Computer Science 162, Fall 2014
David Culler
University of California, Berkeley
Midterm 3
December 15, 2014

Name	
SID	
Login	
TA Name	
Section Time	

This is a closed book exam with one 2-sided page of notes permitted. It is intended to be a 2 hour exam. You have 3 hours to complete it. The number at the beginning of each question indicates the points for that question. Write all of your answers directly on this paper. Make your answers as concise as possible. If there is something in a question that you believe is open to interpretation, please raise your hand to request clarification. When told to open the exam, put your login on every page and check that you have them all. (Final page is for reference.)

By my signature below, I swear that this exam is my own work. I have not obtained answers or partial answers from anyone. Furthermore, if I am taking the exam early, I promise not to discuss it with anyone prior to completion of the regular exam, and otherwise I have not discussed it with anyone who took the early alternate exam.

X_______________________________________

Grade Table (for instructor use only)

Question	Points	Score
1	25	
2	25	
3	25	
4	25	
5	0	
Total:	100	

1. (25 points) **Operating Systems Concepts**

   (a) (18 points) Choose **either** true or false for the below questions. You do not need to provide justifications.

      i. (2 points) Digital certificates bind a host's identity with its public key.
         ◯ True
         ◯ False

      ii. (2 points) Asymmetric cryptography is much faster than symmetric cryptography.
         ◯ True
         ◯ False

      iii. (2 points) A transaction takes the state of a storage system from one consistent state to another.
         ◯ True
         ◯ false

      iv. (2 points) Only the commit entry in the redo log for a transaction need to be idempotent.
         ◯ True
         ◯ False

      v. (2 points) A shared log used by multiple processors to construct transactions can be cached if the entries are in non-volatile storage with sequential consistency.
         ◯ True
         ◯ False

      vi. (2 points) When the accept function on a TCP socket returns it is guaranteed that both the connect on the client and the listen on the server have occured and the initial sequence number is determined.
         ◯ True
         ◯ False

      vii. (2 points) In the MapReduce framework, the Reducers receiver their chunks of map results from each Mapper only after the Map phase has completed.
         ◯ True
         ◯ False

      viii. (2 points) According to the end-to-end principle, reliable network links allow the host end-points to optimize how they handle communication because they can assume other layers will handle loss.
         ◯ True
         ◯ False

      ix. (2 points) Recursive queries in a key-value store allow the directory to serialize puts and gets and provide consistency even with replication.
         ◯ True
         ◯ False

   (b) (2 points) Suppose a TCP connection is able to sustain an average window size of 10 KB with an RTT of 20ms and a maximum packet size of 1000 B. Calculate the average bandwidth over the connection.

(c) (5 points) Which of the following is true? Select **all** the choices that apply.

◯ Deadlocked threads will experience starvation.

◯ If there are deadlocked threads the system will eventually go idle.

◯ Imposing ordering on the acquisition of multiple resources will prevent deadlock.

◯ Deadlock can happen when multiple processes contend for a single resource.

◯ Preempting lock holders can eliminate deadlock but give rise to possible livelock.

2. (25 points) **Networked File System**

The following provides a simple C library that provides functions to read the contents of a local file into a dynamically allocated object and to write an object out to a file. Signatures are in the associated include file. These are the procedures that we are going to run remotely as part of the RPC that implements our networked file access. A simple main that uses this to cat a local file is in the code appendix at the end of this problem. (We are going to give you C code for precision, but your solution can be roughly C pseudocode.)

```
/* Simple file library */
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

#include "flib.h"

size_t readfile(char **fdata, char *filepath){
  size_t fsize = 0;
  char *fbuf = NULL;
  FILE *fp = fopen(filepath,"r");
  if (fp) {                     /* figure out the size */
    fseek(fp, 0L, SEEK_END);    /* seek to end */
    fsize = ftell(fp);          /* get position */
    fseek(fp, 0L, SEEK_SET);    /* seek to start */
    fbuf = malloc(fsize);
    fread(fbuf, sizeof(char), fsize, fp);
  }
  *fdata = fbuf;
  return fsize;
}

ssize_t writefile(char *filepath, char *fdata, size_t dsize) {
  FILE *fp = fopen(filepath, "w");
  if (fp == NULL) return -1;
  return fwrite(fdata, sizeof(char), dsize, fp);
}
```

You are going to build a simple user-level network file system using a remote procedure call mechanism that you will create. Basically, it will read and write files on a remote machine using the operations in the library above as the "remote procedures".

The following shows a simple program where the remote service is passed on the command line and the client rpc mechanism (in cflib.c) is initialized and then used — just as in the local case — to read the file which is on the remote server.

```
#include "cflib.h"

int main(int argc, char *argv[])
{
  char *fdata;
```

```
  if (argc < 4) {
    fprintf(stderr,"usage %s filename hostname port\n", argv[0]);
    exit(0);
  }

  char *filename = argv[1];
  crpc_init(argv[2], atoi(argv[3]));         /* initialize the RPC system */
  size_t fsize = readfile(&fdata, filename); /* read remote file as if local */
  fwrite(fdata, sizeof(char), fsize, stdout);
  return 0;
}
```

To get you started here is a skeleton of the client RPC library, cflib.c. You can assume that `socket_connect` sets up all the addressing, creates the socket, and connects it to the host and port as a TCP stream.

```
static char *host;
static int port;

void crpc_init(char *hostname, int portno) {
  host = hostname;
  port = portno;
}

int socket_connect(char *hostname, int portno);

size_t readfile(char **fdata, char *filepath) {

  int sockfd = socket_connect(host, port);  /* Create and connect the socket to server */

  /* Implement me */

  close(sockfd);
  return ____________
}

ssize_t writefile(char *filepath, char *fdata, size_t dsize) {
  /* write file function - we won't write this*/
  return 0;
}
```

Of course, a networked client needs a server. The appendix has a skeleton for your RPC server, which contains the basic single-threaded socket protocol. It has a little driver so you can see how you might start it up. All the ugly address stuff is there completeness. It calls the following.

```
void serveconnection(int newsockfd) {
/* you get to implement me */
}
```

(a) (9 points) **Design the simpleNFS RPC protocol**

Your protocol is to support two operations on remote files, readfile and writefile. *Assume both the client and server are 32 bit x86 machines running Ubuntu 14.04*

i. (4 points) Draw a diagram and describe how you would write a protocol to handle these actions end to end.

ii. (5 points) Design an (effecient) format for the messages between server and client

(b) (4 points) Implement the simpleNFS RPC protocol for the client read operation in cflib.c, as per the skeleton.

We are not interested in getting the syntax exactly right, but that you have all of the basic parts in a simple, clear implementation of the design above

```
size_t readfile(char **fdata, char *filepath) {
 /* declare what you need */
  char* send_buf;

  char* fdata;

  int send_buf_size = 0;

  int path_size = 0;

  long file_size = 0;

  -------------------------------------------------
```

```
  ------------------------------------------------

  -------------------------------------------------

  ------------------------------------------------

  int sockfd = socket_connect(host, port);  /* Create and connect the socket to server */

  ---------------------------------------

  ---------------------------------------

  ---------------------------------------

  ---------------------------------------

  ---------------------------------------

  ---------------------------------------

  write(sockfd, send_buf, send_buf_size);

  read(sockfd, _____________, _____________);

  ---------------------------------------

  ----------------------------------------

  -----------------------------------------

  -----------------------------------------

  read(sockfd, ____________, _______________);

  close(sockfd);

  return ____________
}
```

(c) (4 points) Implement the corresponding server side for the read (just check for the write and make it no-op) for the skeleton above. Assume this function is called when the server receives a socket connection from the client with the corresponding client socket file descriptor.

```
void serveconnection(int newsockfd) {
  char *fdata = NULL;
  long fsize = 0;

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

    /* invoke the file system method */
    fsize = readfile(&fdata, ________________);

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________

  __________________________________
```

```
    ----------------------------------

    ----------------------------------

    ----------------------------------

  }
```

(d) (2 points) Security Holes: What is the fundamental security flaw in this simple-NFS?

(e) (3 points) Naming: Explain in a few sentences how resolving the file pathname at the server, as in this simpleNFS, is different from resolving them at the client. How does NFS address this issue and provides a coherent file namespace at the client.

(f) (3 points) Security: Suppose that the server maintains a database of users and each file can be associated with a user. Describe how you would modify the protocol above to use a asymmetric key system to ensure that only the user associated with a file could access it remotely. How would you further extend this to maintain confidentiality of the data?

Code Appendix

```
**************************************
/* Local reader example             */
#include "flib.h"
int main(int argc, char *argv[])
{
  char *fdata;
  char *filename = argv[1];
  size_t  fsize = readfile(&fdata, filename);   /* read the file */
  fwrite(fdata, sizeof(char), fsize, stdout);
  return 0;
}

**************************************
/* Remote service wrapper           */

int rpc_server(int portno)
{
  struct sockaddr_in serv_addr;
  struct sockaddr_in cli_addr;
  uint clilen = sizeof(cli_addr);
  int sockfd, newsockfd;

  /* Create Socket to receive requests*/
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* Bind socket to port */
  memset((char *) &serv_addr,0,sizeof(serv_addr));
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port        = htons(portno);
  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    error("ERROR on binding");
  while (1) {
    /* Listen for incoming connections */
    listen(sockfd, MAXQUEUE);

    /* Accept incoming connection, obtaining a new socket for it */
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

    serveconnection(newsockfd);   /* Service the request */

    close(newsockfd);
  }
  printf("Server exiting\n");
  close(sockfd);
  return 0;
}

int main(int argc, char *argv[])
```

```
{
  int portno;
  if (argc < 2) {
    fprintf(stderr,"ERROR, no port provided\n");
    exit(1);
  }
  portno = atoi(argv[1]);
  printf("Opening server on port %d\n",portno);
  return rpc_server(portno);
}
```

3. (25 points) **Two-Phase Commit and KV Store**

This question will deal with the concept of atomic distributed consensus in the context of the 2-phase-commit algorithm (that you implemented in your Project). However, we will dealing with a more complex setup where *the transaction coordinator can also crash at any point in time.*

Please Note the following background about the setup :

- Throughout this question, assume that there is a transaction coordinator (master) and some slave servers, which store each key-value pair. The replication factor is some number N > 1.
- The master and the slaves contain enough logic that they can reconnect with each other after recovering from a crash. (All registration data and open sockets/ports are automatically restored on rebuild)
- To refresh your memory , we have provided the state machine diagram for the coordinator and the slaves : [Figure 1]

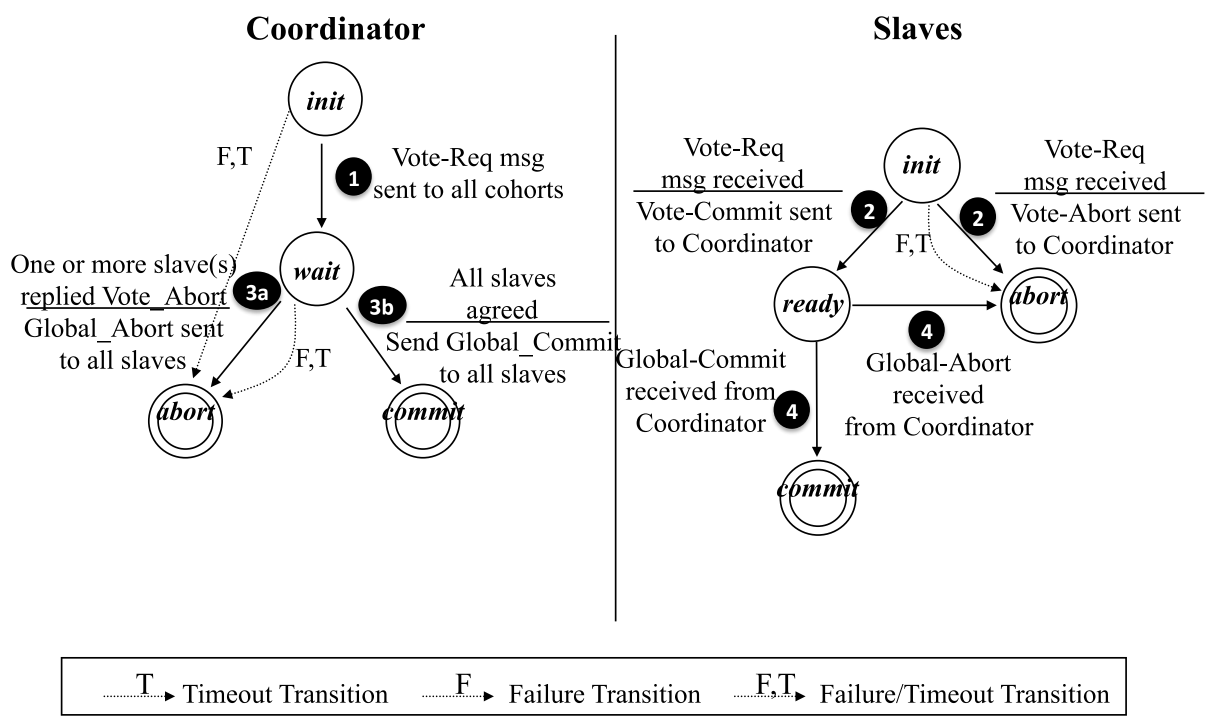

Figure 1: Two phase commit.

(a) (6 points) A client makes a request `PUT(k1, v1)` to your key-value store. Fill in the blanks (some blank lines might be extraneous) to denote the sequence of `SEND(msg)`, `RECEIVE(msg)`, and `LOG(msg)` actions required to execute this client request, while maintaining the robustness of the system to recover from node crashes of EITHER THE MASTER OR THE SLAVE. We have filled in some lines as an example Please note the following to get maximum credit :

- Log the MINIMUM number of messages required.
- Ensure that the master behaves optimally across rebuilds. It must make decisions consistent with the available votes of its slaves (timeouts counts as votes as well). For example, it is suboptimal to issue a GLOBAL-ABORT if it had received VOTE-COMMITs from every node before crashing.

`Phase 1 :`

```
1. At Coordinator :
    a.  _______________________________

    b.  SEND {PUT (k1, v1)} x N slaves

2. At slave :

    a.  _______________________________

    b.  _______________________________

    c.  _______________________________

    d.  _______________________________

Phase 2 :
3. At Coordinator :
    for each slave {
        i.  RECEIVE {VOTE-COMMIT} or RECEIVE {VOTE-ABORT/TIMEOUT}

        ii. _______________________________
    }
    if ( [Case 1] received less than N VOTE-COMMITs or at least 1 VOTE-ABORT ) {

        i.  _______________________________
        for each slave {
            do
                ii. _______________________________
            while ( not RECEIVE{ACK} )

            iii._______________________________
        }
    } else if ( [Case 2] ___________________________________________________ ) {

        i.  _______________________________
        for each slave {
            do
                ii.  _______________________________
            while ( not RECEIVE{ACK} )

            iii._______________________________
        }
    }

4. At slave :

    a.  _______________________________

    b.  _______________________________
    c.  SEND {ACK}
```

```
Post Phase 2: Finish and Return

5.  At Coordinator:
    a.  -------------------------------

    b.  SEND {SUCCESS/ERROR} to CLIENT
```

(b) (6 points) What happens in the following possible scenarios (Keep your answers to three lines or less) ? Steps refer to numbered lines in your solution above.

i. (2 points) The coordinator crashes at the end of step 1, before it has received any votes, and wakes up after one timeout. Describe the sequence of actions required to recover the coordinator until it gets through step 3.

ii. (2 points) The coordinator crashes somewhere in step 3 after receiving all votes. List actions required to rebuild the coordinator and complete step 3 in each case.

iii. (2 points) What happens to the master if it crashes at the VERY BEGINNING of step 5 before any actions 5a or 5b occur?

(c) (6 points) Suppose in phase 2, when a coordinator is down, a slave can contact other slaves to determine the coordinator's GLOBAL-ABORT/COMMIT decision. [1]) Following is a list of scenarios that a slave can learn about the other slaves. What should the former do in the following cases ?

i. (2 points) If at least one other slave has received a GLOBAL-COMMIT/GLOBAL-ABORT message ?

ii. (2 points) If no other slaves have received a global Phase 2 decision but at least one slave performed a VOTE_ABORT .

iii. (2 points) If all participants have made a VOTE_COMMIT but no one has heard back from the Coordinator ?

[1] The slaves may do this to speed quickly finish this transaction and elect a new coordinator for the next transaction

(d) (1 point) With regards to time it takes to atomically complete one distributed transaction and your answers to the previous subparts, what do you think is the most major drawback of the two-phase protocol ?

(e) (6 points) Suppose, we see that the KV-store has many more client PUT requests than GET requests. In order to shorten the time taken for a client PUT request, we make the coordinator respond back to the client after only one slave returns ACK in phase 2 of the two-phase commit. Our objective is to implement GET without modifying the KV-store consistency guarantees.

i. (3 points) What extra information do you need to associate with each (key, value) pair ?

ii. (3 points) Describe in one line or less, how you would implement a GET(key) request.

4. (25 points) **mmap and virtual address space**

Consider a two-level memory management scheme on 24-bit virtual addresses using the following address formats in a big-endian system (i.e. the most significant byte is the first byte in memory):

Virtual Addresses

VPN1 (8 bits)	VPN2 (8 bits)	Offset (8 bits)

Physical Addresses

PPN (8 bits)	Offset (8 bits)

Page table entries are 16 bits stored in big-endian form:

Page Table Entry

Physical Page Number (8 bits)	Unused (7 bits)	Valid (1 bit)

The base page-table pointer is 0x2000.

The following lines of code are run in a program, followed by several operations in `process(data);` that are not shown.

```
...
int16_t *buf;
int16_t val;
int fd = open("cs162.txt", O_RDWR);
char *data = mmap(0x050000, 2048, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
process(data);
...
```

Given the contents of physical memory in the diagram below, write down the values of the variables described. If the value cannot be determined, write the physical address that page faults. Assume all operations affect subsequent parts of this question.

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x0000	E0	F0	01	11	21	31	41	51	61	71	81	91	A1	B1	C1	D1
0x0010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
…																
0x1010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
0x1020	40	07	41	06	30	06	31	07	00	07	00	00	00	00	00	00
…																
0x2000	21	FE	22	FF	25	FF	22	FF	2F	ff	25	FF	30	FF	22	FF
0x2010	40	81	21	FF	25	80	43	83	00	00	00	00	00	00	00	00
…																
0x2500	10	00	00	01	2F	02	2F	03	28	05	10	06	10	07	30	FF
0x2510	18	85	19	85	1A	85	18	85	1C	85	1D	85	1E	85	00	00
…																
0x2800	50	01	51	03	00	00	00	00	00	00	00	00	00	00	00	00
…																
0x2F00	60	03	28	03	22	03	63	00	64	03	65	00	66	00	67	00
0x2F10	68	00	69	00	00	00	00	00	00	00	00	00	00	00	00	00
0x2F20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
…																
0x3000	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0x3010	01	12	23	34	45	56	67	78	89	9A	AB	BC	CD	DE	EF	00

(a) (2 points) What is the size of a page?

(b) (3 points)

```
buf = (int16_t *) data;
```

What is the value of buf in hexadecimal? Write the faulting address if the value of buf cannot be determined followed by the reason why.

(c) (4 points)

```
val = * (data + 0x116);
```

What is the value of val in hexadecimal? Write the faulting address if the value of val cannot be determined and explain what happens to service the fault.

(d) (4 points)

```
val = * data;
```

What is the value of val in hexadecimal? Write the faulting address if the value of val cannot be determined and explain what happens to service the fault. Indicate by circling them any page table entries that are to be changed.

(e) (4 points)

```
buf = (int16_t *) (data + 0x116);
// SEEK_SET means the offset is relative to the beginning of the file
lseek(fd, 0x311, SEEK_SET);
read(fd, buf, 2);
```

What is the value of buf[0] in hexadecimal? Write the faulting address if the value of val cannot be determined and explain what happens to service the fault. Indicate by circling them any page table entries that are to be changed. Update the contents of physical memory after the operations have executed. Cross out the replaced values in the diagram and update them with data.

(f) (3 points)

```
lseek(fd, 0x211, SEEK_SET);
write(fd, data+0x211, 2);
```

What happens when this is executed?

(g) (3 points) Update the contents of physical memory after the following operations have executed. Cross out the replaced values in the diagram and update them with new data.

```
int16_t new_data = 0x1234;
lseek(fd, 0x316, SEEK_SET);
write(fd, &new_data, 2);
```

(h) (2 points) If two processes mapped this file, how would the sharing be represented in the page tables for each of their virtual address spaces.

5. (0 points) **Participation**

Please enter the name of the TA you interact with the most, they will be grading your course participation which is worth 5% of your course grade.

```
/******************************* Sockets ****************************/
int socket(int domain, int type, int protocol);
int connect(int socket, const struct sockaddr *address, socklen_t address_len);
struct hostent *gethostbyname(const char *name);
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *address, socklen_t *address_len);

/***************************** Strings & Processes ****************************/
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

/******************************* Files ****************************/
int open(const char *pathname, int flags);
off_t lseek(int fildes, off_t offset, int whence);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```