University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2019   John Kubiatowicz

Midterm III
SOLUTIONS
May 2nd, 2019
CS162: Operating Systems and Systems Programming

Your Name:	
Your SID:	
TA Name:	
Discussion Section Time:	

General Information:

This is a **closed book** exam. You are allowed 3 pages of notes (both sides). You have 2 hours to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming. Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	Possible	Score
1	18	
2	17	
3	24	
4	10	
5	31	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [18 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT.*

Problem 1a[2pts]: A single file server can be constructed with a Byzantine Agreement algorithm for file commit such that clients are protected against break-ins to the server.

☐ True ☑ False

Explain: *A Byzantine Agreement algorithm requires 3f+1 servers to tolerate f malicious servers. A system with only one server would tolerate f=0 (i.e. no) breakins.*

Problem 1b[2pts]: When using RPC to communicate over the network, a client and host/server could share arguments and return values even if the communicating entities have processors with different "endianness" (i.e. x86 vs MIPS).

☑ True ☐ False

Explain: *During the marshalling process at the source of an RPC packet, data is normalized into an internet standard format (including endianness). At the destination, data is unmarshalled into the local format, translating into the local endianness.*

Problem 1c[2pts]: When producing a queueing model of some system (such as a filesystem) and the distribution of request arrival times is unknown, the best estimate is to use a *deterministic* distribution.

☐ True ☑ False

Explain: *The best distribution to use when you do not have enough information is typically a "memoryless" distribution since the aggregation of many independent processes often behaves in a memoryless fashion.*

Problem 1d[2pts]: Direct Memory Access (DMA) refers to a situation in which the processor bypasses the cache to access DRAM directly.

☐ True ☑ False

Explain: *DMA is a mechanism that permits transfers directly to/from devices to memory without involving the processor (thereby freeing the processor to do other things).*

Problem 1e[2pts]: The Bitcoin blockchain could be used for distributed decision making.

☑ True ☐ False

Explain: *Proposed decisions are submitted to the mining network as potential transactions. The mining network will serialize them, ultimately putting all proposed decisions into a global order, from which a winner could be selected (like earliest wins, or any other function from a set of proposals to a decision).*

Problem 1f[2pts]: The acronym "RAID" refers to RApid Insect Death, and is a general term for a whole class of insecticides.

☐ True ☑ False

Explain: *RAID stands for "Redundant Arrays of Inexpensive Disks" and refers to a technique for combining disks together for reliability.*

Problem 1g[2pts]: In the Fast File System (FFS) *without a buffer cache,* the number of disk accesses to retrieve the first byte of a file is always less than the number of disk accesses to retrieve the last byte of a file.

☐ True ☑ False

Explain: *For small files that can be described entirely with direct pointers, the first and last byte takes the same number of disk accesses, namely two (one for inode, one for datablock).*

Problem 1h[2pts]: The End-to-End principle suggests that the best way to achieve error-free communication is to provide 100% reliability at every hop in the network through error correction coding and retransmission.

☐ True ☑ False

Explain: *In fact, the End-to-End principle suggests exactly the opposite – since reliability must be guaranteed at the endpoints anyway (there are many ways in which data can be lost other than just along a network line), there is no reason to work extra hard in the network to ensure 100% reliability.*

Problem 1i[2pts]: The Journal in a "Journaled file system" has no benefit unless it can be placed on a separate physical disk from the filesystem.

☐ True ☑ False

Explain: *Journaling helps to ensure consistency of the file system by providing transactional updates to the file system – even when the journal is on the same physical disk as the filesystem.*

Problem 2: Multiple Choice [17pts]

Problem 2a[2pts]: In consistent hashing, if we have N servers, how many servers must we move keys from when we add a server (*choose one*):

A: ☑ one (1).

B: ☐ two (2).

C: ☐ log(N).

D: ☐ N

Problem 2b[2pts]: Memory-mapped I/O is *(choose one)*:

A: ☑ A hardware mechanism for assigning physical memory addresses to devices such that processor read and write operations to these addresses become commands to devices.

B: ☐ A software protocol that constructs a coherent distributed shared memory between multiple physical nodes, allowing communication between nodes to occur as reads and writes to the shared memory (address) space.

C: ☐ A technique for communication between processes using the mmap() system call. As a result, processes can interact via reads and writes to shared memory addresses.

D: ☐ A security mechanism for I/O devices that that prevents user-mode applications from directly accessing these devices, forcing device access to go through the system call interface.

Problem 2c[2pts]: What is two-phase commit (2PC) and what problem does it solve? *(choose one)*:

A: ☐ Two-phase commit is an algorithm that can help to make progress on a distributed consensus problem even in the presence of malicious nodes.

B: ☐ Two-phase commit is used in single-server databases to ensure *serializability* of simultaneous transactions by controlling transaction commit order.

C: ☑ Two-phase commit is a distributed decision making algorithm that makes sure that all participants eventually come to the same decision whether to *commit* or *abort* – despite intermittent crashes and restarts of participants.

D: ☐ Two-phase commit is a distributed decision making algorithm that is valuable because it can continue to make forward progress despite crashed or unavailable participants.

Problem 2d[2pts]: What is Network Address Translation (NAT)? (*Select One):*

A: ☐   A mechanism that is part of the Domain Name System (DNS) for handling website names that are not in English.

B: ☐   A mechanism for load-balancing incoming queries among a set of servers in a warehouse (cloud) computing system.

C: ☑   A mechanism for sharing one or more external IP addresses amongst a larger set of local (non-routable) IP addresses.

D: ☐   A mechanism whereby a local network router can handle outgoing messages by forwarding them to an authoritative router that participates the BGP routing protocol.

Problem 2e[3pts]: Which of the following are true about modern hard disks? (*Mark all that apply):*

A: ☐   They have an independent disk head on every surface of every platter. As a result, the disk can simultaneously read or write from different tracks on different platters.

B: ☑   Some of them gain bit density by overlapping tracks.

C: ☑   They have internal caching, allowing them to read a whole track at a time.

D: ☐   They have a lower bit density on the outside tracks from the inside tracks (because the surface of the outside tracks move under the disk head faster than that of the inside tracks).

E: ☑   Their internal controllers can queue requests and perform variants of the elevator algorithm without consulting the operating system.

F: ☐   They support bit densities ≥ 1 Exabit (10^{15})/square inch

Problem 2f[3pts]: Little's Theorem has the following properties (*Mark all that apply):*

A: ☐   It applies only to memoryless arrival distributions.

B: ☑   It says that the average number of jobs in the system is equal to the average arrival rate of jobs multiplied by the average length of time that a job stays in the system.

C: ☐   It shows why the average length of time spent in a queue grows without bound as the system utilization approaches 100%.

D: ☑   It applies only to systems in equilibrium.

E: ☑   It can be used to compute the average time spent in a queue, given the average length of the queue.

F: ☐   None of the above.

Problem 2g[3pts]: The Berkeley FFS has the following properties (*Mark all that apply)*:

A: ☐   It changed the inode format from the BSD 4.1 file system to better reflect the overwhelming presence of small files in a typical UNIX filesystem.

B: ☐   It placed all the inodes together on the inner tracks of the disk for better performance.

C: ☑   It reserved 10% of the blocks to ensure that new files could get sequential groups of blocks for better read performance.

D: ☑   It performed skip-sector allocation of blocks to prevent processor delays during reading from missing blocks and forcing a complete rotation for each block read.

E: ☑   It placed the inodes and blocks for files within a directory close to the blocks and inodes for the directory to improve performance.

F: ☐   It introduced a B-tree format for directories in order get better lookup performance for directories with large numbers of files.

Problem 3: Pintos/File Systems [24pts]

Consider the following inode_disk and indirect_block struct similar to what is in Pintos. Suppose sectors are 512 bytes.

```
struct inode_disk {
    block_sector_t direct[12];
    block_sector_t indirect;
    block_sector_t triply_indirect; // Note that this is a triply
                                    // and that there is no doubly!
    size_t size;
    void unused[113];
};

struct indirect_block { // doubly and triply indirect blocks look like this
    block_sector_t block_nums[128];
}
```

Problem 3a[2pts]: What is the maximum file size supported by this design? Leave your answer unsimplified in the box:

$$512 \times (12 + 128 + 128 \times 128 \times 128)$$

Problem 3b[4pts]: How many sectors of each type are required to represent a maximum sized file in this design? Leave your answer unsimplified in the box:

Data blocks:

$$12 + 128 + 128 \times 128 \times 128$$

Indirect blocks:

$$1 + 128 \times 128$$

Doubly indirect blocks:

$$128$$

Triply indirect blocks:

$$1$$

Problem 3c[6pts]: Suppose that we have a file that is represented using the inode structures of (3a) and (3b) and that the current size of the file is 12 * 512 bytes long. We now want to write 512 characters to the end of this file. You may use the following functions:

```
void block_read(void *buffer, block_sector_t block_num);
void block_write(void *buffer, block_sector_t block_num);
block_sector_t block_allocate();
```

The block sector corresponding to the file's inode_disk struct is 3:

```
block_sector_t FILE_INODE_BLOCK = 3;
```

Fill in the blanks in the following function such that after running this function, 512 characters are written to this file and persisted. Assume that the file is exactly 12 blocks long already. You may not need all of the blanks, but may only have 1 semicolon per line.

```
void Append512Chars () {
    struct inode_disk inode;

    block_read(&inode, FILE_INODE_BLOCK);  // Read inode from disk

    inode.indirect = block_allocate(); // Allocate new block for indirect

    struct indirect_block indirect_block;
    indirect_block.block_nums[0] = block_allocate();
    void buffer[512];
    memset(buffer, 'a', 512);

    inode.size += 512; // Change the size

    block_write(buffer, indirect_block.block_nums[0]); // Persist data

    block_write(indirect_block, inode.indirect); // Persist indirect

    block_write(inode, FILE_INODE_BLOCK); // Persist inode
}
```

Suppose that we build our disk subsystem to handle a high rate of I/O by coupling many disks together. Properties of this system are as follows:
- Has a total of 24 disks, each of which is 2TB in size
- Uses disks that rotate at 12,000 RPM, have a data transfer rate of 81.92 MBytes/s (for each disk), and have an 4.5 ms average seek time, 4KiB sector size
- Has a SCSI interface with an 600μs controller command time. Assume that a group of consecutive sectors can be fetched with a single request.
- Has a file system that groups sectors into 32KiB blocks
- Is limited only by the disks (assume that no other factors affect performance).

Each disk can handle only one request at a time, but each disk in the system can be handling a different request. The data is not striped (all I/O for each request has to go to one disk).

EACH OF THE FOLLOWING ANSWERS SHOULD BE SIMPLIFIED TO SINGLE NUMBERS. HOWEVER, YOU MUST SHOW YOUR WORK TO CREDIT.

Problem 3d[4pts]: What is the average *service time* to retrieve a single disk block from a random location on a single disk, assuming no queuing time (i.e. the unloaded request time)? *Hint: there are four terms in this service time!*

$$T_{ser} = T_{ctrl} + T_{seek} + T_{rot} + T_{xfer} =$$
$$= 0.6ms + 4.5ms + \frac{1}{2} \frac{60,000 \; ms/min}{12,000 \; rev/min} + \frac{32768 \; bytes}{81.92 \times 10^6 \; bytes/s \times 10^{-3} s/ms}$$
$$= 0.6ms + 4.5ms + 2.5ms + .4ms = 8ms$$

Problem 3e[2pts]: Assume that the OS is not particularly clever about disk scheduling and passes requests to the disk in the same order that it receives them from the application (FIFO). If the application requests are randomly distributed over a single disk, what is the bandwidth (bytes/sec) that can be achieved?

BW = 32768 bytes / T_{ser}
*   = 32768 / (8ms × 10⁻³ s/ms)*
*   = 4,096,000 bytes/s = 4.096 MBytes/s*

Problem 3f[2pts]: Suppose that the application has requests outstanding for all disks (but they are still randomly distributed, handed FIFO to disks), what is the maximum number of I/Os per second (IOPS) for the whole disk subsystem (an "I/O" here is a block request)?

1 Disk IOP takes 8ms = 0.008s ⟹ (1/0.008) IOPS = 125 IOPS
24 Disks IOPS = 24 × 125 IOPS = 3000 IOPS

Problem 3g[4pts]: Treat the entire system as an M/M/m queue (that is, a system with m servers rather than one), where each disk is a server. All requests are in a single queue. Assume that the disk system receives an average of 2250 I/O random requests per second. For simplicity, assume that any disk can service any request. Assuming FIFO scheduling by the OS again, what is the mean response time of the system? You might find the following equation for an M/M/m queue (where any server can handle any request from the queue) useful:

$$\text{Server Utilization} \,(\zeta) = \frac{\lambda}{\dfrac{1}{\text{Time}_{server}\,/\,m}} = \lambda \times \frac{\text{Time}_{server}}{m}$$

$$\text{Time}_{queue} = \text{Time}_{server} \times \left[\frac{\zeta}{m(1-\zeta)} \right]$$

λ = 2250 IO/s

T_{server} = 8ms/IO = 0.008 s/IO

ζ = 2250 IO/s × 0.008 s/IO / 24 = 2.25 / 3 = (9/4) / 3 = ¾ = 0.75

$T_{queue} = T_{server} \times \left[\dfrac{0.75}{24(0.25)}\right] = 0.008 \, s \times \left[\dfrac{3}{24}\right] = 0.001s = 1ms$

$T_{system} = T_{queue} + T_{server} = 1ms + 8ms = 9ms$

[This Page Intentionally Left Blank]

Problem 4: Global Potpourri [10pts]
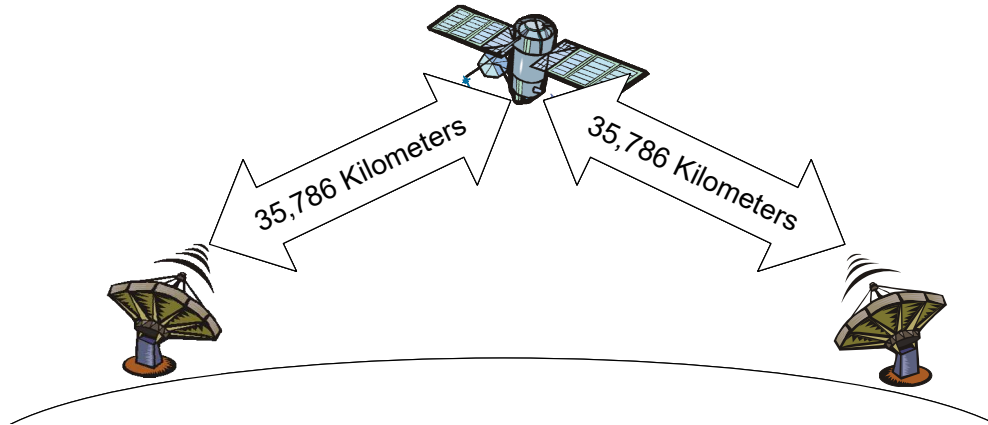
Problem 4a[4pts]: Suppose that you have a dedicated satellite connection between two points on the earth. The satellite will be in a geostationary orbit (at an altitude of 35,786 km). Assume that we will be using TCP over this connection. Assume that the satellite forwards all bits after a 100ms latency (fully pipelined, since there is no routing in the satellite). Also assume that we can use "jumbo TCP frames" with an MTU of 9000 bytes.

Parameter	Value
Speed of light	3×10^5 km/s
Transmission Bandwidth	5×10^9 bits/s
TCP/IP header size	40 bytes
MTU	9000 bytes
Data payload for ACK (has TCP/IP header)	0 bytes

Assume that computational time at the receiver (copying, code execution, etc) is zero. Accounting only for transmission time, what should the sender's TCP window size be to achieve maximum bandwidth? Assume no packets are dropped. Show all of your work (no credit for a single number). *Hint: don't forget that the TCP/IP header is overhead.*

If you do not have a calculator, you may leave the result as a simple mathematical expression. However, PLEASE PLUG IN ALL NUMBERS AND SPECIFY FINAL UNITS!

In order to ensure maximum bandwidth, we need to allow many packets to be in transit before we receive the first ACK. Once data is ACKed, we know that it has successfully made it to the destination and can be discarded. Consequently, the size of the transmission window is simply the amount of data that can be transmitted during one round-trip time.

1 RTT = 2 × 1-way time = 2 × [2 × 35786 km/(300000km/s) +0.100s] = 2× 0.338s = 0.676s

Total BW = data bandwidth = [(9000-40)/9000] × 5×10⁹ = 4.98×10⁹ bits/s = 6.23 ×10⁸ bytes/s

So, Windows = 6.23 × 10⁸ bytes/s × 0.676s = 4.21 × 10⁸bytes

(Note that technically this is: 4.21 ×10⁸/(1024)² MiBytes = 401MiBytes)

Problem4b[3pts]: In class, we discussed the Chord distributed key-value store, which uses Consistent Hashing to store data among a large set of servers. Explain how key-value pairs are distributed among servers in Chord and how is it possible for the '`get(key)`' process to return the value associated with '`key`' in `O(log(n))` steps where n is the number of servers.

Each server has a randomly assigned name in an m bit space (e.g. derived by an m-bit hash over the IP address). Keys are also m bits in length. Servers are arranged in a large virtual ring, positioned by their m-bit names. (the ring is all values from 0 … 2^m-1, wrapping around at zero).

Each key is placed on the server that has the smallest name value that is $\leq$ the key. Since keys and server names are randomly distributed this is probabilistically evenly distributed.

Servers know the identity (IP address) of the servers that are before (predecessor) and after (successor) them in the ring, allowing a guaranteed O(n) lookup of any key. To achieve O(log(n)), servers also dynamically discover "finger" pointers at geometrically distributed points on the ring, namely $\frac{1}{2}$-way, $\frac{1}{4}$-way , $\frac{1}{8}$-way, … $\frac{1}{m}$-way around the ring. These fingers allow short-cut routing to keys that takes, on average, log(n) hops to find a key.

Problem 4c[3pts]: Suppose that we have storage servers in machine rooms on every continent and wish to use them to provide "deep archival" service (the ability to recover data with extremely high probability despite a variety of global disasters). Suppose further that we only have enough total disk space for no more than a factor of four (x4) blowup in the size of the data. What can we do that would be (much) better than simply making four copies of every piece of data? (*Hint: your answer has something to do with encoding and placement*).

We should use an erasure coding (such as Reed Solomon code mentioned in class) which requires any m of n "fragments" of data to be retrieved in order to reconstruct data. If n/m = 4, then the total data size is a 4-times the original data size, thereby meeting our size constraints. For example, we talked about m=16, n=64 in class.

We must also make sure that our various fragments are stored on servers that have independent failure modes. For instance, storing all n fragments in the same building, state, or even continent is a bad idea, because natural disasters could take out so many fragments that we cannot recover our data.

Problem 5: Remote File Server [31 pts]

In this problem, you will design a simple file server and make it efficient by using a pool of threads to handle the file service. The basic idea is illustrated by the diagram below:
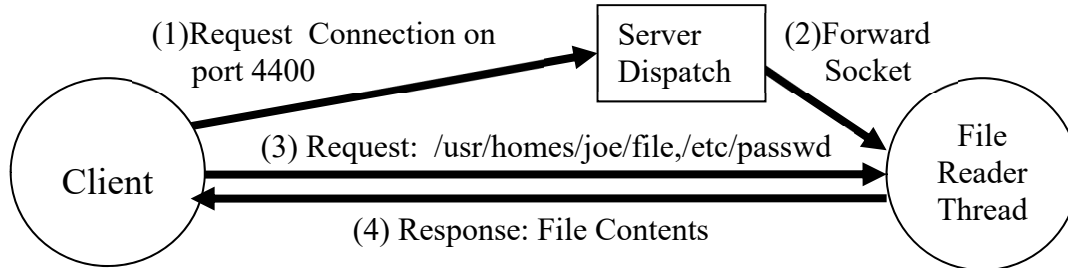
Here, the client sets up a TCP socket connection with the server on port 4400, then sends one or more file names separated by "," and/or newline characters. Finally, the file reader thread sends the contents of the specified file(s). The client terminates by sending a blank line.

Note: READ THE PROBLEM FIRST AND REVIEW DOCUMENTATION AS NEEDED! YOU CAN SCAN THROUGH FUNCTION DEFINITIONS QUICKLY FIRST TIME THROUGH.

We are going to build the server using the C sockets interface coupled with streaming functions. Some possibly helpful operations for establishing server-side connections are as follows:

Function	Usage

Create a socket for further use. Returns file descriptor.	```int sockfd = socket(int family, int type, int prot);```
	
	 ```sockfd: Returned file descriptor```
	 ```family: AF_INET, AF_INET6```
	 ```type: SOCK_STREAM (i.e. tcp), SOCK_DGRAM (i.e. UDP)```
	 ```prot: 0```

Bind a name (i.e. IP address and port) to a socket	```int err = bind(int sockfd, struct sockaddr *addr,```
	 ``` socklen_t addrlen);```
	
	 ```err: 0 (success), -1 (error)```
	 ```sockfd: Socket file descriptor```
	 ```addr: Address to bind```
	 ```addrlen: length of address structure```

Listen on socket for incoming requests. Starts listening process, then returns.	```int err = listen(int sockfd, int backlog);```
	
	 ```err: 0 (success), -1 (error)```
	 ```sockfd: Socket file descriptor```
	 ```backlog: Maximum number of queued requests```

Accept connection on listening socket. Blocks until available connection.	```int confd = accept(int sockfd, struct sockaddr *addr,```
	 ``` socklen_t *addrlen);```
	
	 ```confd: >0 (returned connection), -1 (error)```
	 ```sockfd: Listening Socket```
	 ```addr: Structure for address of remote connection```
	 ``` (can be null)```
	 ```addrlen: Length of address structure (addr)```

Internet addresses are specified in address structures (of type struct sockaddr_in) in the following format:

```
struct sockaddr_in { /* IPv4 socket address */
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
      ... /* other padding to match size of struct sockaddr */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

All values must be specified in *network byte order*. The sin_family component is always set to AF_INET. The s_addr component of sin_addr can either be set to a special value INADDR_ANY, which means bind to "any" local address or set using a special network byte order function. The port must be set in network order as well, using the htons(). For example, to set up an IP address structure pointing at IP address 192.168.1.1 on port 80 we could do:

```
struct sockaddr_in my_addr;
bzero((char *)&my_addr,sizeof(my_addr)); /* Zero out unused bytes*/

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(80);
/* Set IPv4 address field from a "dot-notation"  address. */
inet_aton("192.168.1.1", &(my_addr.sin_addr.s_addr));
```

An alternative that is useful for servers is to bind to all local addresses (but a specific port) by using a wildcard address as follows:

```
/* Alternate: bind to all local addresses (useful for servers) */
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(80);
my_addr.sin_addr.s_addr = INADDR_ANY;
```

Finally, note that pointers to addresses of type "struct sockaddr_in *" are cast to the generic type "struct sockaddr *" for input to socket functions such as bind().

Other functions that might be useful for you:

Function	Usage

Wrap file descriptor into a stream so that streaming functions can be used.	`FILE *mystream = `**`fdopen`**`(int fd, char *mode);`
	
	`fd:    Input file descriptor`
	`mode: "r" (reading), "r+" reading/writing`

Open a file as a stream for reading and/or writing.	`FILE *mystream = `**`fopen`**`(char *path, char *mode);`
	
	`path: Pathname to file`
	`mode: "r" (reading), "r+" reading/writing`

Close stream	`int err = `**`fclose`**`(FILE *mystream);`
	
	`err:      0 (success), -1 (error)`
	`mystream: Stream to close`

Read from stream	`size_t num = `**`fread`**`(void *ptr, size_t size,`
	`                    size_t nmemb, FILE *stream);`
	
	`num:    0 (error or end of file)`
	`        >0 (number of bytes read)`
	`size:   size in bytes of items to be read`
	`nmemb:  MAX number of items (of size bytes) to read`
	`stream: Input stream`

Write to stream	`size_t num = `**`fwrite`**`(void *ptr, size_t size,`
	`                    size_t nmemb, FILE *stream);`
	
	`num:    0 (error or end of file)`
	`        >0 (number of bytes written)`
	`size:   size in bytes of items to be writen`
	`nmemb:  Number of items (of size bytes) to write`
	`stream: Output stream`

Read Line from stream into buffer.	`char *line = `**`fgets`**`(char *buf, int size,`
	`                    FILE *stream);`
	
	`line:  0 (EOF or error)`
	`       <>0: pointer to null-terminated string`
	`            Including the newline character ('\n').`
	`buf:   Char buffer for storing result`
	`size:  Buffer size.  Output will be <= size-1 chars.`

Parse input string for tokens. First call sets up parsing and returns first token, subsequent calls return following tokens	`char *nexttok = `**`strtok_r`**`(char *instring, char *delim,`
	`                         char **saveptr);`
	
	`nexttok:   0 (no remaining tokens)`
	`           <>0: pointer to null-terminated string`
	`instring:  0 (continue parsing previous string)`
	`           <>0: start new parsing job at instring`
	`delim:     String with delimiting characters`
	`saveptr:   Pointer to char pointer for state storage.`
	`           This item initialized when instring <> 0`
	`           and used when instring == 0.`

Code Version #1: First, we are going to make a version of the server that doesn't use threads. Don't worry about error conditions and assume that constants such as MAXNAME are defined elsewhere. Our code looks like this.

```
    main() { /* don't worry about how main thread starts */
1.     int newsockfd;
2.     FILE *newsockstr;
3.     char filenamebuf[MAXNAME], *nextinputline, *nextfn, *tokstate;
4.     FILE *my_file;
5.     char mybuff[BUFSIZE];
6.     int numchars;

7.     struct sockaddr_in serv_addr; /* listen socket address */
8.     int lstnsockfd; /* listen socket file descriptor */
9.     lstnsockfd = socket(AF_INET, SOCK_STREAM, 0);
10.    bzero((char *)&serv_addr, sizeof(serv_addr);

11.    /* Bind address to socket */
12.    /* Setup Server Socket to accept connections */

       while (true) {
13.       newsockfd = /* Get next socket connection */
14.       newsockstr = fdopen(newsockfd, "r+"); /* fd=>stream*/
15.       while (nextinputline = /*  Get input line */ ) {
16.          /* Extract next file name delimited by \r,\n,\t or ,*/
17.          nextfn = strtok_r(nextinputline,"\r\n\t, ",&tokstate);
18.          do {
19.             if (my_file = /* open file for reading */) {
20.                while ((numchars=fread(mybuff,1,BUFSIZE,my_file))> 0) {
21.                   fwrite (mybuff, 1, numchars, newsockstr);
22.                }
23.                fclose(my_file);
24.             }
25.          } while (nextfn=strtok_r(0,"\r\n\t, ",&tokstate));
26.       }
27.       fclose(newsockstr);
       }
    }
```

Problem 5a[4 pts]: Assuming that this server listens on port 4400, complete missing code for Line #11, above. This should be 4 lines of code.

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(4400);
serv_addr.sin_addr.s_addr = INADDR_ANY;

bind(lstnsockfd, (struct sockaddr *)serv_addr, sizeof(serv_addr));
```

Problem 5b[4pts]: Complete missing code for Lines #12, 13, 15, and 19. None of these should be more than 1 line of code.

Line #12:   *listen(lstnsockfd, 4192); // Any queue length ok here*

Line #13:
newsockfd = *accept(lstnsockfd, 0, 0); // No address structure*

Line 15:
while (nextinputline = *fgets(filenamebuf, MAXNAME, newsockstr)*) {

Line 19:
if (my_file = *fopen(nextfn,"r")*   ) {

Code Version #2: A big problem with code version #1 is that it can only process one request at a time. Here is a sketch of our code (don't worry about ordering between dispatch thread and worker thread in the source file):

```
    // Server Dispatch Thread -------------------------------------
    main() {
28.    /* Setup Server Code (same as Lines #7-12 in Version #1) */
29.    while(true) {
30.      pthread_t thread_id;  /* Will ignore this */

31.        /* Server Dispatch Code */
32.    }
    }

    // Worker Thread Code -----------------------------------------
33. void *fileworker(void *arg) {
34.    FILE *newsockstr = (FILE *)arg;

35.    /* Worker Thread Code */

36.    return 0;
37. }
```

Problem 5c[6pts]: Finish writing the above code so that it will produce a multithreaded server. *The server dispatch code should be as short as possible so that the work of interacting with a client is mostly within the* `fileworker()` *thread.* The following pthread library function will be useful:

Function	Usage
Create new thread	```int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg); thread: Space for thread identifier attr: Pointer to attributes. 0=>default start_routine: Pointer to thread run routine arg: Pointer to argument```

Note that every comment in the above sketch may represent one or more lines of code. When you have any lines of code that are duplicated from Version #1, simply use line numbers, for instance "Line 1" to reuse line #1 or "Lines 7-12" to duplicate lines 7-12. *You must reuse every numbered line from Version #1 (except for lines #7-12 which are already included). Don't forget the casts to and from (void *) for the pthread_create interface.*

Code to fill line 31:
```
Lines 1-2;
Lines 13-14;
pthread_create(&thread_id,0,&fileworker,(void *)newsockstr);
```

Code to file line 35:
```
Lines 3-6;
Lines 15-27;
```

Code Version #3: For efficiency, we would like to make sure than only NUMTHREADS threads are ever running at any one time. To do this, we will pre-construct the threads, then hand them requests from a queue. This is called using a "Thread Pool".

First, we must construct an appropriate queue. What we want is an atomic queue that can handle multiple simultaneous enqueue/dequeue operations with the following functions:

```
/* Create new aqueue */
atomqueue *new_atomqueue();

/* Enqueue object on atomic queue */
void atomEnqueue(atomqueue *aqptr, void *obj); /* Enqueue obj*/

/* remove next object or sleep until object ready*/
void *atomDequeue(atomqueue *aqptr);
```

Note that the important aspect of this implementation is that a call to atomDequeue() will sleep on an empty queue, *not* return a null entry. Further, we want to make sure that many threads can simultaneously call atomEnqueue() and atomDequeue() without fear of an incorrect result.

We are going to construct this queue from a non-atomic (non-thread-safe) queue *using a mesa-style Monitor*. The following pthreads functions will be helpful:

Function	Usage	
Pthread mutex functions.	`pthread_mutex_t lock;`	`/* Declare */`
	`pthread_mutex_init(&lock,NULL);`	`/* Init */`
	`pthread_mutex_lock(&lock);`	`/* Lock */`
	`pthread_mutex_unlock(&lock);`	`/* Unlock */`
Pthread conditional variable functions.	`pthread_cond_t CV;`	`/* Declare */`
	`pthread_cond_init(&CV);`	`/* Init */`
	`pthread_cond_wait(&CV, &lock);`	`/* Wait */`
	`pthread_cond_signal(&CV);`	`/* Signal */`

Assume that we have non-atomic methods for a queue as follows:

```
queue *new_queue();                 /* Create new queue */

int checkqueue(queue *myqueue);     /* Number entries on queue*/

void enqueue(queue *myqueue, void *obj); /* Never fails */

void *dequeue(queue *myqueue);      /* remove next object or
                                       return null if empty */
```

Here is a sketch of the code:

```
38. typedef struct aqueue {
39.    queue *myqueue;
40.    pthread_mutex_t lock;
41.    pthread_cond_t CV;
42. } atomqueue;

43. atomqueue *newatomqueue() {
44.    atomqueue* aqptr =(atomqueue *)malloc(sizeof(atomqueue));
45.    aqptr->myqueue = new_queue();  /* Create queue */
46.    pthread_mutex_init(&aqptr->lock);
47.    pthread_cond_init(&aqptr->CV);
48.    return aqptr;
49. }

50. void atomEnqueue(atomqueue *aqptr, void *obj) {
51.    /* Synchronization Entry Code */
52.    enqueue(aqptr->myqueue, obj)
53.    /* Synchronization Exit Code */
54. }

55. void *atomDequeue(atomqueue *aqptr) {
56.    /* Synchronization Entry Code */
57.    void *result = dequeue(aqptr->myqueue);
58.    /* Synchronization Exit Code */
59.    return result;
60. }
```

Problem 5d[8pts]: Fill in the missing pieces of the code. Each of lines #51, #53, #56, and #58 can have more than one line of code – *but no more than 4 lines*. Keep answer really short! Most of these have one or two lines. Credit will only be given for short answers.

Code for line #51:

```
pthread_mutex_lock(&aqptr->lock);
```

Code for line #53:

```
pthread_cond_signal(&aqptr->CV);
pthread_mutex_unlock(&aqptr->lock);
```

Code for line #56:

```
pthread_mutex_lock(&aqptr->lock);
while (!checkqueue(&aqptr->myqueue))
    pthead_cond_wait(&aqptr->CV,&aqptr->lock);
```

Code for line #58:

```
pthread_mutex_unlock(&aqptr->lock);
```

Finally, our version #3 thread-pool code might appear as follows. Recall that, for efficiency, we would like to make sure than only NUMTHREADS threads are ever running at any one time. To do this, we will pre-construct the threads, then hand them requests from a queue. This is called using a "Thread Pool". A server dispatch thread will accept connections and put them on the queue.

```
    // Server Dispatch Thread -------------------------------------
    main() {
61.    atomqueue *myqueue = newatomqueue();
62.    pthread_t thread_id;  /* Will ignore this */
63.    for (int count = 0; count < NUMTHREADS; count++)
64.        /* Start one thread-pool thread */

65.    /* Setup Server Code (same as Lines #7-12 in Version #1) */
66.    while(true) {
67.        /* Server Dispatch Code */
68.    }
    }

    // Worker Thread Code -----------------------------------------
69. void *fileworker(void *arg) {
70.    atomqueue *myqueue = (atomqueue *)arg;

71.    /* Worker Thread Code */

72.    return 0;
73. }
```

Problem 5e[9pts]: Fill in the missing pieces of the above code. Missing chunks of code are in lines #64, #67, and 71 above. These can represent more than one line of code. Keep answers as short as possible. Once again, *you must use every numbered line from Version #1 (except for lines #7-12 which are already included).*

Code to fill in line #64:

```
    pthread_create(&thread_id,0,&fileworker,(void *)myqueue);
```

Code to fill in line #67:

```
    Lines 1-2;
    Lines 13-14;
    atomEnqueue(myqueue,(void *)newsockstr);
```

Code to fill in line #71:

```
    Lines 3-6;
    FILE *newsockstr;
    while(true){
        newsockstr = (FILE *)atomDequeue(myqueue);
        Lines 15-27;
    }
```

[This Page Intentionally Left Blank]

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]