Consider a Raft RSM Consensus configuration with 5 servers. For each failure property listed below, state the minimum number of simultaneous node failures that are necessary for the condition to occur in a scenario involving a combination of failures. There are two cases. **Case 1:** no partition occurs in the scenario. **Case 2:** a network partition occurs as part of the scenario. Your answer for each property/case is an integer between 1 and 5, or 0 if the property cannot occur under any failure scenario.

**Assume:**
Servers maintain their logs and state machines (application state) in memory only, and lose all of this state on a failure. All failures are transient. All node failures are fail-stop. At most one network partition occurs in any scenario: one majority side and one minority side. Response time for a reachable server is variable but bounded by L seconds.

### What is the minimum number of simultaneous node failures needed for each failure property to occur?

<table>
<thead>
<tr>
<th>Failure property</th>
<th>no partition</th>
<th>with partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Service becomes entirely unavailable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Loss of uncommitted operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Loss of operations that committed L seconds ago or less</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Loss of operations that committed more than L seconds ago</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Total loss of all application state</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
P1. Schedules (30 points)

List all possible outputs for the following pseudocode program snippets. Omit any quotes in your outputs. All prints are instantaneous. For the semaphore examples, presume that threads T1 and T2 start concurrently, all semaphores start at 0, p() is “down”, and v() is “up”. For the Unix examples, fork() is the classic Unix fork primitive as described in class.

T1: s1.p(); print("a"); s2.v(); print("b");
T2: print("1"); s1.v(); s2.p(); print("2");

T1: s1.p(); print("a"); s2.v(); print("b");
T2: print("1"); s2.p(); s1.v(); print("2");

T1: s1.p(); print("a"); print("b"); s2.v(); s1.v();
T2: s1.v(); s1.p(); print("1"); print("2"); s2.p();

fork();
printf("a");
fork();
printf("b");

if (fork() == 0) {
    printf("a");
} else {
    fork();
    printf("b");
}
P2. Read! (40 points)

The following pseudocode outlines consist of sequences of classic Unix system calls issued by one or two processes. Presume that the code for each system call is well-formed and executes correctly according to its usual behavior. The outlines culminate in read() system calls. Does each read() return data, an EOF (end-of-file / no data available), an error, or does it block? There might be multiple possible outcomes, depending on other events or states that are not specified. **Circle all that apply.** Presume that each read() supplies a valid buffer and is otherwise well-formed.

| fd1 = open("file1", O_RDWR, ...); dup2(fd1, fd2); /* dup fd1 onto fd2 */; read(fd1, ...); | Parent: pipe(fd); /* fd[0] = out, fd[1] = in */ fork(); read(fd[1], ...); Child: read(fd[0], ...); |
|---|---|---|
| data EOF error block | data EOF error block | data EOF error block |
| read(fd2, ...); | read(fd[1], ...); | Child: read(fd[0], ...); |
| data EOF error block | data EOF error block | data EOF error block |
| fd = open("file1", O_RDWR, ...); dup2(fd, 0); exec="/bin/cat", ...); .... read(0, ...); | Parent: pipe(fd); /* fd[0] = out, fd[1] = in */ fork(); read(fd[0], ...); close(fd[1]); Child: read(fd[0], ...); |
|---|---|---|
| data EOF error block | data EOF error block | data EOF error block |
| s = socket(...); bind(s, <server address>) listen(s, ...); read(s, ...); | Parent: pipe(fd); /* fd[0] = out, fd[1] = in */ fork(); close(fd[1]); fork(); |
|---|---|---|
| s = socket(...); connect(s, <server address>); read(s, ...); | Parent: pipe(fd); /* fd[0] = out, fd[1] = in */ fork(); close(fd[1]); |
|---|---|---|
| data EOF error block | Child: read(fd[0]); | data EOF error block |
P3. Raft protocol and server behavior. (40 points)
In which modes does a Raft server take the following actions? Circle all that apply.

1. Send an AppendEntries RPC.
2. Send a RequestVote RPC.
3. Send a ResetEntries RPC.
4. Reply to an AppendEntries RPC.
5. Reply to a RequestVote RPC.
6. Reply to a ResetEntries RPC.
7. Write an entry to the local log.
8. Commit (mark as committed) an entry in the local log.
9. Overwrite a committed entry in the local log.
10. Overwrite an uncommitted entry in the local log.
11. Apply an entry to the local state machine.
12. Respond with current term to a message from a peer with an earlier term.
13. Transition to a different mode upon receiving a message from a peer with a later term.
14. Transition to a different mode as a result of a timer firing.
15. Send a message as a result of a timer firing.
16. Roll back the current term to a prior term.
17. Increment the current term by one.
18. Increase the current term value by more than one.
19. Set a timer upon (in connection with) receiving an RPC message.
20. Set a timer upon (in connection with) sending an RPC message.
Consider the following C code, which is divided into two separate C source files as shown. Assume that it builds and runs as a process without errors, e.g., header file #includes etc. that are necessary for correct builds are omitted. Answer the questions below. Suppose that the machine is a 32-bit machine in which a virtual address has a 20-bit VPN (e.g., it is a standard IA32).

**Answers**

1. Draw (or list) the stack contents at the point of the `return` statement in `fill()`, in the box above.

2. How many external symbol references does the linker resolve to link it? List them in this box →

3. How many of these symbols are from the C standard library?

4. How large is a virtual page on this machine (in bytes)?

5. How many system call traps does it take? List the system calls that it issues in this box. --------

6. How much memory space (in pages) does this program use on its stack when it runs?

7. How much memory space (in pages) does it use in its global data segment?

8. How much memory space (in pages) does it use in its code (text) segment?

For 9-10, assume the OS allocates page frames on demand and does not evict. Ignore VM activity in the library.

9. How many page faults does the process take?

10. How many page table entries (PTEs) have valid translations when the process exits? (Assume no page evictions.)