Pseudo-code:

```
Start (server.c)
main function
  print prompt "Enter the name of the file containing the Pokemon descriptions: \n"
  WHILE TRUE
    IF input is “q”
      THEN exit
    Take the file name from the user
    Open the file for read
    IF can’t open it
      THEN print "Pokemon file is not found. Please enter the name of the file again or press 'q' to quit.\n”
      continue
    ELSE Break

  Create the server socket
  IF can’t open socket
    THEN print "*** SERVER ERROR: Could not open socket.\n", and exit
  Setup the server address
  Bind the server socket
  IF could not bind socket
    THEN print "*** SERVER ERROR: Could not bind socket.\n”, and exit
  Set up the line-up to handle up to 5 clients in line
  IF could not listen on socket
    THEN print "*** SERVER ERROR: Could not listen on socket.\n”, and exit

  WHILE TRUE
    Accept incoming client connection
    IF could not accept incoming client connection
      THEN print "*** SERVER ERROR: Could not accept incoming client connection.\n", and exit
    Print "SERVER: Received client connection.\n"
    WHILE TRUE
      Get the message from the client
      Put a 0 at the end so we can display the string
      Print the received message
      Moves the file marker to beginning of the file.

      WHILE not reach the end
        Read each line in the file
        IF the type1 matches the input
          THEN send this line to client
          Receive an acknowledgement from the client after sending one line

      When a search is complete, respond with an "OK" message
      IF the received message is "done" or "stop"
        THEN break

    Close this client's socket
    IF the client said to stop
      THEN break
  Close the file
  Close the socket
Stop

Start (client.c)
main function
  Create the client socket
  IF can't open socket
    THEN print "*** CLIENT ERROR: Could not open socket.\n", and exit
  Setup the client address
  Connect to server
  IF unable to establish connection
    THEN print "Unable to establish connection to the PPS!\n", and exit

  Allocate memory for Data struct that will be passed to thread functions
  Initialize the semaphore
  IF something wrong
    THEN print "Error: on semaphore init.\n", and exit

  WHILE TRUE
    print prompt "a. Type search\n" "b. Save results\n" "c. Exit the program\n"
    Get the input from the user
    IF input is "a"
      THEN print prompt "CLIENT: Enter the type1 to send to server ... \n"
      Get the type1 and send it to server
      Initialize a pokemon pointer to save pokemons in one search.
      WHILE TRUE
        Receive message from the server
        IF buffer is not "OK"
          THEN num_lines++
          Allocate memory for the new pokemon
          Call the line_to_pokemon function to convert this line to a pokemon
          Send an acknowledge to server
        ELSE break

      Reallocate the memory for the pokemons array used to save all the completed searches
      Increase the number of queries
      Append all the pokemons from one search to polemons array
      Increase the total number of pokemons saved in the pokemon array.
      Free the temporary array used to save pokemons from one search
      continue
    ELSEIF input is "b"
      Create a thread to save all pokemons in memory
      Join the thread
      continue
    ElSE
    Print the total number of queries completed
    Print the names of new files created during the session
    Destroy the mutex, and free memory
    Ask the user whether to shut down the server, and send command to server, and break
  Close the socket
Stop

Start (dataProcess.c)
saveFunc function
  Ask the user for the file name to save pokemons
  Concatenate the filename in a string for later display.
  FOR int I to number of pokemons
    Call pokemon_to_line to convert a pokemon to a line and save it to the file
  Close the file

line_to_pokemon function
  Call strsep(&line, “,”) to get each attributes from the line
  Copy each attribute to corresponding position in a pokemon struct

pokemon_to_line function
  Use strcat(line_to_write, pokemon_to_write) to convert a pokemon to a line
  Line_to_write = strcat(line_to_write, pokemon_to_write)
Stop
```

There are three c files and one header file in this program, and we use client/server model (TCP). In this model, one process acts as a server that receives requests from clients and then performs tasks accordingly. The server.c will communicate with clients and return the requested information to the client, and the client.c will use a while loop to ask user for the name of the file containing the Pokemon descriptions. Once opening the file successfully, it always prompts a menu for the user to choose among search for files, save the search, and exit the program. The main thread will send the type1 that user provided to server and temporary save the information from server if user choose option a. If user choose b, one thread will be created to save the search pokemons (only completed search will be saved). If user choose c, the total number of queries completed and file names used to save search will be displayed, then the program will be terminated. Line_to_pokemon function and pokemon_to_line function are helper functions to either convert a line from the file to a pokemon struct or convert a pokemon struct to a string.

Starting with the server, we need to create a stream socket. This can be done with the socket() function which is defined in the <sys/socket.h> header, and the <netinet/in.h> header file contains definitions for the internet protocol family. Also, the <arpa/inet.h> header makes available the type in_port_t and the type in_addr_t as defined in the description of <netinet/in.h>. In order to create threads, we use the pthread_create() function which is defined in the <pthread.h> header file. A semaphore is defined as a sem_t type and we need to include the <semaphore.h> header in our code in order to use it.

We set each pokemon as a struct and save it in a pokemon array in a search, then append this array to a large array for later use. In order to protect these shared data from corruption, we use semaphores to accomplish this. The semaphore acts as a locking mechanism to prevent other threads from accessing or modifying a resource at the same time. While the resource is locked, other threads are waiting.