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.

(b) (2 points)

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

◯ True.

◯ False.

ii. Explain.

(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.

(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.

(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.

(f) (2 points)

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

◯ True.

◯ False.

ii. Explain.

(g) (2 points)

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

◯ True.

◯ False.

ii. Explain.

(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.

◯ True.

◯ False.

ii. Explain.

(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.

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

(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.

(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.

(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.

(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.

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.*

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

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

(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.*

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.*

(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.*

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;
}
```

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

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

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.

(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)

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

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

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

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)

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

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

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

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

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

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

(h) 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.

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

ii. (1 pt) 0x8006047

iii. (1 pt) 0x2026047

iv. (1 pt) 0x3060162

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]_______
}
```

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

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

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

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

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

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

(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.

☐ Between lines 4 and 5.

☐ Between lines 5 and 6.

☐ Between lines 6 and 7.

☐ Between lines 8 and 9.

☐ None of the above.

(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.

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

7. Extra Credit

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