P0. A heap of trouble (20 points)

This question traces the operation of a simple heap manager as in the lab P0. Assume a 32-bit machine. Heap blocks have an 8-byte header and no footer. The heap is 256 bytes. It uses a first-fit policy and has no prologue or epilogue. Suppose the bytes of the heap are addressed by offsets starting at zero.

Consider the following sequence of malloc and free calls to allocate and release blocks in this heap. The malloc argument is a size, and in this problem the free argument identifies the freed block by its size (there is at most one block of each size).

For each malloc call, determine the byte offset returned from the malloc in the corresponding box on the left. In the boxes on the right, enter S and/or C for any split and/or coalesce operations triggered by the corresponding malloc or free. If there are multiple split or coalesce actions, enter an S or C for each. If the call triggers no split or coalesce actions, enter an N.

Finally, when you are done, enter all five malloc offsets in one line separated by commas in the box below:

8, 84, 212, 84, 180
P1. Sweet base 16 (20 points)
Consider a simple page table for a simple model machine. This machine has a byte-addressable memory with sixteen 16-byte pages/frames, so an address (virtual or physical) fits in one 8-bit byte. A page table entry (PTE) is also one byte, consisting of (in order) a zero bit (unused for this example), a valid bit (set iff the PTE has a valid translation), a write-enable bit, an execute-enable bit (these permission bits are set iff the corresponding permission is enabled) and a 4-bit page frame number (PFN).

Assume the OS kernel initializes the page table with contents as shown to the right:
Consider the five instructions listed, each with its virtual address operand. What does the machine do for each instruction? Enter into the corresponding box an E if the instruction executes, an F if it gives a page fault, and a P if it gives a protection fault. Only one outcome is possible for each instruction. List any additional assumptions you make, outside of the boxes:

Finally, when you are done, enter all five outcomes in one string with no punctuation in the box below:

FFEFP
P2. Unix process synchronization (20 points)
List all possible output strings for the following numbered pseudocode program snippets. Confine your answer to the corresponding numbered space on the left. Answer with the strings and only the strings. Use commas to separate possible output strings. Assume all prints are instantaneous (unbuffered). Assume that all of these pseudocode snippets are correctly coded with no errors and no hidden code, e.g., processes/threads exit when they finish the code as shown, and there are no other processes. Unix system calls behave as described in class. Each child is the result of the fork shown in the corresponding parent.

1. acb, cab
c before b

2. bac, abc
a comes before c

3. acdb, cadb
Tricky because listen can unblock connect, so a can precede c.
**kill this one; free credit**

4. 1122 in any order
Parent dies silently
Two children survive and fork
### P3. Thread synchronization (30 points)

List **all possible output strings**… following the instructions and assumptions as for the previous problem. All semaphores start at 0. Thread primitives behave as in lab p1t, but the scheduler is **nondeterministic** (schedules admit preemption and multicore).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a12b, 1a2b, a1b2, 1ab2, ab12</td>
</tr>
<tr>
<td></td>
<td>a before 2</td>
</tr>
</tbody>
</table>

| 2 | 1 before a, 2 before b |
|   | 1a2b, 12ab               |

| 3 | a12b, 12a |
|   | Once T2 takes the lock 12 are together. And they are before b given the wait. B deadlocks if not waiting before T2 |

| 4 | 123 are together once T2 takes lock. And the 1,2 signal is always lost. If T1 goes first then both signals lost. |
|   | a123b, 123a            |

| 5 | xy together; if before a, T1+2 deadlock |
|   | axy1b2, axyb1, 1xya, xya1, xy1a |
|   | If b before 1, then T2 deadlocks, no 2 |

| 6 | asz0, as1z, azs1 |
|   | sa1z |
|   | sza0, sza<deadlock> |
|   | zsa1, zsa <deadlock> |

---

```c
T1: thread_lock(1); printf("a"); thread_wait(1,1); printf("b");
    thread_unlock(1);
```

```c
T2: thread_lock(1); printf("1"); thread_signal(1, 1); printf("2");
    thread_unlock(1);
```

```c
T1: thread_lock(1); printf("a"); thread_wait(1,1); printf("b");
    thread_unlock(1);
```

```c
T2: thread_lock(1); printf("1"); thread_signal(1, 1); printf("2");
    thread_unlock(1);
```

```c
x = 0;
T1: thread_lock(1); printf("a");
    if (x==0)
        thread_wait(1,1);
    printf("%d", x);
    thread_unlock(1);
```

```c
T2: thread_lock(1); printf("s");
    x=1;
    thread_unlock(1);
    thread_signal(1, 1);
```

```c
T3: thread_lock(1); printf("z");
    x=0;
    thread_unlock(1);
```
P4. Raft safety properties. (30 points) These are true/false questions: please label each statement with T or F in the corresponding box. Assume Raft as described in class and in the Raft paper, and that all assumptions of Raft are true.

1. Every log entry with the same index and term has the same action, across all replicas.  
2. If two replicas agree on an entry (index, term, action) then they agree on all prior entries.  
3. If two replicas apply an entry at index \( i \) to the state machine then they both agree on that entry.  
4. At most one replica runs in Leader mode with term \( T \), for any value \( T \).  
5. At most one replica runs in Candidate mode with term \( T \), for any value \( T \).  
6. If an entry is committed, then it is replicated on a majority of replicas.  
7. If an entry is replicated on a majority of replicas, then it is committed.  
8. If an entry from a Leader’s current term is replicated on a majority of replicas, then it is committed.  
9. If a Candidate receives a majority of votes in an election then its log contains all committed entries.  
10. A Follower may have a log entry with a higher term than its current Leader’s current term.  
11. If a Follower has an entry with its current Leader’s current term, then it has no entries with a higher term.  
12. If an entry was present in a replica’s log at the time of its election as Leader, then that replica does not overwrite the entry as a Leader.  
13. A replica can be elected Leader in term \( T \) even if it receives a RequestVote response from a replica with a higher term.  
14. After a replica declares as Candidate for a term \( T \) it may become a Leader for term \( T \).  
15. After a replica declares as Candidate for a term \( T \) it may become a Follower for term \( T \).
P5. Threads in the P3 Raft implementation. (15 points)
Consider a Raft RSM system as implemented for lab P3 with N=3 replicas. Consider the following code states for a replica. The question asks how many threads exist within a replica for each state. For each code state, enter in the corresponding box a pair of numbers (separate by a comma) stating how many threads are ready/running and how many threads are blocked. In the space below each question, list the objects that the blocked threads are blocked on, i.e., what are they waiting for? You may assume that a single thread handles incoming RPC calls.

1. Leader has sent a round of AppendEntries RPC, but has received no responses yet.
   Timer thread blocked on heartbeat timer, RPC client threads waiting for two followers to respond, one RPC server thread waiting for incoming call.
   0, 4
   Timer thread blocked on heartbeat timer, RPC client threads waiting for two followers to respond, one RPC server thread waiting for incoming call.

2. A Follower has received a RequestVotes RPC, but has not responded yet.
   RPC server thread ready/running, timer thread blocked on leader timeout.
   1, 1
   RPC server thread ready/running, timer thread blocked on leader timeout.

3. A Candidate is about to restart a failed election.
   Election timer thread running, 0-2 RPC client threads waiting for responses, 1 RPC server thread waiting for incoming.
   1, 1-3
   Election timer thread running, 0-2 RPC client threads waiting for responses, 1 RPC server thread waiting for incoming.

P6. A touch of crypto (30 points)
This questions asks how crypto primitives are used in HTTPS (or other secure networking based on SSL/TLS) as described in class. For each listed action, enter into the corresponding box a C if an HTTPS client performs the action, an S if an HTTPS server performs the action, and/or an A if a Certifying Authority performs the action. If multiple parties perform the action then please enter them in the order C-S-A, e.g., “SA”. If none of them do, then enter an N.

1. Encrypt under a private key (e.g., RSA)
   A
   Encrypt under a private key (e.g., RSA)

2. Encrypt under a public key (e.g., RSA)
   C
   Encrypt under a public key (e.g., RSA)

3. Encrypt under a symmetric secret key (e.g., DES)
   CS
   Encrypt under a symmetric secret key (e.g., DES)

4. Compute a secure hash (e.g., with SHA*)
   A
   Compute a secure hash (e.g., with SHA*)

5. Transmit a public key to another party
   SA
   Transmit a public key to another party
Note on Threads in Raft implementation

I was curious to know if students really understand the thread structure of the software they spend so much time writing and debugging in the Raft lab. Life is too short to spend time debugging software whose thread structure you do not understand. It goes much faster if you spend the time up front to understand it. And the job is more interesting.

I got lots of responses that referred to the “leader thread” or “candidate thread” or the “main thread”. There is no such thing. Some of these responses might have been directed at all replicas together on the presumption that each replica runs a single thread. But that’s not how it works. Anyway, the question asked about the threads in a single specific replica at a specific point in the code.

The Raft lab is a nice example of modern multi-threaded event-driven structure. Each Raft replica is both an RPC (RMI) client and a server. An RPC server maintains a pool of RPC server threads. Each incoming RPC call from another replica (AppendEntries, RequestVote) runs on one of these RPC server threads. If an RPC server thread has no incoming call to execute, it blocks awaiting an incoming call. On the client side, when a replica initiates a round of outgoing AppendEntries or RequestVote RPC calls, it creates one RPC client thread per peer replica that issues the call to that replica and blocks waiting for the response. This structure is common for RPC clients and servers. These concepts are discussed in the slides on RPC.

In addition, each replica maintains one or more Java timers that prompt it to take some action at some future time. Various other exam questions have tested your understanding of these timers. Each timer has a dedicated thread. The timer thread blocks waiting for the timer to fire. When it fires, the timer thread wakes up and calls the registered timer handler.

So: when a replica is “at rest”, it has a timer thread and one or more RPC server threads waiting for something to happen. Once a replica is initialized, all of its Raft logic runs on one of these threads to respond to events—incoming calls and timer firings. That’s all that ever happens in Raft and in many networked services.

Note on A touch of crypto

A certifying authority issues certificates under its digital signature. To sign, A computes a secure hash (#4) over a byte string (e.g., the certificate), encrypts the hash with its private key (#1), and appends the encrypted hash to the string. Anyone who knows A’s public key can validate that the certificate is authentic. A web/SSL certificate contains the public key (or public key hash) of its subject (#5). An HTTPS/TLS/SSL server (S) authenticates to a client by presenting its certificate and public key to the client (#5). As described, the client C validates the certificate, generates a secret symmetric key, and encrypts the symmetric key under the public key of S (#2). Only S can decrypt the symmetric key because only S knows the private key of S. Once both C and S know the symmetric key, they use it to encrypt/decrypt any data they send over the connection (#3).
P7. A little synchronization (35 points)

This problem asks you to implement a simplified thread-safe pipe API with two calls: `read(int n)` and `write(int n)`. The `write` call produces a sequence of \( n \) bytes into the pipe inlet; `read` consumes the next \( n \) bytes in order from the pipe outlet. Each `write` (or `read`) must complete before the next `write` (or `read`) is permitted to begin.

The buffer is bounded: the pipe holds at most \( N \) bytes. `Read` sleeps if the buffer is empty, and `write` sleeps if the buffer is full. A `read` or `write` with \( n > N \) is legal, but it will always block the caller at least once.

“Any kind of pseudocode is fine as long as its meaning is clear.” Ignore the details of the data transfer and buffer layout in memory: just say “copy \( x \) bytes” (for some \( x \)) in place of the data transfer code.

```c
read(int n)
{
    readerMx.lock();
    pipeMx.lock();
    for (i = 0; i++ < n) {
        while (no bytes in pipe)
            dataCv.wait();
        move one byte from pipe into buf[i];
        spaceCV.signal();
    }
    pipeMx.unlock();
    readerMx.unlock();
}
```

```c
write(int n)
{
    writerMx.lock();
    pipeMx.lock();
    for (i = 0; i++ < n) {
        while (no bytes in pipe)
            spaceCv.wait();
        move one byte from pipe into buf[i];
        dataCV.signal();
    }
    pipeMx.unlock();
    writerMx.unlock();
}
```

Soda machine with byte streams! This problem is the “exception that proves the rule” about nested locks. You need one lock here. (7 pts) Note also the usual use of two conditions in a cross-wise configuration. The problem is discussed as an example in the class slides. (end of unix-tree.pptx in 18s)