CPS 310 midterm exam #2, 4/9/2018

Your name please: _____________________________  NetID:__________

Sign for your honor: _____________________________

Answer all questions as directed in the spaces provided. If you answer outside of the spaces provided you will lose points. Additional marks may or may not be considered for partial credit. You get 10 points of credit just for writing your name and NetID and filling out the exam legibly without having to use any extra/replacement pages. You have 75 minutes plus standard grace.

P0. Raft failures (30 points)

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>with partition</th>
<th>no 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>
All p0 questions presume an RSM replica group with N=5 nodes/replicas that store their operation logs and replicated app state in memory only. A failed node stops, discards its local logs and its copy of the app state, and optionally recovers. You are asked for the minimum number of simultaneous node failures (fail stop) needed to cause various failure and loss conditions for the service as a whole, with and without a network partition. “Loss” means loss by the entire service. Normally a restarting RSM replica can recover its lost state from another replica. The question is: what is the minimum number of failures that makes this recovery impossible?

Since the question asks for a minimum in any qualifying scenario, we should suppose that any partition is a 3/2 partition as shown in the figure. That is the worst case because it has a bare and fragile majority.

Some students are surprised by these questions and respond: never! RSM cannot lose app state or committed operations! But RSM is based on assumptions and I am asking what happens when the assumptions are violated.

**Service becomes entirely unavailable**

**Available** means that the service can execute operations on behalf of clients. A Raft service can execute operations only if a majority of nodes are up and connected to one another. Without a partition, 3 node failures are necessary and sufficient: the 2 survivors cannot make a majority. With a 3/2 partition it is sufficient for any 1 of the 3 nodes on the majority side to fail, so that neither side has a majority.

**Loss of uncommitted operations**

An uncommitted operation could be stored on a single replica, i.e., the leader. Maybe the leader just received the request from the client and has not transmitted it to the followers yet. Thus a single failure is necessary and sufficient to lose an uncommitted operation, with or without a partition.

**Loss of operations that committed L seconds ago or less**

Operations must be stored on at least 3 nodes to commit. If they committed recently, they might not have propagated to the other nodes yet, even if there is no partition. So failure of the minimum 3 nodes is necessary and sufficient to lose these updates. I accepted 5 in the non-partition case because there is some ambiguity and room for argument about it.

**Loss of operations that committed more than L seconds ago**

By assumption these are replicated on 5 nodes in the non-partition case (barring failures) and possibly on three nodes in the partition case. Any scenario must have all 5 nodes failing, or 3 nodes in the partition case.

**Total loss of all application state**

Barring failures and partitions, every replica keeps a copy of the application state and the log of operations that produced that state. All five replicas must fail to lose all copies of the log. I accepted 3 for the partition case because the partition could have been present since the system started, in which case only 3 replicas would store application state.
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.

1ab2, 1a2b

“1” must print first because T1 does not pass s1 until T2 prints the “1”. Then T2 cannot pass s2 until after T1 prints “a”. After the leading “1a” the last char from each of T1 and T2 could be in either order.

1

Then deadlock: T1 can’t pass s1 and T2 can’t pass s2.

12, ab12

“12” is possible because T2 may consume its own s1.v(). “ab12” occurs when T1 consumes the s1.v() in T2. Note: once T2 prints “1” T1 can never unblock from s1.p(). If T1 prints “a” then it must be followed by a “1” because T1’s s1.v() wakes T2’s s1.p().

aabbabb

Must be 2a and 4b with an ordering constraint: must start with an “a” and end with two “b”. The second “a” can occupy any position subject to that constraint.

ababbb

abbabb

a

b

bab

bba

Must be 1a and 2b in any order. Note that fork() returns 0 to the child and non-zero to the parent. So only the child prints the “a”, but then the parent forks again and both parent and new child print “b”.

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, ...);
read(fd2, ...);

Parent:
pipe(fd); /* fd[0] = out, fd[1] = in */
fork();
read(fd[1], ...);

Child:
read(fd[0], ...);
```

```
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[1], ...);
close(fd[1]);

Child:
read(fd[0], ...);
```

```
data  EOF  error  block
```

```
s = socket(...);
bind(s, <server address>)
listen(s, ...);
read(s, ...);
```

```
Parent:
pipe(fd); /* fd[0] = out, fd[1] = in */
fork();
read(fd[0], ...);
close(fd[1]);

Child:
read(fd[0]);
```

```
data  EOF  error  block
```

```
s = socket(...);
connect(s, <server address>);
read(s, ...);
```

```
Parent:
pipe(fd); /* fd[0] = out, fd[1] = in */
close(fd[1]);
fork();
```

```
Child:
read(fd[0]);
```

```
data  EOF  error  block
```
Notes on Read! and file descriptors

This problem deals with the Unix process model and I/O model and how they interact. The Unix process model leads to a hierarchy of processes (via fork) that can execute programs (via exec*). Processes hold references on I/O objects (file, socket, pipe) named by numbered I/O descriptors—including but not limited to stdin 0, stdout 1, and stderr 2—that store and transmit byte streams via read and write. The pipe and socket calls create a pipe or socket and return a reference to it as a descriptor; pipe returns two descriptors, one to write into the pipe and one to read from the pipe. Processes also acquire I/O references by inheritance (on fork) and by cloning (via dup*). Processes release their references by close on the descriptor, or implicitly on exit.

For sockets, various calls (connect, bind, listen, and accept) establish connections and prepare socket descriptors for I/O: a process may initiate a connection as a client with connect, or accept an incoming connection as a server. An established connection is a bidirectional data channel between two sockets (a socketpair) and the processes that hold references to them. These processes may reside on different nodes and may use the sockets to communicate over a network with (e.g.) TCP/IP protocols.

If a read can return data, then it can also block waiting for that data to arrive (e.g., from a disk, from the network, from a pipe). And it can return an EOF when the data is gone. Read returns EOF when a process reads past the end of a file. For a pipe or socket the reader receives an EOF only after all potential writers release their references to the object, and the reader has consumed all of the data that has been written.

This problem tests your understanding of that whole structure. Second guessing made it harder. The pseudocode sequences are "complete" for the listed processes, but I left some ambiguity about other events that could occur that are not shown, so I had to accept a wider range of answers for behaviors that could result if more code was added to the example. (Grey boxes are optional.) But if your answer relied on hidden code, then I required you to list all of the cases for what that code could do, and what would happen without it. As noted at the exam, you also had to assume that the fork pseudo-calls have suitable conditionals to separate the code paths for the parent and child.

open and dup

The first two should yield the same behavior, since after a dup2(fd1, fd2) both fd1 and fd2 refer to the same underlying I/O object, an open file. The reads return data if the file is non-empty, and EOF after the data has been read. And they might block waiting for I/O, e.g., disk.

exec*

Some were confused by this because exec “does not return”. But the calling process continues after a successful exec—it just runs a different program, but it has the same I/O descriptors as before. This example shows the usual expected behavior for a cat: read from stdin. And stdin was bound to a file by the earlier open, so it has the same behavior as above.
More notes on Read! and file descriptors

socket/bind/listen/accept (server)

This is an error because there is no accept. A bound server socket is suitable for accept, but not for I/O (e.g., read). The accept returns a new socket that is connected to a particular client and is suitable for I/O.

socket/connect (client)

Blocks waiting for the server to accept and/or send data. In an RPC or web transaction it is typical for the client to send data (e.g., a GET and URL) after the connect succeeds, but before reading. A server generally does not send data into a connected socket without first receiving a request. In this case the read blocks forever. If the server closes its end of the connection, the reader receives an EOF. There is no rule that prohibits the server from sending unsolicited data, in which case the read receives data possibly followed by EOF.

Pipe 1

Parent. This is an error because the read is on the write end of the pipe, i.e., the descriptor for sending data “in” to the pipe, which is not valid to read data “out” of the pipe. Pipes are unidirectional.

Child. The read blocks because no data was written into the pipe, and both parent and child still have the write end open. If some hidden piece of code in either process writes data, then the reads could return data, but if there could be a hidden write then there could be hidden closes yielding an EOF.

Pipe 2

Both reads block because no data was written into the pipe, but both parent and child still have the write side open. If some hidden piece of code in either process writes data, then a read could return data and EOF.

Pipe 3

This is an EOF: the writer has closed the write end of the pipe before the fork, so neither process can write to the pipe. The pipe is empty and has no writer, so the read returns EOF.
### P3. Raft protocol and server behavior. (40 points)

In which modes does a Raft server take the following actions? Circle all that apply.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Send an AppendEntries RPC.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>2.</td>
<td>Send a RequestVote RPC.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>3.</td>
<td>Send a ResetEntries RPC.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>4.</td>
<td>Reply to an AppendEntries RPC.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>5.</td>
<td>Reply to a RequestVote RPC.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>6.</td>
<td>Reply to a ResetEntries RPC.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>7.</td>
<td>Write an entry to the local log.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>8.</td>
<td>Commit (mark as committed) an entry in the local log.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>9.</td>
<td>Overwrite a committed entry in the local log.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>10.</td>
<td>Overwrite an uncommitted entry in the local log.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>11.</td>
<td>Apply an entry to the local state machine.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>12.</td>
<td>Respond with current term to a message from a peer with an earlier term.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>13.</td>
<td>Transition to a different mode upon receiving a message from a peer with a later term.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>14.</td>
<td>Transition to a different mode as a result of a timer firing.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>15.</td>
<td>Send a message as a result of a timer firing.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>16.</td>
<td>Roll back the current term to a prior term.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>17.</td>
<td>Increment the current term by one.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>18.</td>
<td>Increase the current term value by more than one.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>19.</td>
<td>Set a timer upon (in connection with) receiving an RPC message.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
<tr>
<td>20.</td>
<td>Set a timer upon (in connection with) sending an RPC message.</td>
<td>leader</td>
<td>follower</td>
<td>candidate</td>
</tr>
</tbody>
</table>
Raft modes

In the Raft protocol for RSM consensus, each node/replica is in exactly one of three possible modes (LCF) at any given time. The mode determines how the node responds to incoming RPCs (AppendEntries, RequestVote) from peer nodes. A Leader sends periodic AppendEntries calls clocked by a timer. A Follower who does not receive an AppendEntries from a leader becomes a Candidate after a timeout. A Candidate increments its term and sends a RequestVote RPC, then waits for a specified time for a quorum of votes to become Leader. A Candidate or Leader steps down to Follower if it receives a RequestVote or AppendEntries with a higher term. Followers vote for at most one Candidate in any given term. A Follower follows its Leader until a timeout, or until a new Leader or Candidate appears with a higher term. This question tests your understanding of the Raft modes and their behavior and transitions between modes.

1. AppendEntries RPC is sent by a Leader only.
2. RequestVote RPC is sent by a Candidate only.
3. There is no ResetEntries RPC in Raft.
4. A node may receive an AppendEntries RPC in any mode.
5. A node may receive a RequestVote RPC in any mode.
6. There is no ResetEntries RPC in Raft.
7. A Leader logs new entries to its local log and sends AppendEntries to Followers to command them to log those entries too.
8. A Leader commits new entries in its local log after receiving a quorum, then it notifies its Followers to commit those entries too.
9. A Raft node never overwrites committed entries with different entries, in any mode.
10. A Leader may command a recovering Follower to overwrite an uncommitted entry in its local log with a more recent entry.
11. Leaders and Followers apply committed entries to the local state machine after learning of their commitment.
12. Any node in any mode responds to any message with a stale term (earlier term than its own) by notifying the sender of its term.
13. A Leader or Candidate transitions to Follower upon learning it is stale, i.e., by receiving a message from a peer with a higher term.
14. A Follower transitions to Candidate if it has not heard from a Leader for a timeout interval.
15. A Leader sends an AppendEntries clocked on its heartbeat timer. A Candidate sends RequestVotes after a timeout with no Leader.
16. Terms increase monotonically: no Raft node ever rewinds its term.
17. A Candidate increments its current term by 1 before starting an election for the new term by sending a RequestVotes RPC.
18. Any Raft node updates its term to the highest that it learns of, which may be much higher than its current term.
19. A Follower resets a timer after hearing an AppendEntries RPC from a Leader; it expects another RPC before the timer fires.
20. A Leader resets a timer before sending an AppendEntries RPC: it sends another RPC before the timer fires. A Candidate resets a timer before sending a RequestVote RPC: it checks and/or completes the election when the timer fires.
P4. Third time’s a charm (50 points)

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).

**main.c**
```c
int size = 8;
int main()
{
    char buffer[8];
    fill(buffer);
    printf("%s\n", buffer);
}
```

**fill.c**
```c
void fill(char* buf)
{
    int len = size;
    strncpy(buf, "yikes!", len);
    return;
}
```

**stack**
```c
fill
len (4 bytes)
saved RA, FP (8 bytes)
buf (4 bytes)
main
buffer (8 bytes)
saved RA, FP (8 bytes)
```

**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.)
C environment and the process virtual address space

A program runs as a process with a virtual address space that consists of a set of segments, each segment occupying a disjoint region of virtual address space. Segments may have different access permissions (e.g., write-enable, no-execute). The hardware represents memory permissions and residency in physical memory at the granularity of fixed-size pages. Therefore, each segment is an integral number of virtual pages and each valid virtual page is part of exactly one segment. Specifically, the OS maintains a page table in memory with a page table entry (PTE) for each virtual page number (VPN) in the process virtual address space. The PTE provides the corresponding page frame number (PFN) for the page—if the page is resident and a translation is available—and a few bits that represent the permissions allowed for accesses to the page. The page table is used by the OS and machine and is not visible to the process itself.

C code is compiled and linked to form a complete executable program. A program file contains initial data for its text and static/global data segments, and any constraints on their virtual address locations at runtime. The linker creates the program file by examining object files that result from compiling the source files of the program, together with any libraries/archives that the program uses. The linker extracts and gathers up text for all procedures in the program, and initial values for any initialized global data, and writes these into sections of the program file. The kernel uses this data to set up the process virtual address space for the program at exec* time.

An external symbol is any string name the programmer uses for a procedure or global data item that can be referenced by its name across multiple source or object files. Each such symbol is declared in a C header file (so that many source files can reference the symbol) and defined in exactly one source or object file (so that the linker knows where to extract it from). The linker resolves references to named symbols by searching the symbol tables of the object files and libraries for a definition at link time, and extracting the defined value. An object file’s symbol table also identifies any symbols that the object references, and the instructions that reference those symbols.

This question probes your understanding of these concepts.

**Stack.** Each procedure call pushes a frame on the stack. The standard instruction sequence for a procedure call pushes any arguments on the stack, followed by a return address (RA) and (optionally) a frame pointer and other saved registers, followed by local variables for the called procedure. In this example main() calls fill() with a specific argument and specific local variables. They should show up on the stack in roughly the right place. This should be familiar from lab p2.

**External symbol references.** The procedure fill is an external text symbol defined in fill.c and referenced from main.c. The size variable is an external data symbol defined in main.c and referenced from fill.c. Printf and strncpy are symbols for well-known procedures in the standard library, familiar at least from lab p2. The program also uses main and exit although they are not referenced in this source code, and write for the printf. Write and exit are the only system call traps.
More on the C environment and the process virtual address space

Virtual page size

Memory is byte-addressable. A 32-bit virtual address has a 20-bit VPN. That means there are 12 bits left to represent a byte offset in the virtual page for the addressed byte. A 12-bit offset can address $2^{12} = 4096$ bytes = 4KB.

Memory space in each segment

The program needs at least text, data, and stack segments to run. It has a global variable size (data) and a little bit of text (main and fill + library routines) and it does not use its heap, so we can ignore that. Each segment must have at least a page, the smallest unit of memory the page table can represent in a segment. None of the segments requires any more than a page: they all have well under 4KB of data. Main and fill require at most a few dozen instructions and bytes of stack space, and size is only four bytes. It is reasonable to say that they need only one page per segment.

Page faults and PTEs

By assumption the OS allocates physical memory only as needed, on demand in response to page faults as the process references its pages. Then it must take one page fault to load the page for each of stack, data, text. If it does not evict a page that is in use (i.e., it keeps every loaded page until its process exits) then these three pages all have valid translations when the process exits.
Grades

Here is the Gradescope distribution out of 190 points. (10 more free for signing your name).

The scores were a little disappointing. The median is about 15 points lower than typical for my midterm exams. In large measure this reflects the limited partial credit on may questions. It is more work to give partial credit with Gradescope.

Some students said they were surprised by the Raft questions (60/180 points, not counting the 10 points that were ungraded and given free for the last five Raft mode questions). But it was in scope and these are relatively easy questions for anyone who understands the basics of how Raft works, as required for the lab p3. Nobody should have been surprised by “Third Time’s a Charm” because I asked a variant of this same question on both midterms the last time I taught the course (17s). These questions should be easy and there is no avoiding this material. I also heard that some expected more emphasis on lab p2 (buggyserver), like the second midterm of 17s, but I only asked about the stack and how socket descriptors behave. The lesson here is that it is a gamble to try to predict what will be on the exam: if you take shortcuts you might get unlucky.

People did pretty well on the Schedules, but some still aren’t considering non-determinism and/or aren’t reasoning well about all possible orderings.