Maiah Pardo, mapardo
11/23/21

DESIGN for Asgn3: httpproxy

Files and Structs:

- Queue.c
  - Enqueues and Dequeues
- List.c
  - Note: This is my own List.c file that I made in CSE 101 in Fall 2020. I updated it with a few new functions, for the purpose of this assignment. I will only discuss the functions in List.c that I use in httpproxy.c here.

  - Data structs:
    - The List.c file contains a File data struct called file_t which contains the internal variables:
      - char filename[20]; // Stores the filename
      - char *contents; // Stores the file contents
      - char date[500]; // Stores the modification date of file
      - size_t filesize; // The size of the file contents (in bytes)
    - Node struct which contains data of type file_t, and next and prev pointers to Nodes.
    - List struct which contains pointers to the front of the list, back of the list, a cursor (can point to any node of my choosing), an index (an integer of the index of the cursor), and length (an int) consisting of the number of elements in the List.

  - length(List L):
    - Returns the number of elements in L
  - Find(List L, char* filename):
    - Given a filename, returns the node in List L that contains the filename and sets the internal cursor to this node. If node is not found, returns NULL.
  - addToContents(List L, uint8_t Buffer, size_t BufferSize, int idx, int filesize):
    - Adds to the node pointed to by the internal cursor of the List object
    - Given a Buffer and BufferSize (amount of bytes contained in Buffer), adds to the file_t object's contents (contained in the node pointed to by the internal cursor) all the bytes in Buffer, beginning at the index idx. In this way, we can add multiple times to the file_t object's content, starting again where we left off. The last place we left off is indicated by 'idx'.
  - deleteContentsUpdateDate(List L, char *date, int max_file_size)
    - Removes the file type object's contents and updates the file object's date with the new modification date passed in as 'date'.
  - moveNodetoFront2(List L, int max_file_size):

        - Moves the current node pointed to by the internal cursor to the front of the list. (used when doing the modified caching).

httpproxy.c Contents and Information:

Global Variables :

- **List variable:**
    - CACHE; // holds cache contents

- **int variables :**
    - CACHING; // 1 if caching, 0 otherwise
    - MODIFY_CACHE_BEHAVIOR; // 1 if -u present, 0 otherwise
    - MAX_FILE_SIZE; // stores max size of file that can be cached
    - CACHE_CAPACITY; // how many items allowed in cache at one time
    - REQ_HEALTH; // num of requests until we perform a healthcheck probe
    - NUM_SERVERS; // how many servers we have to use
    - TOTAL_REQS; // number of total requests our program has received

- **pthread condition variables:**
    - cond_var // signal for worker threads to process request
    - health_thread_cond // signals healthcheck thread

- **pthread mutex variables:**
    - queue // locks queue accesses
    - server_to_use // locks portions of code when choosing next best server
    - total_reqs // locks when updating TOTAL_REQS variable
    - cache_access // locks when accessing cache

Structs:

- The httpproxy file contains several structs, below is each struct name, a quick description, and it's internal variables:
    - <u>server_t:</u>
        - the server_t struct represents a single server object.
        - uint16_t port // stores the port number of the server
        - int status // 1 if server is available, 0 otherwise
        - int num_errors // num of errors the server has processed
        - int num_requests // num of requests the server has processed
    - <u>servers_t:</u>
        - the servers_t struct represents an object containing all of the servers in an array, and also contains a pointer which points to the next best server to use in the array.
        - server_t* servers_array // array of server_t objects
        - server_t* server_to_use // pointer to the server to use in servers_array

- worker_t:
  - the worker_t struct represents a worker thread which contains data local to the thread, so that it may hop around to different functions and contain local information for easy access.
  - servers_t* servers_array
    - this servers_array is a pointer to the same array (in memory) in servers_t. In this way, when the servers_array is updated by an instance of the servers_t object, the worker will also see the change in the array.
  - server_t* server_to_use
    - this server_to_use is a pointer to the same server to use (in memory) in servers_t. When a servers_t object updates the server to use, the worker threads will also see this change.
  - int noServerUsed
    - 1 if no server was used to process the request (in the case of an error by the client, we do not update num errors of the server)
  - int storeFileContents
    - 1 if we must store the file contents in a cache item, 0 otherwise. Only used when we are caching.
  - char filename[]
    - contains the file name of the request we are processing
  - int idx
    - an index into the last written array slot of file contents buffer inside of the cache. Used so that we may know the last place we left off writing, so that we may continue to write more file contents later. Only used when we are caching.
  - int contentlen
    - the content length, or size of the file we are working with for the current request. Only used when we are caching.
  - int storeFromGet
    - 1 if we are to store file data from a GET request to save into the cache. Only used when we are caching.

Functions:

- **main()**
  - Parse the arguments given to our program:
    - I loop through the arguments, collecting the port numbers and headers. There must be at least two valid port numbers, and the first given will become our listening port. If there are less than two ports or a port number given is invalid, the program quits. While looping through arguments, I also check for the known flags: -N, -R, -s, -m. When a known flag appears as the previous element in the list of args, I check that the current element is a valid integer and store it for later use. If the value

given with -N or -R is 0, this is considered invalid and I exit the program. If -s or -m is 0, this is considered valid but there is no caching to be done later in the program. To indicate this, I set the global variable: 'CASHING' to 0. Finally, if we see the flag, -u, there is no value, and I set the global variable 'MODIFY_CACHE_BEHAVIOR' to 1 to indicate that our cache behavior is modified later.

- After the parsing is complete, we have saved all the server port numbers into a local array of uint16_t ports named "servers"
- Now, check if either the cache capacity or max file size is 0. If either are 0 there is no caching to be done. If both are non-zero, set the caching variables appropriately and create a new list (call the newList() function in List.c) to create the cache List.
- If we have less than 1 server, exit upon error, since we need atleast one connect-to-port.
- malloc space for a servers_array in a servers_t object, based on the number of servers * the size of a server_t objects. (We have an array of server_t objects of size num of servers).
- Now save the array of uint16_t ports named "servers", into the server_t object inside of the servers_t servers_array.
- Now create an array of pthread_t workers, of size number of threads +1.
- create a listening socket given the listening port.
- Call pthread_create() for each worker thread in the pthread_t array, initializing it's internal variables with starting values.
- Finally pthread_create the healthcheck thread.
- Set TOTAL_REQS (the total requests of the proxy) to 0.
- call signalHealthcheck() to do an initial healthcheck probing of all servers.
- Begin a while(1) loop:
  - Check for new incoming client connections
  - If we receive one which is valid, we enqueue it into our queue which holds client connection file descriptors.
    - We must protect the call to enqueue with a mutex for queue operations. Performing operations with the queue is not thread safe – if we enqueued or dequeued at the same time with different threads, two threads could end up with the same client connfd.
  - Signal to a worker thread on the condition variable, that there is work to do so a thread will begin it's run in doWork().
  - call the signalHealthcheck() function, to see if we must signal a new healthcheck probing.

- **doWork()**
  - Threads begin their run in this function.
  - doWork takes in a worker thread as its only argument, and returns nothing.
  - Begin a while(1) loop:

        - Dequeue from the queue and save the value as a client file descriptor.
            - The call to dequeue must be protected with a mutex.
        - If the queue is empty (the client file descriptor has the value -999) then we ask our thread to wait on our condition variable and queue mutex until a client connection file descriptor is available for us in the queue.
        - lock the mutex, server_to_use.
        - call update_server_to_use() function, given the worker thread. This will update the server_to_use variable inside the worker object.
        - Loop from 1 to the Number of servers in our array.
            - Check that the server_to_use is up, by calling create_client_socket() and checking the return value. (returns -1 if server is down).
            - If the server connection file descriptor does not return -1 and it's status is 1 then the server is up and can be used to process the request, close the server file descriptor and break.
            - Else, (the server is down or server has a status of 0 (it was down at some point between healthchecks) we are not allowed to use a server that was marked as down between healthchecks, so we set it's status as 0 and calle update_server_to_use() again.
        - Grab the value of SERVER_TO_USE and save into a local variable. This call must be protected by a mutex, since we do not want to grab this value at the same time it is being updated or we could end up with the wrong value or errors. If we save it as a local variable, even if the value is updated, we will have this current value.
        - unlock the server_to_use mutex (we have finished storing the data we need to use our server.
        - If the status of the server is 0, send an internal server error to the client, otherwise create a server file descriptor (by calling create_client_socket()) and call create_duplex_connection().
        - Then close the server file descriptor.
        - Finally, check the worker's internal variable to check whether we used a server (noServerUsed, set with 1 if we did not use a server) If we did not use a server, update the number of requests for the server (surround this by calls to lock and unlock the mutex, server_to_use, which protects calls to update the number of requests and errors for the server.
        - Finally reset all the worker's internal variables to starting values.

- **update_server_to_use()**
    - Takes one argument, a worker_t* worker object, and returns nothing.
    - This function's purpose is to choose the next server that will handle a request.
    - Choose the best server to be the first server.
    - Loop through the servers_array that is an internal variable of the worker object. Check that the current server is up (status = 1), if it is, and the best server's status is 0 (down), set the current server to be the best server. If the best

server's status is not down, compare the current server's number of requests with the best server's number of requests, if one has less than the other, choose the one with less requests to be the best server. Finally, if both servers are up and the both have the same number of requests, pick (between the best server and the current server) the one with the fewest amount of errors. If they both have the same errors, choose the best server.

- Finally, set the worker's internal variable, server_to_use, with the address (in the array) of the best server.

- **create_duplex_connection()**
  - This function is given the arguments: a client file descriptor and a server file descriptor, a server_t** object, a worker* worker object, and the port of our chosen server, and returns nothing.
  - The purpose of this function is to decide which direction a stream of data is to be received: from client or from server. We do this job with select()
  - Enter while(1) loop:
    - Use select() to determine the number of descriptors that are deemed ready
    - If select() returns -1, we return back to doWork() as we have a select() error
    - If select() returns anything but 0:
      - one of our descriptors is ready. If the client fd is ready, set a variable, isRequest to be 1, indicating we have a request to forward from client to server. If server fd is ready, set the variable, isRequest to be 0, indicating we have a response to forward from server to client.
      - If neither descriptors are ready, send internal server error
    - If select() returns 0: neither client nor server file descriptors are ready, so we do nothing and continue looping
    - Now that the isRequest variable is set, we check it.
      - If its value is 1, we call a function 'forward_request_or_response' to forward the bytes (a request) from client to server.
      - If its value is 0, we call the same function, 'forward_request_or_response' but indicate that we are forwarding a response from the server to back the client.

- **forward_request_or_response()**
  - Takes the arguments: client file descriptor, server file descriptor, an integer named isRequest which, if 1, tells us we are forwarding a request from client to server or, if 0, tells us we are forwarding a response from server back to client. It also takes a server_t** server object, a worker_t* worker object, and the server port. We return an integer, which indicates that we successfully received and sent all bytes from one descriptor to another (0), have not finished sending bytes (a number >0) or that we were unsuccessful (-1).

- We recv() data from the variable set as client_fd, and receive data of size 4096.
    - if the bytes received are less than 0, the connection ended or there was an error during the recv(), we return -1 and return back to create_duplex_connection(), otherwise, if the recv returns 0, we have collected all bytes and return 0.
- If we are handling a request (isRequest has the value 1), this means that we have received bytes from the client. Now we handle the headers (and the cache) by calling handleHeadersandCache() given the data that was received. handleHeadersandCache() will then return a value, -999 indicating that we should use the cached file contents instead of a server, 200 meaning we proceed to sending the bytes to a server, or a number greater than 200 indicating an error has occurred. Otherwise, we are handling a response back to the client and do not need to parse the headers.
    - If the value is -999:
        - send the headers to the client (based on the data we stored in the cache, includes the content length of the file and the modification date) and then all the bytes of the file from the cache.
    - If the value is greater than 200:
        - send an appropriate error message to the client based on the status code.
    - If the value is -999 or greater than 200, return 0.
- Otherwise (isRequest is not 1), we are responding back to the client.
    - send the received bytes to the function handleResponse(), which will return 200 or a value greater than 200 indicating an error status.
        - If the result is not 200, we lock the mutex called server_to_use, then increment the number of errors for the server passed in, then unlock the server_to_use mutex.
- Then we send the bytes to the variable called server_fd.
    - If the send() returns -1, return -1 (error). If the send() returns 0, we have not sent any bytes and have finished sending, return 0.
- Lock the cache_access lock.
- If our variable in the worker, storeFileContents = 1, then we know we must store the bytes sent from server to client into the respective cache node.
- We find the correct node in the cache, and if the node exists, (is not NULL), we find the index after the '\r\n\r\n' (if it exists) in the bytes and begin placing this data into the file_t object's contents variable.
- Call addToContents() method in the List.c file, given the Cache List, the file contents (stored in a buffer), how many bytes of the file are contained in the buffer, and the worker's index variable. Then update the index variable with the number of bytes we added from the file.
- Unlock the cache_access lock and return the number of bytes we initially received from the client_fd.

- **handleHeadersandCache()**

- This function takes in a char* Buffer containing the headers from a client's request, an integer representing the number of bytes in the Buffer, a worker_t* worker, a server file descriptor and a server port number. It returns an integer representing a status number. The status tells us some information about what to do in the function we return to. This will be discussed below and again in the description of the function we return to, which is called forward_request_or_response().
- Use strtok() to split the Buffer containing the headers by "\r\n". Upon the first portion of the string, grab the request type (GET or other), the filename, and the httpversion.
    - If the request type is not a GET, return 501 (representing not implemented)
    - If the httpversion is not HTTP/1.1 or the filename is not 19 or less alphanumeric characters or "." or "_" characters, return 400 (representing bad request)
- Continue the strtok() and check that each header is of the form "header: body", if not, return 400 (representing bad request)
- If caching is on, (the global variable CACHING is set)
    - save the filename we received from the headers into our worker's internal variable 'filename'.
    - lock the cache_access mutex variable (we are going to be accessing the cache and do not want changes made to our cache while we are searching through it or adding/removing from it).
    - Look for the file in the cache using the List's Find() function. If the file is found in the cache, it will return the node in the List, otherwise it will return NULL.
    - If the node is not in the Cache List:
        - we must send our request to a server. To do this, I set the worker's internal variable 'storeFromGet' to be 1. (Note: This indicates that after we send our client's request to a server and then receive from the server, we must store the data contained in the server's response (the headers) into our cache node, which contains the needed file data.) Then we unlock the cache_access lock and return 200 (indicating that we must now receive in the function we return to).
    - If the node IS in the Cache List:
        - Send the server file descriptor a request for a HEAD on the file.
        - Use select() to determine if the server is prepared to send us bytes (so that we may not hang on recv())
        - If select determines that the server is not prepared to send bytes, we return 500 (representing internal server error).
        - Otherwise, we recv() from the server, scan the contents using sscanf() and store the status number, status message, content length and modification date from the response.

- Now we compare if the date received from the HEAD response with the date in stored in the cache entry for the file. I do this by using strptime() and time_t variables. Then I use difftime() on the time_t variables to see which date is newer than the other.
    - If the modification date of the file from the HEAD response is not newer than the file in the cache, I set my internal worker variable, storeFileContents, to be 0. This will indicate later that I do not need to store file contents in the cache for the current request. Then, if we are modifying the cache to be LRU, (MODIFY_CACHE_BEHAVIOR will be set to 1) then I move the cache node containing the file to the front of the cache List and update the internal cursor of my cache List to point to this node. Then I unlock the cache_access mutex and return -999 (indicating that we are to send cached file contents to the client in the return function).
    - If the modification date of the file from the HEAD response is newer than the file in the cache:
        - I first check if the updated content length of the file is larger than the max file size we can store in the cache. If it is, I delete the cache entry.
        - Otherwise, if I set my internal worker variable, storeFileContents, to be 1. I then check if we are modifying the cache to have LRU behavior. If so, I move the node to the front of the cache List (to indicate most recently accessed) and set the internal cursor of the cache to this node. This will indicate that we must store file contents in the return function (update the cache entry with newest file data). Then I set my worker's internal variable idx (index into the cached file's contents variable to be 0. Then I call deleteContentsUpdateDate() given the cache List, the date received from the response, and the max file size.
    - Lastly, I unlock the cache_access lock and return 200.

- **handleResponse()**
    - Given a char* Buffer, an integer representing the number of bytes in Buffer, and a worker_t* worker variable, this function checks the status code of the response from the server and returns the status code of the response. It also handles some caching duties, described in detail below.

- 'Buffer', the buffer given to this function, contains bytes received from a server. These bytes can contain headers, headers and a content body, or just content body.
- Immediately, the status contained in the headers is checked. If it is above 200 (representing an error), then we return this status and leave the function.
- Otherwise, we continue by removing the bytes of the contents (if there are any).
- If we never found any headers (just file contents) we just return 200.
- Otherwise, we scan the header contents (using sscanf()) to grab the http type, status number, status message, content length number, and modification date.
- We lock our cache_access mutex and check that we are caching.
- If we are caching, check if the cache is at capacity (if it is, the List.c function length() will tell us how many items are in the list, and we can compare this to the global variable CACHE_CAPACITY.
    - If the content length of the file is less than the max file size, add the file data to a new node placed at the front of the cache List (call prepend()) and set the worker's idx variable to 0 indicating we have not stored any contents yet. Then set the worker's storeFileContents variable to 1, indicating we must now store file contents into the cache entry.
    - If the cache is at capacity, delete the back element (call deleteBack() function in the List file)
- If the content length

- **signalHealthcheck()**
    - takes in a single integer denoting whether or not we have a new request and returns nothing.
    - lock the mutex total_reqs.
    - If we have a new request (the input variable is set to 1) then increment the global variable TOTAL_REQS denoting the total requests processed by the proxy.
    - Then check if TOTAL_REQS variable modulus REQ_HEALTH (the number R of requests processed by the proxy before we ask for a healthcheck) is equal to 0. If it is, signal to the health thread to begin working. This is done by calling pthread_cond_signal() on the condition variable, health_thread_cond.
    - Finally, unlock the total_reqs mutex.

- **doHealthCheck()**
    - Takes one argument, a servers_t* object and returns nothing.
    - Begin a while(1) loop:
    - Lock the server_to_use mutex.
    - loop through all the servers in our servers_array (stored inside of the servers_t object)
        - Set the current server's status to 1 (indicating it is up)
        - check the return value of create_client_socket given the server's port, if the value is -1, the server is down, set it's status as down (0)

        - Otherwise (the server is not down), send a GET healthcheck request to the server (using dprintf()) and parse it's response. (Gather the response in a Buffer of size 4096 by using the send() function).
        - Scan the items we received using sscanf()
        - If the number of items scanned is correct, and the status code of the server's response is 200, save the number of requests and errors (in the body of the response) into the server_t object's respective internal variables.
        - If the number of items scanned is incorrect or the server returned a response with a code anything other than 200, set the server as down (status = 0).
        - Finally close the current server file descriptor.
    - After exiting the for loop, now loop again through all of the servers, choosing the next server to use based on their number of requests, number of errors and whether or not the server is down.
    - Finally update the server_to_use in the server_t object, and unlock the server_to_use mutex.
    - Next, lock the mutex total_reqs.
    - call pthread_cond_wait() on the condition variable health_thread_cond and on the mutex total_reqs.This will cause the thread to wait until it is signaled again by the health_thread_cond condition variable.
    - Finally unlock the mutex total_reqs.

- **isValid()**
    - Takes in a single string and returns an int (1) if the string can be converted to an integer, or (0) otherwise.
    - Checks that the argument given can be converted to an integer using strtol.

- **isValidNonZero()**
    - This function is provided to us.
    - Takes in a single string and returns an int (1) if the string can be converted to an integer and is greater than zero, (0) otherwise.
    - Checks that the argument given can be converted to an above-Zero integer using strtol.

- **create_listen_socket()**
    - This function is provided to us.
    - Takes in a uint16_t port and returns a socket file descriptor.
    - Creates a socket for listening for connections. Closes the program and prints an error message on error.

- **create_client_socket()**
    - This function is provided to us.

    - Creates a socket for connecting to a server running on the same computer and listening on the specified port number. Returns the socket file descriptor on success. On failure, returns -1 and sets errno appropriately.

- **strtouint16()**
    - Provided to us.
    - This function takes in a char array and returns the number if it can be converted to a valid UINT16 value and is non-zero, or (0) otherwise.