues[i])) {
return list_entry(list_pop_front(&ready_queues[i]), struct thread, elem);
}
```

(c) (1 pt) [C] (max 1 line)

```
return idle_thread
```

(d) (1 pt) [D] (max 1 line)

```
t->priority = MIN(t->priority + 1, MAX_PRIORITY)
```

(e) (1 pt) [E] (max 1 line)

```
t->priority = MAX(t->priority - 1, MIN_PRIORITY)
```

(f) (1 pt) [F] (max 1 line)

```
list_push_back(&ready_queues[t->priority], &t->elem)
```

(g) (2 pt) Note that our implementations above rely on the `yielded` flag being set appropriately. Suppose that we have added a line initializing `yielded` to `false` in `schedule` and now we need to modify `thread_yield` to set it to true. Consider starting with the following implementation of `thread_yield`.

```
1: void thread_yield(void) {
2:     struct thread* cur = thread_current();
3:     enum intr_level old_level = intr_disable();
4:     if (cur != idle_thread) {
5:         thread_enqueue(cur);
6:  }
7:     cur->status = THREAD_READY;
8:     schedule();
9:     intr_set_level(old_level);
10: }
```

Select of all the following locations in which we could insert `yielded = true` and produce the correct behavior.

Clarification: should read `cur->yielded = true` instead of `yielded = true`.

- [x] Between lines 4 and 5.
- [ ] Between lines 5 and 6.
- [ ] Between lines 6 and 7.
- [ ] Between lines 8 and 9.
- [ ] None of the above.

Note that we need to mark that the thread has yielded before we make the call to `thread_enqueue`, as otherwise we will not be able to promote/demote the priority appropriately.

(h) **(2 pt)** Suppose that instead of demoting the priority of threads that fail to yield, we instead keep it at the same priority. Explain, in two sentences or fewer, how users can exploit this.

A user might intentionally write a handful of yields at the start of their program to get it promoted to the top queue, and then it will stay there forever even if it is performing a long-running computation.

(i) **(2 pt)** In terms of design, why do we want promote threads that yield to higher priority queues?

Threads that yield often are frequently interactive threads requiring user I/O, so we want them to be more responsive.

7. Extra Credit

(a) (1 pt) What is our professor's full name?

Natacha Crooks

8. Empty Page

9. Empty Page

10. Physical Memory in Bytes

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00000000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
00000010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
...																
00001010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
00001020	40	03	41	01	30	01	31	03	00	03	00	00	00	00	00	00
00001030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
00001040	10	01	11	02	31	03	13	04	14	01	15	03	16	01	17	00
...																
00100000	00	00	10	65	00	00	20	67	00	30	03	00	00	00	40	07
00100010	00	00	50	03	00	00	00	00	67	20	20	67	00	00	00	00
...																
00102000	00	20	01	00	00	00	00	00	00	30	03	00	00	40	00	07
...																
00103000	11	22	00	05	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
00103010	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	67
...																
001F0000	00	00	20	00	00	00	10	65	00	00	10	67	00	00	20	65
001F0010	00	00	20	67	00	00	30	67	00	00	40	65	00	00	50	07
...																
001FFFF0	00	00	00	00	00	00	00	00	10	00	00	67	00	00	40	65
...																
00200000	00	1F	F0	07	00	10	10	07	00	10	20	07	00	10	30	07
00200010	00	10	20	07	00	10	50	07	00	10	60	07	00	10	70	07
00200020	00	10	00	07	00	00	00	00	00	00	00	00	00	00	00	00
...																
00200FF0	00	00	00	00	00	00	00	00	00	1F	E0	07	00	1F	F0	07
...																

11. Reference Functions

```
/* 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_broadcast(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_insert_ordered(struct list* list, struct list_elem* elem, list_less_func* less, void* aux);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
struct list_elem* list_pop_front(struct list* list);
struct list_elem* list_pop_back(struct list* list);

/* 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 open(const char *pathname, int flags, mode_t mode);
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);
```

```
/* Memory */
void *memcpy(void *dest, const void * src, size_t n)
```

12. Powers of 2

$2^5 = 32$

$2^6 = 64$

$2^8 = 256$

$2^{10} = 1024$

$2^{20} = 1048576$