CS552 Operating Systems **Spring 2020**

24-Hour Written Assignment

- Please gsubmit your solution by 8:00pm US Eastern time (GMT-4) on Wednesday, April 29.
- All solutions are to be produced individually.
- Please do not consult with others.
- The time allotted should be sufficient no matter where you are in the world.
- A small bonus will be given in the timestamped order of submitted solutions. The formula will be: $0.25 \times (min(class\_size, \#submissions) - order\_of\_submission)$. If you are the first submission and everyone in the class of 57 students submits a solution, then you will receive 14 bonus points. I will factor timezone differences.
- You must either print the assignment and hand write your solutions in the space provided or use separate paper to hand write your solutions. You should then use a camera, scanner or smartphone to take clear images of your solution, in gif, jpg, png, or pdf format. **A single pdf document is the preferred format.**
- Please gsubmit to a folder called **takehome** a file called **takehome.zip** or **takehome.tgz**, which includes the compressed solution file to your written assignment. Do NOT type your solution, but instead make your written solution as clear as possible. It is your responsibility to make sure the solution is readable for grading.

1. **Scheduling and Context Switching.**
   - **(8 points)** Consider a proportional share scheduler (e.g., EEVDF) for a set of n tasks competing for a single CPU. If each task has a time quantum, q, what is the worst-case *lag* between ideal and actual usage of the CPU experienced by any one task? **Prove your answer**. NOTE: You can assume all tasks have equal weights (and, hence, priorities).

2. **Address Translation.**

Consider a *32-bit x86 system* that has combined segmentation and paging. Let logical addresses be specified as <segment selector>:<segment offset>. The segment selector is 16 bits, Each 8-byte entry in a segment table maintains a 32-bit *base* linear address and a corresponding 32-bit *limit* value, interpreted according to the Intel IA-32 Specification. The following partial page table shown in Table 1, along with the (partial) segment table shown in Table 2 have been initialized by an OS, working with a page-size of 4KB:

Page Number	**Frame Number**
1024	72
1023	17
1022	9
1021	43
1020	212
1019	127
1018	56
1017	6
1016	100

Table 1: Page Table

Segment Number	**Base Linear Address**	**Limit**
0x1770	0x00000000	0x0000FFFF
0x1771	0x00010000	0x0001FFFF
0x1772	0x00020000	0x0002FFFF
0x1773	0x00030000	0x003FFFFF
0x1774	0x00000000	0xFFFFFFFF

Table 2: Segment Table

(a) **(4 points)** What is the physical address corresponding to the logical address <0xBB88>:<0x3F0000>? You only need to calculate a valid address if it is within the segment limits, otherwise simply state that a segment violation has occurred.

(b) **(4 points)** What is the physical address corresponding to the logical address <0xBB98>:<3FE00D>? You only need to calculate a valid address if it is within the segment limits, otherwise simply state that a segment violation has occurred.

(c) **(4 points)** What is the physical address corresponding to the *linear* address 4181920 (base 10)?

NB: In the above questions, please explain your approach to perform address translation. Showing the answers without correct explanation will not receive credit.

(d) **(6 points)** Consider a virtual machine system in which *guest* processes and their OSes are *hosted* by a virtual machine monitor (VMM). In this system, guest virtual addresses (GVAs) are translated to guest physical addresses (GPAs) using *two-level* page tables managed by the corresponding guest OSes, within their own virtual machines. GPAs are then translated to machine physical addresses (MPAs) using *three-level* shadow page tables managed by the VMM.

In the absence of a TLB, how many (machine) physical memory references are needed to access the contents of a translated GVA? Explain your answer by describing the *basic* steps needed to translate a GVA to a final MPA, using a diagram where appropriate.

3. **Critical Sections, Semaphores, Condition Variables, Threads and Deadlocks.**

As part of a Halloween adventure you and some friends are going to a haunted house. The haunted house has two entrances for visitors: one on the east side, and one on the west side. Initially, *nine* lanterns and *nine* skeleton keys are located at the east-side entrance. Visitors from that side must each take *one* lantern and *two* keys when they enter the house, and they each must leave their lantern and keys for visitors at the west-side entrance after exiting the house. Likewise, visitors arriving at the west-side must wait to acquire a lantern and two keys before entering the house and proceeding through the haunted rooms to the east-side.

You can assume arbitrary visitors arrive at either entrance and each may be in one of three states: *waiting_to_enter*, *walking_through_house*, and *outside_haunted_house*. Initially, each visitor is *outside_haunted_house* but moves into state *waiting_to_enter* when attempting to obtain a lantern and pair of keys. When all resources are acquired, a visitor can enter into state *walking_through_house*.

Assume that you have the following counting semaphores:
lanterns_east, *lanterns_west*, *keys_east* and *keys_west*.

(a) **(14 points)**
Using the above *counting semaphores* for lanterns and keys, as well as *mutex* variables (or binary semaphores) where appropriate, write the pseudo-code using *wait()* and *signal()* operations on your semaphores, for visitors to *both* east and west entrances. Think of each visitor as a thread of control. If you need additional mutex variables besides the counting semaphores shown above, please indicate what they are and how you use them. Your solution will be correct if arriving visitors at either entrance can successfully move through the house without a deadlock. NB: You don't need to show the internals of wait() and signal().

(b) **(6 points)** In your solution, what is the maximum number of visitors at any one entrance that can be holding one key and waiting for another, without leading to a deadlock? For *any* valid solution with a maximum of N visitors waiting at any one entrance, what is the largest number that can be holding one key and waiting for another, without leading to a deadlock? Briefly explain why.

(c) **(6 points)** Considering the following piece of POSIX code using pthreads to implement a consumer, in a producer-consumer problem, using a shared circular buffer of N slots, briefly explain what is wrong with the consumer and how it should be fixed:

```
typedef struct {
  char *data; // Slot data.
  size_t size; // Size of data.
  pthread_t id; // ID of destination thread.
} slot_t;

typedef struct {
  slot_t slot[N];
  int count; // Number of full slots in buffer.
  int in_marker; // Next slot to add data to.
  int out_marker; // Next slot to take data from.
  pthread_mutex_t mutex; // Mutex for shared buffer.
  pthread_cond_t occupied_slot; // Condition when >= 1 full buffer slot.
  pthread_cond_t empty_slot;    // Condition when 1 empty buffer slot.
} buffer_t;

char *consume (buffer_t *b) {

  char *item;
  pthread_t self=pthread_self();

  pthread_mutex_lock (&b->mutex); // Don't worry about checking return values
  if (b->count <= 0)
    pthread_cond_wait (&b->occupied_slot, &b->mutex);

  assert (b->count > 0);

  // Check id of target thread to see message is for this thread.
  if (b->slot[b->out_marker].id != self) {
    // Data is not for this thread.
    pthread_mutex_unlock (&b->mutex);
    return NULL;
  }
  item = (char *) malloc (b->slot[b->out_marker].size);

  memcpy (item, b->slot[b->out_marker].data, b->slot[b->out_marker].size);
  free (b->slot[b->out_marker].data);
  b->out_marker++;
  b->out_marker %= BUFFER_SIZE;
  b->count--;

  pthread_mutex_unlock (&b->mutex);
  pthread_cond_signal (&b->empty_slot);

  return (item);
}
```

4. **Memory Ordering.**
(10 points) Consider the following code to implement a spinlock on an x86 multicore processor:

```
spinlock_lock:
.test:
  bts $0, (mlock) // Assume mlock is a memory variable
  jc .test
  ret

spinlock_unlock:
  btr $0, (mlock)
  ret
```

State what is wrong with the above code to implement the lock and unlock functionality, and explain how to fix it. How would you use `bts` and `btr` to implement an improved version of `spinlock_lock` and `spinlock_unlock`? Show the assembly code for the improved versions below.

5. **System Protection.**

   (a) **(3 points)** Suppose you had to setup a user-space at ring 3 on an x86 machine, with a flat 32-bit memory model for code and data. Show the bit-level values of the code and data segment descriptors you would need to add to a system such as FIFOS, which was one of the programming assignments. Explain the reasoning for the bit values.

(b) **(3 points)** Supppose you wanted to support a syscall using `int 0x80` that passed control from user-space to kernel-space. Show the bit settings for an x86 interrupt gate descriptor to allow the syscall to invoke a function `syscall_handler` at kernel address 0xFFC10000. Explain the reasoning for the bit values.

(c) **(6 points)** Is it safe for kernel code to directly call a user-space function using a `call` or `jmp` instruction? Explain your answer using a small amount of assembly code to describe the way a user-space program is started or resumed.

(d) **(4 points)** Suppose you have an interrupt service routine (ISR) on an x86 machine. Briefly explain if it is safe to issue a `push ret_addr; ret` pair of instructions to resume the interrupted code at `ret_addr` as the last two instructions of the ISR.

6. **Memory Management.**
Consider a system that utilizes a buddy memory allocator for allocations that are in powers-of-two contiguous page frames. Each page frame is 4KB is size. Suppose the system has 10 lists, where the ith list keeps track of contiguous blocks of memory of size 2^i frames (given $0 \leq i \leq 9$). The memory allocator starts out with one largest contiguous memory region, and the 10 lists are updated according to the rules of buddy memory management as memory is allocated and de-allocated.

   (a) **(8 points)** Show the final state of the buddy system lists after the following sequence of allocations (from left to right): 60KB, 8KB, 7KB, 57KB, 513KB. Call this state of the system "State A".

   (b) **(6 points)** Given the buddy system is in "State A" show the state of the buddy lists after the following sequence of memory deallocations (from left to right): 8KB, 57KB, 60KB. Note, you can assume these deallocations correspond to the allocations of the same size in Part (a), above, for purposes of determining whether free blocks are contiguous.

(c) *Slab* memory management is used in systems such as Solaris and Linux, to deal with frequent memory allocations and deallocations of commonly used objects (e.g., for file objects and process control blocks). A slab memory allocator divides memory into a series of *caches*, which hold allocated and free memory *objects*.

Suppose a system comprises page frames that are 4096 bytes. A slab memory allocator in this system is capable of allocating memory objects of size 2^k bytes, where $9 \leq k \leq 16$. That is, objects range in size from 2^9 to 2^{16} bytes, in powers of two, so there are 8 object sizes.

Let the slab allocator consist of 132 page frames, divided into *two* equal-sized caches, each having 8 slabs. Each slab in a cache stores objects of one size only, so there is one slab per cache for each object of size 2^k bytes.

NOTE: You can assume the page frames for one slab are *independent* of the page frames for another slab. Also, different slabs in a cache can be different sizes.

i. **(6 points)** Suppose at time, t, a number of object allocations and deallocations results in the *same number of free objects*, n_{free}, *in each slab*. What is the maximum value of n_{free}? (Show your calculations.)

ii. **(2 points)** Given your answer for n_{free}, what is the minimum number of page frames needed for the largest slab in a cache? (Explain your answer to receive credit)