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 only 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:

```
```
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.

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:

<table>
<thead>
<tr>
<th>0</th>
<th>V</th>
<th>W</th>
<th>X</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page table
0: 0x00
0x68
0x69
0x6a
0x2b
0x2c
0x66
7: 0x47
0x41
0x20
0x2c
0x12
0x53
0x54
0x55
f: 0x1c
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

Parent: ...fork();...
    printf("a");
    pid = wait(...);
    printf("b")

Child: printf("c");

2

Parent: pipe(fd);
    ...fork()..;
    printf("a");
    write(fd[1],...); /* fd[1] = in */

Child: printf("b");
    read(fd[0],...); /* fd[0] = out*/
    printf("c");

3

Client: s = socket(...);
    connect(s, <name>);
    printf("a");
    read(s, ...);
    printf("b");

Server: s = socket(...);
    bind(s, <name>);
    listen(s);
    printf("c");
    s1 = accept();
    printf("d");
    write(s1, ...);
    printf("e");

4

Parent:
    int i = 1;
    if (fork() != 0) {
        i = i + 1;
        if (fork() != 0)
            exit(0);
    }
    fork();
    printf("%d", i);
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).
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.

2. A Follower has received a RequestVotes RPC, but has not responded yet.

3. A Candidate is about to restart a failed election.

P6. A touch of crypto (30 points)

This question 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)
2. Encrypt under a public key (e.g., RSA)
3. Encrypt under a symmetric secret key (e.g., DES)
4. Compute a secure hash (e.g., with SHA*)
5. Transmit a public key to another party
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) {
}

write(int n) {
}
```