1. [20 points] Jobs A, B, C have the following arrival times and service durations (in seconds):
   - A: arrival time 0.0; service duration 3.5.
   - B: arrival time 0.5; service duration 3.0.
   - C: arrival time 3.5; service duration 2.5.
   
   a. [4 points] Assume fifo ready queue (“runnable queue” in GeekOS), no pre-emption, and zero context switch time. Complete the following table with a row for each successive service interval; each row indicates the interval and job being served.

   **SOLUTION**

<table>
<thead>
<tr>
<th>Interval</th>
<th>Job served</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 – 3.5</td>
<td>A</td>
</tr>
<tr>
<td>3.5 – 6.5</td>
<td>B</td>
</tr>
<tr>
<td>6.5 – 9.0</td>
<td>C</td>
</tr>
</tbody>
</table>

   b. [16 points] Assume round robin with 1 second quantum, fifo ready queue, and zero context switch time. Complete the following table with a row for each service quantum. Indicate when a job departs.

   **SOLUTION**

<table>
<thead>
<tr>
<th>Interval</th>
<th>Job served</th>
<th>[run; ready front; · · ·; ready back] at end of quantum. Each job tagged with remaining service time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 – 1.0</td>
<td>A</td>
<td>[A 2.5; B 3.0]</td>
</tr>
<tr>
<td>1.0 – 2.0</td>
<td>B</td>
<td>[B 2.0; A 2.5]</td>
</tr>
<tr>
<td>2.0 – 3.0</td>
<td>A</td>
<td>[A 1.5; B 2.0]</td>
</tr>
<tr>
<td>3.0 – 4.0</td>
<td>B</td>
<td>[B 1.0; A 1.5; C 2.5]</td>
</tr>
<tr>
<td>4.0 – 5.0</td>
<td>A</td>
<td>[A 0.5; C 2.5; B 1.0]</td>
</tr>
<tr>
<td>5.0 – 6.0</td>
<td>C</td>
<td>[C 1.5; B 1.0; A 0.5]</td>
</tr>
<tr>
<td>6.0 – 7.0</td>
<td>B</td>
<td>[B 0.0; A 0.5; C 1.5]</td>
</tr>
<tr>
<td>7.0 – 7.5</td>
<td>A</td>
<td>[A 0.0; C 1.5]</td>
</tr>
<tr>
<td>7.5 – 8.5</td>
<td>C</td>
<td>[C 0.5]</td>
</tr>
<tr>
<td>8.5 – 9.0</td>
<td>C</td>
<td>[C 0.0]</td>
</tr>
</tbody>
</table>

   **Grading**  
   - 1 point for serving C instead of A in quantim 4.0–5.0
2. [30 points] This question concerns GeekOS.

a. [10 points] At the end of GeekOS initialization (and before the user does anything), how many threads exist and what is each thread doing.

**SOLUTION**

There are 9 threads: [1 point]
- Initial kernel thread (aka mainThread): waiting on initProcess. [1 point]
- Idle thread: paused at halt instruction (“current thread”) [2 points]
- Reaper thread: waiting for input (from reap queue). [3 points]
- Shell (aka initProcess) thread: waiting for input (from keyboard) [2 points]
- IDE_request thread: waiting to for input (from IDE request queue) [1 point]
- Floppy_Request_Thread [1 point]
- Alarm_Handler_Thread, Forwarding_Thread, Net_Device_Receive_Thread. [0 points]

**Grading** Points allocated as shown above. Lose points for assigning threads to every IO device. Lose points for assigning threads to memory management (?!), scheduler (?), syscalls, whatnot.

b. [5 points] During GeekOS initialization, Init_Keyboard installs an interrupt handler, but Init_Screen does not. Why not?

**SOLUTION**

There are two reasons:
First, the screen is an output-only device.
Second, the screen and CPU interact via video memory, so CPU does not have to wait between successive outputs to screen.

**Grading** 3 points for only one reason.

(Can keyboard inputs be done (perhaps inefficiently) without interrupts?)
c. [5 points] During GeekOS initialization, does Init_IDE have to install an interrupt handler? Explain briefly.

**SOLUTION**

**Answer 1:** [3 points] There should be an interrupt handler (for efficiency) because the CPU can issue IDE requests (input or output) faster than the IDE can handle them.

**Answer 2:** [5 points] It’s not necessary to have an interrupt handler. The CPU can simply busy wait for IDE IO to finish. Inefficient but doable.

(What does GeekOS do?)

d. [10 points]

In GeekOS, from an interrupt occurrence to the interrupt handler being executed, the CPU does an action and then executes code involving `Handle_Interrupt`, `g_entryPointTable`, `s_IDT`, `g_interruptTable`.

Write down the order in which these are done and briefly state happens in each.

**SOLUTION**

First, CPU does the following action (in hardware):

- if user level thread was interrupted, push user SS and SP on (kernel) stack.
  - push EFLAGS, CS, EIP, and error code (if present) on stack
  - get new CS, EIP, privilege level from `s_IDT`;

[3 points] [2 points]

Second, CPU executes code in `g_entryPointTable`:

- push an error code (if not already present) on stack
  - push interrupt number on stack

[2 points]

Third, CPU executes code in `Handle_Interrupt`:

- pushes rest of CPU regisiters, constructing “interrupt state”
- go to address pointed to be `g_interruptTable` entry.

[2 points]
3. [20 points] You are given buffer buff of max size N items and the following non-blocking functions:
num(), returns the number of items in buff; add(x), adds item x to buff; and rmv(), removes and returns an item from buff. Initially buff is empty.

Obtain functions enQ(x) and deQ() that satisfy the following requirements.

1. They can be called by multiple threads simultaneously.
2. Semaphores are the only synchronization construct (no atomic read-modify-write, no disabling interrupts, no access to PCBs, no wait/wakeup, etc.). No busy waiting.
3. enQ(x) calls add(x) exactly once, waiting if num() = N holds.
4. deQ() calls rmv() exactly once, waiting if num() = 0 holds.
5. If a thread is in enQ and num() < N holds, then an enQ invocation returns.
6. If a thread is in deQ and num() > 0 holds, then an rmv invocation returns.

Be neat and clear. You lose points if I can’t understand your code in a reasonable time.

SOLUTION 1 (from multi-threading note)

Shared variables:
Semaphore(1) mutex
Semaphore(0) gateE // enQ thread waits here if buff full
int nE ← 0 // tracks number of enQ threads waiting on gateE
Semaphore(0) gateD // deQ thread waits here if buff empty
int nD ← 0 // tracks number of deQ threads waiting on gateD

enQ(x):
mutex.P()
if num() = N
nE ← nE + 1
mutex.V()
gateE.P()
else
nE ← nE - 1
add(x)
if nD > 0
gateD.V()
else
mutex.V()

SOLUTION 2 (from multi-threading note)

Shared variables:
Semaphore(1) mutex
Semaphore(N) spaces // enQ thread waits here if buff full
Semaphore(0) items // deQ thread waits here if buff empty

enQ(x):
spaces.P()
mutex.P()
add(x)
mutex.V()
items.V()

deQ():
mutex.P()
if num() = 0
nD ← nD + 1
mutex.V()
gateD.P()
nD ← nD - 1
x ← rmv()
if nE > 0
gateE.V()
else
mutex.V()
return x

Grading
• 15 points if you had the framework but details were wrong.
• 10 points if you had only some elements of the framework.
• 5 points if your solution did not satisfy requirement 2 but otherwise worked. (You’ve had ample warning about this, in class, in the notes, and in the practice exam.)
4. [10 points] This extends problem 3.
You are given buff, num(), add(x), and rmv() as in problem 3.
Obtain functions enQ(x), deQ() and enQ2(x) that satisfy the following requirements.
1 – 6. Same as in problem 3.
7. enQ2(x) calls add(x) exactly twice, waiting if num() ≥ N-1 holds.
8. If (a thread is in enQ2 or enQ) and (num() < N-1 holds continuously),
then an invocation of enQ2 or enQ returns.

Be neat and clear. You lose points if I can’t understand your code in a reasonable time.

NOT A SOLUTION
It’s not clear how to solve this along the lines of solution 2 in problem 3. A natural attempt is to define
enQ and deQ as in problem 3 and define enQ2 as follows:

```c
enQ2(x):
    mutex.P()
    if num() >= N-1
        nE2 ← nE2 + 1
        mutex.V()
        gateE2.P()
        nE2 ← nE2 - 1
    add(x)
    if nD > 0
        gateD.V() // 1 V
    else
        mutex.V()
```

This does not satisfy requirement 5 as follows.
Suppose there is only one space in buff,
threads call enQ and enQ2,
and the enQ2 thread completes the first spaces.P().
Then both threads are stuck, which violates requirement 5
(num() < N holds and the enQ thread is stuck).

SOLUTION
Whereas it can be solved along the lines of solution 1 in problem 3.

```c
Shared variables:
    Semaphore(1) mutex
    Semaphore(0) gateE // for enQ thread to wait
    int nE ← 0 // number of enQ threads waiting
    Semaphore(0) gateE2 // for enQ2 thread to wait
    int nE2 ← 0 // number of enQ2 threads waiting
    Semaphore(0) gateD // for deQ thread to wait
    int nD ← 0 // number of deQ threads waiting

enQ(x):
    mutex.P()
    if num() = N
        nE ← nE + 1
        mutex.V()
        gateE.P()
        nE ← nE - 1
    add(x)
    if nD > 0
        gateD.V()
    else
        mutex.V()
```

```c
deQ():
    mutex.P()
    if buff.size = 0
        nD ← nD + 1
        mutex.V()
        gateD.P()
        nD ← nD - 1
        x ← buff.remove
    if (nE > 0)
        gateE.V()
    else if (nE2 > 0 and num() < N-1)
        gateE2.V()
    else if nD > 0
        gateD.V() // note
    else
        mutex.V()
        return x
```

Grading
- 5 points if your solution works except for requirement 5.
  E.g., the one under “NOT A SOLUTION” gets 5 points.
- Zero points if you did not satisfy requirement 2.