**1. tfs\_read(int fd, void \*p, int n)**

1. Assign f as a pointer to a struct file.
2. Call fd\_to\_file function passing fd and the address of f.
   1. Check if fd is valid, and if it is not, return -1.
   2. Get the file associated with fd and store it in f.
3. Call fileread function passing f, p, and n.
   1. Read n bytes from f into the buffer pointed by p.
   2. Return the number of bytes read.

**2. fd\_to\_file(int fd, struct file \*\*pf)**

1. Declare a pointer f to a struct file.
2. Check if fd is invalid or the file associated with fd is not found. If it is, return -1.
3. Check if pf is not NULL. If it is not, set the value of \*pf to f.
4. Return 0 to indicate success.

**3. fileread(struct file \*f, char \*addr, int n)**

1. Declare an integer variable r.
2. Check if f is readable. If it is not, return -1.
3. Check if the type of file pointed to by f is FD\_INODE. If it is, execute the following steps:
   1. Call the readi function passing the inode associated with f, the buffer pointed to by addr, the offset of the file in the inode, and the number of bytes to read.
   2. Read n bytes from the inode into the buffer pointed by addr.
   3. Return the number of bytes read.
   4. Update the offset of the file by adding r to f->off.
   5. Return r to indicate success.
4. If the type of file pointed to by f is not FD\_INODE, panic.
5. Return -1 to indicate an error.

**4. readi(struct inode \*ip, char \*dst, uint off, uint n)**

1. Declare an unsigned integer variable tot and another unsigned integer variable m.
2. Check if the offset off is greater than the size of the inode or off+n would wrap around. If it is, return -1 to indicate an error.
3. If the sum of off and n is greater than the size of the inode, set n to the difference between the size of the inode and off.
4. For tot initialized to 0, loop while tot is less than n.
   1. Calculate the block number to read by calling bmap function passing the inode associated with ip and off/BSIZE.
   2. Read BSIZE bytes from the block number calculated in the previous step into the buffer buf.
   3. Check if the read is successful. If it is not, panic.
   4. Set m to the minimum value between n-tot and BSIZE - off%BSIZE.
   5. Copy m bytes from buf + off%BSIZE to dst.
   6. Increment tot by m.
   7. Increment off by m.
   8. Increment dst by m.
5. Return n to indicate success.

**5. readfsinfo**()

1. Read the first block of the file system into the buffer buf.
2. Copy the contents of the buffer buf to the superblock structure sb.
3. Read the second block of the file system into the buffer buf.
4. Copy the contents of the buffer buf to the inode bitmap.
5. Read the third block of the file system into the buffer buf.
6. Copy the contents of the buffer buf to the data bitmap.
7. For each of the four blocks containing inodes, do the following:
   1. Read the block into the buffer buf.
   2. If the read is unsuccessful, panic.
   3. For each of the eight inodes in the block, copy the contents of the buffer buf to the corresponding inode in the array inodes.
8. For each inode, copy the value of nlink to ref.
9. Return from the function.

**6. panic(char \*s)**

1. Print the string pointed to by s to the console.
2. Call the exit function passing 1 as an argument to terminate the program with a non-zero exit status.

**7. bmap(struct inode \*ip, uint bn)**

1. Declare an unsigned integer variable addr.
2. Check if bn is less than NDIRECT. If it is, execute the following steps:
   1. Check if the block number bn is allocated for the inode. If it is not, allocate a new block and assign its address to the block number bn for the inode.
   2. Return the address of the block number bn.
3. If bn is greater than or equal to NDIRECT, panic.
4. Return -1 to indicate an error.

**8. balloc()**

1. Declare an unsigned integer variable m.
2. Loop over each bit in the databitmap starting from the ninth bit (index 8) up to the 1024th bit (index 1023).
   1. Calculate the bit mask by shifting 1 left by bi % 32.
   2. Check if the block corresponding to the current bit is free by performing a bitwise AND operation between the bit mask and the corresponding word in databitmap. If the result is 0, the block is free.
   3. If the block is free, mark it as in use by performing a bitwise OR operation between the bit mask and the corresponding word in databitmap.
   4. Write all zeros to the new block by calling bwrite with the block index and a buffer filled with zeros.
   5. Return the index of the newly allocated block.
3. If no free block is found, panic.
4. Return -1 to indicate an error.

**9. bwrite(uint block, char \*buf)**

1. Calculate the offset of the block in the file system by multiplying the block number block by BSIZE.
2. Set the file pointer to the calculated offset by calling lseek function passing the file descriptor fs, the calculated offset, and SEEK\_SET as the third argument.
   1. If the return value of lseek is less than 0, panic.
3. Write BSIZE bytes from the buffer pointed to by buf to the file system by calling write function passing the file descriptor fs, the buffer buf, and BSIZE as the third argument.
   1. If the return value of write is less than 0, panic.
4. Return 0 to indicate success.

**10. fdalloc(struct file \*f)**

1. Declare an integer variable fd.
2. Loop over each file descriptor from 0 to NOFILE-1.
   1. Check if the file pointer at the file descriptor index fd of the open file table of the current process is NULL. If it is, execute the following steps:
   2. Set the file pointer at the file descriptor index fd of the open file table of the current process to the given file pointer f.
3. ii. Return the file descriptor index fd.
4. If no free file descriptor is found, return -1 to indicate an error.

**11. tfs\_write(int fd, void \*p, int n)**

1. Declare a pointer to struct file named f.
2. Call the fd\_to\_file function passing the file descriptor fd and the address of the pointer f.
   1. If the return value of fd\_to\_file is less than 0, return -1 to indicate an error.
3. Call the filewrite function passing the file pointer f, the buffer pointer p, and the number of bytes to write n.
   1. Return the return value of filewrite to indicate success.

**12. filewrite(struct file \*f, char \*addr, int n)**

1. Declare an integer variable r.
2. Check if the file is writable. If it is not, return -1 to indicate an error.
3. If the file type is FD\_INODE, execute the following steps:
   1. Calculate the maximum number of bytes that can be written in a single write transaction without exceeding the maximum log transaction size. This includes the inode, indirect block, allocation blocks, and 2 blocks of slop for non-aligned writes.
   2. Declare an integer variable i and initialize it to 0.
   3. While i is less than n, do the following:
   4. Calculate the number of bytes to write in this transaction by subtracting i from n and taking the minimum of the result and max.
4. ii. Call the writei function passing the inode pointer f->ip, the buffer pointer addr + i, the offset in the inode f->off, and the number of bytes to write n1.
   1. If the return value of writei is greater than 0, increment the file offset f->off by the return value of writei.
   2. If the return value of writei is less than 0, break out of the loop.
   3. If the return value of writei is not equal to n1, panic.
   4. Increment i by the return value of writei.
   5. If i is equal to n, return n to indicate success. Otherwise, return -1 to indicate an error.
5. If the file type is not FD\_INODE, panic and return -1 to indicate an error.

**13. writei(struct inode \*ip, char \*src, uint off, uint n)**

1. Declare unsigned integer variables tot and m.
2. Check if the offset is greater than the size of the file or if the sum of the offset and the number of bytes to write is less than the offset. If either condition is true, return -1 to indicate an error.
3. Check if the sum of the offset and the number of bytes to write is greater than the maximum file size. If it is, return -1 to indicate an error.
4. Loop over each block needed to write the given number of bytes.
   1. Read the block containing the offset from disk by calling bread function passing the block number obtained from bmap function, and a buffer buf.
   2. If the return value of bread is less than 0, panic.
   3. Calculate the number of bytes to copy in this block by subtracting tot from n and taking the minimum of the result and the number of bytes remaining in this block.
   4. Copy the bytes to the block buffer by calling memmove function passing the destination buffer buf + off % BSIZE, the source buffer src, and the number of bytes to copy m.
   5. Write the updated block back to disk by calling bwrite function passing the block number obtained from bmap function and the block buffer buf.
   6. Increment tot by the number of bytes copied to the block m, the offset off by the number of bytes copied to the block m, and the source buffer src by the number of bytes copied to the block m.
5. If the number of bytes to write is greater than 0 and the offset is greater than the size of the file, update the file size in the inode and return the number of bytes written n. Otherwise, return the number of bytes written n.

**14. tfs\_dup(struct file \*f)**

1. Declare an integer variable fd.
2. Call the fdalloc function passing the file pointer f.
   * 1. If the return value of fdalloc is less than 0, return -1 to indicate an error.
3. Call the filedup function passing the file pointer f.
4. Return the file descriptor index fd to indicate success.

**15. filedup(struct file \*f)**

1. Check if the file reference count is less than 1. If it is, panic.
2. Increment the file reference count by 1.
3. Return the file pointer f to indicate success.

**16. tfs\_close(int fd)**

1. Declare a pointer to a file f.
2. Call the fd\_to\_file function passing the file descriptor fd and a pointer to the file pointer f.
3. If the return value of fd\_to\_file is less than 0, return -1 to indicate an error.
4. Set the file pointer at index fd of the open files array of the current process to 0.
5. Call the fileclose function passing the file pointer f.
6. Return 0 to indicate success.

**17. fileclose(struct file \*f)**

1. Declare a file structure ff.
2. Check if the file reference count is less than 1. If it is, panic.
3. Decrement the file reference count by 1.
   * 1. If the file reference count is greater than 0, return.
4. Copy the contents of the file pointer f into ff.
5. Set the file reference count of f to 0 and its type to FD\_NONE.
6. If the type of ff is FD\_INODE, call the iput function passing its inode pointer.
7. Return.

**18. iput(struct inode \*ip)**

1. Check if the inode reference count is 1 and the number of links is 0.
   * 1. If the condition is true, set the inode type to 0 and call itrunc to truncate and free the inode.
2. Decrement the inode reference count by 1.

**19. tfs\_mkdir(char \*path)**

1. Declare a pointer to an inode ip.
2. Call the create function passing the path path and the file type T\_DIR to create a directory inode.
   * 1. If the create function returns 0 (i.e. an error occurred), return -1 to indicate failure.
3. Return 0 to indicate success.

**20. create(char \*path, short type)**

1. Declare variables off, name, ip, and dp.
2. Call the nameiparent function passing the path path and a buffer name to get the parent directory and the name of the directory/file to be created.
   1. If the nameiparent function returns 0 (i.e. an error occurred), return 0 to indicate failure.
3. Call the dirlookup function passing the parent directory dp, the file/directory name name, and a pointer to off to get the inode of the file/directory if it already exists.
   1. If the dirlookup function returns an inode, check if the inode type is T\_FILE and the requested type is also T\_FILE. If true, return the inode. If false, return 0 to indicate failure.
4. Call the ialloc function to allocate an inode of type type.
   1. If the ialloc function returns 0 (i.e. an error occurred), panic.
5. Set the inode link count to 1.
6. If the inode type is T\_DIR, increment the link count of the parent directory dp.
7. If the inode type is T\_DIR, call dirlink function to add . and .. entries to the new directory inode.
   1. If the dirlink function returns an error, panic.
8. Call dirlink function to add the new directory/file to the parent directory.
   1. If the dirlink function returns an error, panic.
9. Return the new inode.

**21. nameiparent(char \*path, char \*name)**

1. Call the namex function passing the path path, the flag 1 indicating that the function should return the parent inode of the last component in the path, and a buffer name to store the name of the last component.
2. If the namex function returns 0, return 0 to indicate failure.
3. Return the inode pointer returned by the namex function.

**22. namex(char \*path, int nameiparent, char \*name)**

1. If the first character of the path is '/', call iget function passing the ROOTINO constant to obtain the inode for the root directory. Otherwise, call idup function passing the current working directory (cwd) of the current process to obtain the inode for the working directory.
2. Enter a while loop that iteratively calls skipelem function to extract the next element from the path and store it in the buffer pointed to by name. The loop executes until skipelem function returns a null pointer.
3. Check if the type of the inode pointed to by ip is not T\_DIR. If it's not, return 0 to indicate failure.
4. If the nameiparent flag is set and the current element in the path is the last element, return the inode pointer ip.
5. Call dirlookup function passing the inode pointer ip, the current element in the path, and a flag 0 indicating that the function should not check for the presence of a parent directory. Store the return value in the next variable.
6. If the next variable is 0, return 0 to indicate failure.
7. Set the ip variable to the inode pointed to by next.
8. If the nameiparent flag is set after the loop terminates, call iput function passing the inode pointer ip and return 0 to indicate failure. Otherwise, return the inode pointer ip.

**23. inode\* iget(uint inum)**

1. iget(uint inum) is called with an unsigned integer argument 'inum'.
2. Two pointers to struct inodes, 'ip' and 'empty' are declared.
3. 'empty' is initialized to NULL.
4. A loop is started which iterates from 1 to sb.ninodes-1.
5. In each iteration, the pointer 'ip' is set to the address of the 'ino'th inode in the global array 'inodes'.
6. If the inode is already in the cache and has a positive reference count and its inode number matches the passed in 'inum', then its reference count is incremented and it is returned.
7. If an empty slot for an inode is found in the cache, 'empty' is set to point to this slot.
8. If no inode slots are empty, a kernel panic is triggered with the error message "iget: no inodes".
9. A pointer 'ip' is then set to the empty slot found.
10. The 'inum' of this inode is set to the passed in 'inum'.
11. The reference count of this inode is set to 1.
12. The inode pointer 'ip' is returned.

**24. idup(struct inode \*ip)**

1. This function takes a pointer to an inode as input and increments its reference count by one before returning the pointer.
2. The function increments the ref count of the inode by one, indicating that another reference to the inode has been made, and then returns the pointer to the inode.

**24. skipelem(char \*path, char \*name):**

1. Declare and initialize a pointer to a character s, and an integer len.
2. Use a while loop to skip all leading '/' characters in the path string.
3. Check if the first character in the path string is '\0' (end of string). If it is, return 0.
4. Set the s pointer to the current position in the path string.
5. Use another while loop to find the end of the current path element (the next '/' character or the end of the path string).
6. Calculate the length of the current path element by subtracting the position of s from the position of path.
7. If the length of the path element is greater than or equal to DIRSIZ (a pre-defined constant), use memmove to copy the first DIRSIZ characters of the path element to the name buffer.
8. If the length of the path element is less than DIRSIZ, use memmove to copy the entire path element to the name buffer and add a '\0' character to the end.
9. Use another while loop to skip any additional '/' characters in the path string.
10. Return a pointer to the start of the next path element in the path string.

**25. dirlookup(struct inode \*dp, char \*name, uint \*poff):**

1. Declare a variable off of type uint to keep track of the current offset in the directory, and a variable inum of type uint to hold the inode number of the file with the matching name.
2. Declare a variable de of type struct dirent to hold the directory entry being read.
3. Check if the type of the directory inode dp is not T\_DIR (a pre-defined constant indicating a directory inode). If it's not, panic with an error message.
4. Use a for loop to iterate over all directory entries in the directory inode.
5. Read the directory entry at the current offset off using the readi function, which reads from the file system's underlying storage device.
6. Check if the read was successful by comparing the size of the read data to the size of a directory entry. If they are not equal, panic with an error message.
7. Check if the inum field of the directory entry is 0. If it is, skip to the next directory entry.
8. Compare the name parameter with the name field of the directory entry using the namecmp function, which returns 0 if the names match. If they match, continue with the next steps.
9. If the poff parameter is not NULL, set the value it points to (\*poff) to the current offset off.
10. Set the inum variable to the value of the inum field of the directory entry.
11. Call the iget function to get the inode for the file with the matching name. The iget function returns the inode with the given inode number and adds it to the inode cache if it's not already there.
12. Return the inode of the file with the matching name.
13. If no directory entry with the matching name is found, return 0.

**26. ialloc(short type):**

1. Use a for loop to iterate over all inode numbers in the inode table (inodes).
2. Check if the type parameter matches the type field of the inode. If they match, continue with the next steps.
3. Check if the type field of the inode is 0, which indicates a free inode. If it is, continue with the next steps.
4. Use memset to set all bytes of the inode to 0.
5. Set the type field of the inode to the value of the type parameter.
6. Set the inum field of the inode to the current inode number.
7. Get the current time using the time function and store it in the seconds variable.
8. Copy the first 4 bytes of the seconds variable to the c\_time array in the inode's ctime field.
9. Return a pointer to the newly allocated inode.
10. If no free inode is found, panic with an error message and return 0.

**27. dirlink(struct inode \*dp, char \*name, uint inum):**

1. Declare an integer variable off to keep track of the current offset in the directory, a struct dirent variable de to hold the directory entry being read or written, and an struct inode variable ip to hold the inode of the file with the given name.
2. Call the dirlookup function to check if a file with the given name already exists in the directory. If it does, release the inode and return -1.
3. Use a for loop to find an empty directory entry in the directory. The loop iterates over all directory entries in the directory until an entry with a 0 inum field is found.
4. Read the directory entry at the current offset off using the readi function, which reads from the file system's underlying storage device.
5. Check if the read was successful by comparing the size of the read data to the size of a directory entry. If they are not equal, panic with an error message.
6. Check if the inum field of the directory entry is 0. If it is, break out of the loop, as an empty directory entry has been found.
7. Copy the name of the file (name) to the name field of the de directory entry using strncpy.
8. Set the inum field of the de directory entry to the given inum value.
9. Write the de directory entry to the directory at the current offset off using the writei function, which writes to the file system's underlying storage device.
10. Check if the write was successful by comparing the size of the written data to the size of a directory entry. If they are not equal, panic with an error message.
11. Return 0 to indicate success.

**28. tfs\_open(char \*path, int flags, int mode):**

1. Declare an integer variable fd to hold the file descriptor of the opened file, a struct file pointer f to hold the file structure for the opened file, and a struct inode pointer ip to hold the inode of the file being opened or created.
2. Check if the TO\_CREATE flag is set in the flags parameter. If it is, call the create function to create a new file with the given path and file type T\_FILE (a pre-defined constant indicating a regular file inode type). If the create function returns 0, indicating failure, return -1.
3. If the TO\_CREATE flag is not set in the flags parameter, call the namei function to get the inode of the file with the given path. If the namei function returns 0, indicating failure, return -1.
4. Check if the type field of the inode ip is T\_DIR (a pre-defined constant indicating a directory inode type) and the flags parameter is not TO\_RDONLY (a pre-defined constant indicating read-only mode). If they are, return -1, as directories can only be opened in read-only mode.
5. Call the filealloc function to allocate a new file structure. If the function returns 0, indicating failure, or call the fdalloc function to allocate a new file descriptor. If the function returns a negative value, indicating failure, release the allocated file structure and return -1.
6. Set the type field of the file structure f to FD\_INODE (a pre-defined constant indicating an inode-based file descriptor), set the ip field to the inode ip, set the off field to 0, set the readable field to true if the TO\_WRONLY flag is not set in the flags parameter, and set the writable field to true if the TO\_WRONLY or TO\_RDWR flags are set in the flags parameter.
7. Return the file descriptor of the opened file.

**29. filealloc():**

1. Declare a pointer f of type struct file.
2. Use a for loop to iterate over all file structures in the file table (ftable.file).
3. Check if the ref field of the file structure pointed to by f is 0, which indicates an unused file structure. If it is, set the ref field to 1 to mark it as used and return the file structure pointer f.
4. If no unused file structure is found, return 0 to indicate failure.

**30. tfs\_chdir(char \*path):**

1. Declare a pointer ip of type struct inode to hold the inode of the directory being changed to.
2. Call the namei function to get the inode of the file with the given path. If the function returns 0, indicating failure, return -1.
3. Check if the type field of the inode ip is not T\_DIR (a pre-defined constant indicating a directory inode type). If it's not, return -1, as only directory inodes can be used as current working directories.
4. Release the current process's current working directory inode by calling iput on the cwd field of the curr\_proc structure.
5. Set the cwd field of the curr\_proc structure to the inode ip.
6. Return 0 to indicate success.

**31. print\_inodes():**

1. Use a for loop to iterate over all 32 inodes in the inodes array.
2. For each inode, print its ref, type, size, inum, and ctime fields using printf and the corresponding format specifiers %x (for hexadecimal), %d (for decimal), and %x (for hexadecimal).
3. Print a newline character to separate the output for each inode.

**32. writefsinfo():**

1. Declare a buffer buf of size BSIZE (a pre-defined constant indicating the size of a disk block).
2. Use memset to clear the buffer buf.
3. Use memcpy to copy the superblock structure sb to the buffer buf.
4. Call the bwrite function with the disk block number 1 and the buffer buf. The bwrite function writes the contents of the buffer to the corresponding block on the underlying storage device.
5. Use memset to clear the buffer buf.
6. Use memcpy to copy the inode bitmap inodebitmap to the buffer buf.
7. Call the bwrite function with the disk block number 2 and the buffer buf.
8. Use memset to clear the buffer buf.
9. Use memcpy to copy the data bitmap databitmap to the buffer buf.
10. Call the bwrite function with the disk block number 3 and the buffer buf.
11. Use a nested for loop to iterate over all inodes in the inodes array.
12. For each group of 8 inodes (i.e., each disk block group), use memset to clear the buffer buf.
13. Use memcpy to copy the next 8 inodes (starting from j+i\*8) in the inodes array to the buffer buf.
14. Call the bwrite function with the disk block number i+4 and the buffer buf.
15. Check if the return value of bwrite is less than 0. If it is, panic with an error message.

**33. bfree(uint bi):**

1. Calculate the index of the bit corresponding to the given block index bi in the databitmap array by taking the integer division of bi by 32.
2. Calculate a bitmask m corresponding to the bit at the calculated index by shifting the value 1 by bi % 32 bits to the left.
3. Check if the value of the databitmap array at the calculated index ANDed with the bitmask m is 0. If it is, panic with an error message, as the block is already free.
4. Use a bitwise AND with the bitwise NOT of the bitmask m to clear the bit at the calculated index in the databitmap array, effectively marking the block as free.

**34. itrunc(struct inode \*ip):**

1. Use a for loop to iterate over all direct block pointers in the blocks array of the inode ip.
2. For each non-zero block pointer, call the bfree function with the block index to free the corresponding block on the underlying storage device.
3. Set the size field of the inode ip to 0 to indicate that the inode is now empty.
4. The iupdate function is commented out and not called, as the inodes on disk are updated when the function exits.

**35. stati(struct inode \*ip, struct tfs\_stat \*st):**

1. Set the ino field of the st structure to the inode number inum of the inode ip.
2. Set the type field of the st structure to the type field of the inode ip, which indicates the type of file the inode represents (e.g., regular file, directory, symbolic link).
3. Set the nlink field of the st structure to the nlink field of the inode ip, which indicates the number of hard links to the file.
4. Set the size field of the st structure to the size field of the inode ip, which indicates the size of the file in bytes.

**36. namecmp(const char \*s, const char \*t):**

1. Use the strncmp function to compare the first DIRSIZ bytes of the strings pointed to by s and t.
2. Return the result of the comparison as the return value of the namecmp function. If the first DIRSIZ bytes of the two strings are equal, strncmp returns 0; otherwise, it returns a non-zero value.

**37. namei(char \*path):**

1. Declare a character array name of size DIRSIZ (a pre-defined constant indicating the maximum length of a directory entry name).
2. Call the namex function with the path argument, a value of 0 for the wantparent argument (indicating that the function should return the inode of the final path component, not its parent directory), and the name array as the name argument.
3. Return the value returned by the namex function.

**38. createfs(char \*name, uint blks, uint dblks, uint inds):**

1. Open a file with the given name in read-write mode with O\_CREAT and O\_TRUNC flags to create the file if it doesn't exist and truncate it to zero length.
2. If the file descriptor returned by the open system call is negative, panic with an error message.
3. Declare a character array b of size BSIZE (a pre-defined constant indicating the size of a disk block) and a pointer c to b.
4. Initialize b to all zeros using the memset function.
5. Copy the string "Block 0 - Not used." to c.
6. Write the contents of b to the file using the write system call, and store the return value in sz.
7. If sz is negative, panic with an error message.
8. Initialize b to all zeros using the memset function.
9. Use a for loop to iterate over all block indices from 1 to blks (inclusive).
10. If the block index is greater than 7 (indicating a data block), set the first byte of b to the block index.
11. Write the contents of b to the file using the write system call, and store the return value in sz.
12. If sz is negative, panic with an error message.
13. Declare a struct superblock sb and initialize its fields with the given blks, dblks, inds, and name values.
14. Initialize the nlog field of sb to 0, and set the name of the file system to the given name value by copying it to sb.name.
15. Copy the contents of sb to b using the memcpy function.
16. Write the contents of b to the file system using the bwrite function with the block index 1 (the first block is reserved for the superblock).
17. If the return value of bwrite is negative, panic with an error message.
18. Close the file using the close system call.
19. Return 0 to indicate success.

39. openfs(char \*name):

1. Open a file with the given name in read-write mode with the O\_RDWR flag.
2. If the file descriptor returned by the open system call is negative, panic with an error message.
3. Return 0 to indicate success.

**40. closefs():**

1. Close the file system file descriptor fs using the close system call.
2. Return 0 to indicate success.

**41. get\_opts(int count, char \*args[]):**

1. Initialize variables opt, len, i, and good.
2. While good is true and the getopt system call returns a valid option:
   1. Depending on the option, do the following:
   2. If the option is -s, check that the value is a number. If it is not a number, print an error message and set good to 0. Otherwise, convert the value to an integer using the atoi function and store it in the variable start\_block.
   3. If the option is -l, check that the value is a number. If it is not a number, print an error message and set good to 0. Otherwise, convert the value to an integer using the atoi function and store it in the variable blocks.
   4. If the option is :, print an error message indicating that the option is missing a value.
   5. If the option is ?, check the value of optopt to determine which option caused the error. Print an error message accordingly and set good to 0.
   6. If good is still true and the optind variable is greater than or equal to count - 1, print an error message indicating an invalid number of arguments and set good to 0.
   7. Otherwise, if good is still true, copy the value of the argument at index optind to the filename variable.
3. Return good to indicate whether the function succeeded or failed.