System-level Programming

Avery Karlin

Fall 2015

Contents

Teacher: Dyrland-Weaver

1 Learning C

1.1 C Primitive Variable Types

1. All C primitives are numeric, divided purely based on variable size, and integer or floating point
   (a) C variables have sizes based on the platform they were compiled by and for, such that sizeof(*type*) can be used to determine the size in bytes
   (b) On a standard computer, int = 4 ($-2^31, 2^31 - 1$), short = 2, long = 8, float = 4, double = 8, and char = 1 bytes (8 bits to a byte)
   (c) Types can also be specified as unsigned, such that it is not able to be given a negative value
   (d) Types can be placed within types of larger size and the same format without any form of conversion, but not of a different format
   (e) sizeof(*type*) returns the size in bytes of the type
2. Boolean values are numbers, such that 0 is false, and all nonzero numbers are considered true
3. Character literals can be represented inside single quotes rather than use a number, and Strings, though not an object, can use a double quotes literal
   (a) Strings are created by character arrays, using a null character (value 0), to show the end of the array, allowing it to be modified easier
4. Variables are able to be initialized within a for loop, but are not able to be declared, such that it must be before the loop
5. "typedef *var-type new-name*" allows you to call an existing variable by a different name
   (a) Typedef is typically used within the header to allow the term to be used throughout the program

1.2 C Compilation

1. They are compiled through "gcc *file.c* -o *program_name*", then run through "./*program_name*"
2. C programs can be compiled into one executable, such that there is just one file after another in command line, though since they are compiled in the same namespace, they cannot have the same global variables or function names
3. The same is true for files called within other files, though those are compiled automatically
4. C programs must have exactly 1 main() function to compile, creating issues when multiple files have a main()
5. "gcc -c *file.c*" creates a binary object .o file, which can be run through other exectuable files, and can be compounded similarly during compiling
   (a) Binary object files are used for files without a main method, and are non-executable
   (b) Binary object files can be linked to a .c file through compound compiling with the first file as the .c file

(c) Thus, the .o file is dependant on all .h files and the .c file, while the executable is dependent on the .o files, .h files, and the main .c file

(d) The .o file can then be compiled identically to .c files

6. If "-o *program_name*" is not put in, the default executable name is "a.out"

7. Projects with large numbers of dependencies can have a file called "makefile" to organize creation of the executable with dependencies

   (a) The makefile is ordered from the most dependent level to least dependent (ignoring non-dependent files and standard libraries)

   (b) Format is:
   Target-Filename: dependency1 dependency2
   <tab> *Terminal compilation code for file*

   (c) "make *target-filename*" command is used when in the directory to compule the project using the make file

   (d) The clean targets can be used to remove files from the directory after compilation

   (e) The all target can be used to compile the entire program

   (f) Any bash commands can have a makefile target made, such that it can be used with a "run" target to run commands faster (though any target can be used)

   (g) The first target in a makefile is the default if just "make" is written in terminal

1.3 C Programming

1. All C programs are made up of a series of functions, run within the main function, which returns an integer (typically 0, or other values for errors) item Libraries are added, either .h files from the current directory through #include "*file*.h" or through premade libraries by #include <file.h>

   (a) All files typically start with calling the C library with #include<stdio.c> (standard io) and <stdlib.h> (standard library)

2. The man pages, called by "man *command*" or "man *section command*", give information on both bash and C commands

   (a) (1) is user commands, (2) is system calls, (3) is library functions, such as the C libraries, (4) is devices, (5) is file formats, (6) is games and amusements, (7) is conventions and miscellany, and (8) is system admin and priveledged commands

   (b) (L) is used for local commands, installed by certain programs

3. C functions are pass by value, such that they put the value into a new variable created by the function, though if pointers are passed, it is equivelent to pass by reference, due to being a ppointer to the same location

   (a) C functions are written similar to java, with the exception of the lack of the protection

   (b) Due to C being functional, the functions are created in the order written, such that it should already have created all functions and commands used within the function being compiled

(c) Failure to declare first leads to an implicit declaration warning that it has not been formally declared yet, though it will still work if it is declared later

(d) Headers can also be placed at the top of the function in addition to where they are defined, to avoid implicit declaration, or in a seperate header file

1.4 C Structures

1. "printf(*text, var1, var2*)" is used to print a String in terminal, where the text is a formatted string, with placeholders for variables following

   (a) %f is a placeholder for a float, %d for double, %c for char, %s for string, %f for pointer, %lf for double, %ld for long, and %d for int

   (b) println can be used instead of printf for non-formatted strings (without variables)

   (c) Print functions do not automatically add "
   n" at the end of a line, and must be written in the string

2. Arrays in C are non-dynamic, such that they must have a fixed size, with no length function, and there are no errors for going outside boundries, rather going to a different point in memory

   (a) Arrays are declared by "*type*[*size*];" and must be initialized each part at a time

3. String functions are held within the string.h library, always assuming the strings are null-terminated

2 Memory Management

2.1 Memory Allocation

1. Memory allocation is either during compile time (static stack memory), or during runtime (dynamic heap memory)

2. Compiler allocated memory is packaged within the binary, unable to be overwritten by other programs due to protected memory, without a default value, where variables and arrays are allocated

   (a) Memory addresses of variables are fixed once they are placed, such that the data can be changed, but the location cannot be

   (b) Variable names are not stored, but rather substituted for memory locations during compilation

   (c) Once the function/scope under which the stack memory created is finished, the memory is automatically released

3. Systems have a bit limit which they can read at once, such that 32 bit systems are limited to 32 bit unsigned values, such that $[0, 2^32 - 1]$ is possible, or 4 GB

4. Pointers are variables designed to store memory addresses, stored within the stack

   (a) %*variable* is used to get the address of a variable, such that the number returned can be the value of a pointer

(b) When a pointer is incremented, the location moves the number of bytes of the variable type which the pointer applies to

(c) * is used before a variable name to declare a pointer, and is also used when calling a variable to get the value of the item at that location, preceeding before numeric operators except ++ and –

(d) Thus, for some array a, with *a as the pointer, a[i] = *(a + i)

(e) Null pointers (set equal to the location 0) are often used to signify the end of a list

5. Other sections in memory are code (the executable section of an object file, often stored in ROM until run), bss (global or static uninitialized variables), and data (global or static initialized variables)

6. Variables marked by the "const" keyword have the value stored in the read-only portion of the same memory space they would normally exist in

7. String literals are generally stored within the code section, though it can occasionally be stored in the data section as a static variable

2.2 Strings and Arrays

1. Strings can be declared by several methods, "char *str[byte_num]*" to do basic allocation, or it can be set on the same line, with a null put in the byte after the last letter

   (a) It can also be declared with an empty byte number, but set such that it will be given the exact amount of space needed

   (b) It can also be declared as a pointer to the array by "char **str* = *data*", created the array the exact correct size, and a pointer to the array under that variable name

   (c) After declaration, each character must be set individually, instead of using the equal sign

   (d) On the other hand, if a pointer is used, the pointer can be changed to apply to a seperate array, using an equal sign, even after declaration

2. The null character at the end is needed for string functions in string.h to work correctly, but is not a requirement

3. String/array variables are functionally immutable pointers to the first item in an array (such that the location cannot be changed)

4. Pointer-defined strings are literals, such that they are made in protected memory, where the pointer location can be redefined, but the string cannot be

   (a) Literals of the same string will point at the same location as previously made literals, stored fully in static stack memory

2.3 Dynamic Memory Allocation

1. (*type* *)malloc(int *byte-num*) allocates that number of bytes from the head, returning the location of the first byte, typecasting the pointer to the type specified

   (a) sizeof(*type*) is often used to allocate the correct amount of memory

   (b) calloc is a similar function that sets each bit to 0, otherwise acting like malloc, though with a first parameter to determine the number of data pieces created
   (c) malloc returns a void pointer, such that it can be typecast to any type of pointer

2. realloc(void *p, int *new-byte-num*) will return any extra bytes, or add additional bytes to the allocation

3. Normal memory allocation happens on the stack, automatically released after the function which created it ends

4. Dynamic memory allocation happens on the heap, kept even after the function that created it is removed from the stack, such that it must be released
   (a) free(*pointer*) releases the dynamically allocated memory which the pointer goes to
   (b) Dynamic memory should always be released in the program when created, and can prevent filling the memory
   (c) After the program is ended, the memory is freed automatically, but it can freeze the computer if filled before then, such as in an infinite recursion

2.4 Shared Memory

1. Shared memory is a segment of heap memory able to be accessed by multiple processes, not released when a program exits, accessed by a specific key

2. Shared memory functions are within "sys/shm.h", "sys/ipc.h", and "sys/types.h"

3. Shared memory can be created or removed, as well as accessed or attached/deattached from a variable once per process

4. shmget(*key, size, flags*) - Creates or accesses a shared memory segment, where key is a unique identifier
   (a) Size is the number of bytes requested
   (b) Flags are the permissions, able to be combined by bitwise OR with IPC$_CREAT(createsthesegmentifitdoesn'$
   $1/errnoifitfails$)

5. ftok(*path, x*) - Generates a unique key, based on the path to some accessible file, and an integer
   (a) The generated key will be the same for the same path and integer, each time run

6. shmat(*descriptor, pointer, flags*) - Attaches the shared memory to the pointer, or to available memory space if null pointer is given
   (a) Returns the pointer or -1/errno if it fails
   (b) SHM_RDONLY attaches the shared memory as read-only, such that the pointer value cannot be modified

7. shmdt(*pointer*) - Deattaches pointer variable from shared memory, such that while the data is still in shared memory, it can no longer be accessed in the program

8. shmctl(*descriptor, command, buffer*) - Controls shared memory by the command (IPC_STAT for populating buffer struct with metadata information, IPC_SET for setting segment info to buffer, or IPC_RMID for removing segment)

    (a) Meta-data is stored within a "struct shmid_ds" containing last access, size, PID of creator, PID of last modification, etc

3 Structural Functions

3.1 String Functions

1. String functions are found within ¡string.h¿, assuming a null character at the end

2. int strlen(char *s) returns the length of s, ignoring the null character

3. int strcmp(char *s1, char *s2) returns 0 if equal, ¿0 if s1 ¿ s2, and ¡0 otherwise

4. char* strcpy (char *destination, char *source) copies the string to destination, assuming the allocated destination space is the same size or larger

5. char* strcat (char *destination, char *source) adds source to the end of destination

6. strncat and strncpy has an integer as a final parameter, using only the first n characters of the source string, such that if it is longer than the string, it uses up to the null character

7. char* strsep(char* source, char* delimeter) - Replaces each occurance of the delimeter, replacing the first character of the first occurance of the delimeter with NULL

    (a) The source must be a pointer to the string pointer (not array, due to needing to be mutable) variable, and is modified such that it points to the character after the delimeter, or null if there is no instance

    (b) Returns the original source, before changed

8. sprintf(*char* str, char* format, var1, var2,...*) - Converts the variables into a string of the format specified by the format string, made up of placeholders, stored within str, returning the number of characters in the string

3.2 Struct

1. Structs are a collection of values within a single data type, declared by struct{*type1 var1; type2 var2*} as the type name

    (a) Typedef is used to create a simple type name for it

    (b) If the struct type itself is needed within the struct, it can be declared implicitly within the typedef, by putting "struct type-name var" instead of just the type

2. *struct-name.subvar* is used to call a specific item within the struct

3. Since . operator has precedence over *, pointers to structs either must get the struct data before getting a specific piece of data, or use "*pointer-to-struct -¿ struct-var* to do

3.3 Union

1. Union data types are defined similarly to structs, using the keyword "union" instead, used to store multiple types of data within the same memory
2. Unions only set the maximum size to the largest of the possible types, such that the data is used similarly to a struct item
3. Unions are only able to store a single type of data at a time

4 File Programming

4.1 Permissions

1. There are 3 permission areas, each with their own permission value, first the creator (user), then a specific group of users (group), then everyone else (others), each mutually exclusive
   (a) The owner always has the ability to delete and change the permissions of a file, even if the permission value is 0
2. ABC is the permissions of an area in binary, where A = read, B = write, C = execute, such that it can be converted into a number 0-7 in octal (which must be written in code with a 0 first, to tell the compiler it is octal)

4.2 Bitwise Operators

1. & is the bitwise AND operator, — is OR, is NOT, and îs XOR
2. Bitwise operators are used to modify the actual binary of equal lengths, going bit by bit

4.3 File Usage

1. The file table is a list of all files used by a program while it it running, containing basic information such as location and size
   (a) The file table has a limited size of 2^n, typically 256, where getdtablesize() in "unistd.h" returns the size value;
   (b) Each file is given a descriptor, or an integer index from 0, and the table records the path, location, and other data
   (c) File descriptor 0 (or STDIN_FILENO) always refers to stdin (standard command line input), and 1 (or STDOUT_FILENO) refers to stdout
   (d) File descriptor 2 (or STDERR_FILENO) refers to stderr (standard error), which contains all error messages produced by the compiler, similar to stdout
   (e) These are automatically opened on the opening of a C program
2. open(*file_path, flags, mode*) - Opens/adds the file to the first open file table space, returning the file descriptor
   (a) File descriptor -1 is returned when the file opened does not exist, or there is not permission to access the file

(b) When open() fails, the variable "errno" in "errno.h" is automatically set for the specific type of error

(c) strerror(*int*) in "string.h" returns a string describing the integer stored in errno

(d) Found within "fcntl.h"

(e) Flags are used to declare what the file is being used for, while mode is only used if creating the file, setting the permissions as an octal number

(f) Flags can be:

- O_RDONLY - Read only
- O_WRONLY - Write only
- O_RDWR - Read and Write
- O_APPEND - Write to end of file only
- O_TRUNC - Erase file and write over
- O_CREAT - Create file and open it if it exists
- O_EXCL - Can be combined with O_CREAT to return an error if it exists

3. close(*file_descriptor*) removes the file from the file table, found within "unistd.h"

4. read(*file_descriptor, buffer, amount*) reads text from a file, where buffer is a pointer to where the text is placed, and amount is the bytes read

   (a) Returns the number of bytes read, or -1 if it fails (setting the errno value)

5. unmask(*mask*) sets the file creation permission mask, found within "sys/stat.h"

   (a) Files are not initially given the permissions in the mode argument when opened, when they are created, such that a mask must be applied to modify

   (b) The new permissions when opened are thus mask & mode, shutting off all permission from the mask

   (c) The mask should be in octal form, adding a new mask to the permissions

6. lseek(*file_descriptor, offset, whence*) sets the current position in an open file, where offset is the number of bytes to move by

   (a) Whence can either be SEEK_SET (beginning of the file), SEEK_CUR (current position in the file), or SEEK_END (end of the file)

   (b) Returns the number of bytes from the current position to the start of the file, or -1 in case of error, setting errno

   (c) Found within "unistd.h"

7. write(*file_descriptor, buffer, amount*) writes to the file the data from the buffer (which must be a pointer), where amount is the number of bytes written

   (a) Returns the int number of bytes written

   (b) Found within "unistd.h"

4.4 Metadata

1. Metadata is information stored in the filesystem outside of files
2. "struct stat" is a built-in struct for metadata, where the stat(*path, stat_buffer*), where the stat buffer is the pointer to the stat struct, setting the struct to refer to the file
3. Stat struct fields include:
   - st_size - int, file size in bytes
   - st_uid - int, user id
   - st_gid - int, group id
   - st_mode - int, file permissions
   - st_atime - time_t, last access time
   - st_mtime - time_t, last modification time

4.5 Directories

1. Directory streams are stored within the DIR variable, and its pointer, stored along with their functions in "dirent.h"
2. opendir(*path*) - Returns a pointer to the directory stream, but does not change the current working directory (cwd)
3. closedir(*directory_stream*) - Closes the directory, and frees the stream variable automatically
4. readdir(*directory_stream*) - Returns a pointer to the next entry within the directory stream, or null if the stream is at the end
5. rewinddir(*directory_stream*) - Moves the directory stream to the start
6. "struct dirent" is a built-in struct for entries within a directory streams, stored within "sys/types.h"
   - d_name - char[], name of files within the directory
   - d_type - int, file type

4.6 Inputs

1. scanf(*format_string, var1, var2, ...*) - Takes a string from stdin, matching the items from the string to variables within the format string (made up of placeholders), stored within the pointers, var1, var2, ...
   (a) sscanf(*string, format_string, var1, var2, ...*) - Takes the string, scanning data from it similarly to scanf
2. fgets(*string_pointer, bytes, file_pointer*) - Reads data from the file pointer to the string pointer
   (a) It stops after writing the byte number - 1 (giving space for 0), first newline, or the end of file character, placing 0 (null at the end)
3. "int argc, char *argv[]" can also be put as the parameters of the main() function, to allow command line arguments, starting from the bash commands themselves

5 Processing

5.1 Processes

1. Each running program is a process, such that each core in a processor can handle one process per cycle, only appearing to multitask due to the high speed
2. Each process has a unique PID, recycled when the machine is restarted
3. PIDs can be stored in pid_t type variables, found in "sys/types.h", displayed as an integer number

5.2 Process Functions

1. getpid() - Returns the current PID, stored within "unistd.h"
2. getppid() - Returns the current process's parent PID, stored within "unistd.h"
3. sleep(*int seconds*) - Makes the processor rest for that many seconds, stored within "stdlib.h"
4. fork() - Stored within "unistd.h", creating a child subprocess from the parent
   (a) All aspects of the parent, including memory and file table are copied to the child process when it is forked, continuing from the same line in the child
   (b) Threads are different from child processes, such that they do not have their own copied memory, rather sharing the same memory
   (c) 0 is returned to the child, while the child PID is returned to the parent
   (d) If it fails, -1 is returned and errno is set
5. wait(*&int status*) - Waits in the parent process until at least one child process has returned information
   (a) status stores a return value as the first byte from the process (not the C return, but the process return), followed by other info, such that WEXITSTATUS(status) gives only the return value
   (b) Returns the PID of the child exited, or -1 (setting errno) if it fails
6. waitpid(*pid, &status, options*) - Waits for a specific chid to complete, where options defaults to 0 for normal behavior, and where -1 as the pid waits for any child
7. exit(*int return*) - Ends the process and returns the process return value, found in "unistd.h"

5.3 Exec Function Family

1. Exec functions can be used to run other programs within programs, such that the other program replaces the program within the process, such that code after is ignored
2. Exec functions are found within "unistd.h"
3. execl(*path-to-including-command, command, arg1, arg2, ...,*, NULL) - Runs the command program, found at the path, where all non-NULL variables are strings, args are command line arguments, and NULL shows the end of the arguments

4. execlp(*path, command, arg1, arg2, ...,*, NULL) - Runs similar to execl, except using the PATH variable for the start of the path

5. execvp(*path, argument-array*) - Runs similar to execlp, except with an array of the command and command line argument strings

6 File Redirection

6.1 Command-Line Redirection

1. *command* > *new_output* - Redirects stdout from the command into the new_output file, overwriting the file

2. >> - Redirects, but appends instead of overwrites

3. 2 > and 2 >> - Redirects stderr instead of stdout

4. & > and & >> - Redirects both stdout and stderr

5. *command* < *new_input* - Redirects stdin from the file into the command

6. *command1* — *command2* - Pipes from command1 to command2, or puts the stdout of command1 as the stdin of command2

6.2 C Redirection

1. dup2(*fd1, fd2*) - Redirects the fd2 file descriptor value to refer to fd1, found in "unistd.h"

2. int dup(*fd1*) - Creates a new entry in the file table, the decriptor of which is returned, which refers to the same file as fd1, found in "unistd.h"

7 Signaling

7.1 Command-Line Signals

1. Signals are a limited way of sending information to a process

2. kill ¡PID¿ - Command line utility to send signal 15 (SIGTERM) to terminate the PID

   (a) kill ¡SIGNAL¿ ¡PID¿ - Sends the specified signal instead, such that the process may not be terminated

   (b) killall ¡PROCESS_NAME¿ - Sends the terminate signal to all processes with PROCESS_NAME in the name

   (c) killall ¡SIGNAL¿ ¡PROCESS_NAME¿ - Sends the specified signal to those processes

7.2 Signals in C

1. All signal functions are found within "signal.h"

2. kill(*PID, SIGNAL*) - Sends the signal to the process, returning 0 if succeeding, -1 (with errno) otherwise

3. signal(*SIGNUMBER*, sighandler) - Attaches the signal to the signal handling function, such that it is redirected to the function
    (a) Signal handling function must have the header "static void sighandler(int signo)"
    (b) Sighandler is static, such that it can only be called from the function it is defined in
    (c) Some signals, such as SIGKILL, cannot be attached to the sighandler function

7.3 Semaphores

1. Semophores are IPC (interprocess communication) used to control access to shared resources, to prevent editing errors
    (a) These act as a counter to determine the number of slots for accessing a process, often used in pairs (one for read, one for write)
    (b) Semaphores can be set, created, removed, incremented (either signal(), P(), S++, or Up(S) as non-code notation), decremented (wait(), V(), S–, or Down(S))
    (c) Semaphores are not connected to any particular memory, but rather cause the program to wait until the change in the value is possible (without ending less than 0)
2. Semaphore functions require "sys/types.h", "sys/ipc.h", and "sys/sem.h" to function
3. semget(key_t key, int amount, int flags) - Creates/accesses a semaphore, returning the descriptor or -1/errno if it fails
    (a) The key is the unique semaphore indentifier, determined by ftok()
    (b) Amount is the number of semaphores within the set, while flags are the permissions for the semaphore
    (c) Using the bitwise OR, IPC_CREAT (creates semaphore, setting to 0), and IPC_EXCL (fails if IPC_CREAT is used and semaphore exists) can be added to the flags for if being created
4. semctl(*semaphore-descriptor, index, operation, data*) - Controls the semaphore
    (a) Index is the index of the semaphore within the set/array identified by the descriptor, while data is only if a specific value is inputted to the semaphore operation
    (b) This can be used to set, remove, get the value, and get information about the semaphore
    (c) Operations include IMP_RMID (remove), SETVAL (set), SETALL (sets all semaphores within the set), GETVAL (returns value), IPC_STAT (populates buffer with information about the semophore from *data*)
    (d) Data is a union semun data type, able to store the data for any of the possible uses
        i. The union is not included in the header, such that it must be written as: "union semun int val; struct semid_ds *buf; unsigned short *array; struct seminfo *__buff;;
5. semop(*descriptor, operation, amount*) - Modifies the value of the semaphore, as an atomic operation that the memory can not be seen changed until completion
    (a) The amount is the number of semaphores in the set which are being operated on

(b) Operation is a pointer to a "struct sembuf", precreated, containing short sem_flg, short sem_num (index of semaphore), and short sem_op (-n for down by n, n for up, 0 to have the process sleep until the semaphore is 0)

(c) Possible values for sem_flg are SEM_UNDO (allows the OS to release the semaphore automatically in case the program crashes, and was unable to release it), IPC_NOWAIT (if the semaphore is unavailable, it returns an error instead of waiting)

8 Connections

8.1 Pipes

1. Pipes are a conduit between two processes, with each pipe containing a read and write end, unidirectional (constant direction), acting like files to transfer data

2. Unnamed pipes have no external identification, such as a name or descriptor

   (a) pipe(*int descriptors[2]*) - Creates an unnamed pipe, found within "unistd.h", returning 0 if created, -1/errno if it fails

(b) The function stores in the descriptors array, the descriptors of the read end and the write end (in that order) of the pipe, such that the data written is stored in the pipe until read

3. Named pipes, also called FIFO (first in, first out), have a name that can be used to identify them from another program, unidirectional as well

   (a) mkfifo(*char* name, int permission*) - Creates a named pipe, requiring "sys/types.h" and "sys/stat.h", returning 0 if success, -1/errno if it fails

   (b) The FIFO acts as a file, with permissions inputted, blocking on open (stopping the process from writing or reading) until a connection is established on both ends

   (c) The FIFO is not opened immediately after created, and is used otherwise the same as any other file, though will block extra processes from allowing reading or writing

   (d) remove(*char* path*) - Deletes the given file, used to remove the named pipe conduit after connection is established, returning 0 if success, -1/errno otherwise

   (e) The pipe file descriptor is still valid for use after the conduit is removed, such that communication is still possible, but no new connections can be created

8.2 Server-Client Protocol

1. The handshake procedure is used to ensure a connection is established, first created a named pipe, called a well-known pipe, by the server, which then waits for a connection

   (a) The client then creates a named pipe, called a private pipe, which connects to the well-known pipe, and sends the name of the private pipe, after which it waits for the connection

   (b) The server then recieves the name of the private pipe, removes the well-known pipe, which then connects to the private pipe

2. The basic server uses the handshake, followed by the connection, then after the client exits, the private pipe is removed and a new well-known pipe is created

3. The forking server allows a subserver for each client, allowing for multiple client connections simultaneously

   (a) The handshake is done similarly until the point of connecting to the private pipe by the server, at which point it forks a subprocess, and has that connect, finishing the handshake process
   (b) The variable to the location of the original well-known pipe is then kept by the subserver, but closed in the main server, to allow the new well-known pipe to be created
   (c) After the connection, the private pipe is removed in the subserver, and the subprocess ends
4. Central dispatch server maintains communication with each of the clients simultaneously through maintaining pipes to each of the subservers, allowing communication with clients