Karen Shay West
9 Shannon Marie Way, North Easton, MA 02356

508-844-9776 http://www.linkedin.com/in/karenshaywest KarenWest15@gmail.com

SUMMARY OF RECENT PROJECT OF MY OWN IN C on Linux–

- **a C shell Linux command interpreter, currently being completed with plans to expand with traveling salesman process and thread algorithm programs and a sorter of their paths**
- see my Linked In Project Section for details and github links to files:

- **Systems Programming to go into further depth in Linux Programming and IO: So far, I've implemented a C Shell Linux Command Interpreter, but it's not done yet:**
  - **Goals:**
    - to become more familiar with the primitives used in process control and inter-process communication
    - simulate a multi-threaded, multi-process environment, that implements inter-process communication, signal handling, and a strong understanding of flow control
    - 2 primary aspects of this project:
      - implementing a shell environment to process command line input
      - specification and execution of a parallel program that is either comprised of multiple processes or threads
    - Concepts Used to complete this project included: inter-process communication, (including pipes, signals, shared memory), process creation and execution. Relevant functions include: getcontext/makecontext/setcontext/swapcontext, sigsetjump/siglongjump, kill, pipe, popen, shmget, shmctl, shmat, fork, execve, execvp, etc.
    - **These are the 4 programs involved, and the one so far that's mostly been written is the basic shell in the list below:**
      - **Basic shell (done in C)** for specifying and executing programs, called "myshell".
      - A **multi-process version of the traveling salesman problem**, called "tsp_p". **Note that I've studied and implemented something similar in my Python algorithms and MATLAB communication systems studies, but not yet in C. Here the plan is to expand my Linux systems knowledge and combine it with this type of algorithm.**
      - A **multi-threaded version of the traveling salesman problem**, called "tsp_t", with an added bonus of implementing it without the SYSV context functions. Solutions that bind user level threads with kernel threads are also a bonus, and using pthreads or any other threads library is not the approach to take, and instead, you create your own threads. It was also recommended NOT to rely on the Linux clone() call in the thread creation solution, if at all. The solution can bind user level threads onto user level threads, but you still need to use a different approach for making the portable user-level threads.
      - A **sorter program**, called "tspsort", that sorts the paths found in the traveling salesman problem, from shortest path to longest path distance. **Note that I've studied this part as well in my Python algorithms and MATLAB communication systems studies, but here it is combined with the Linux systems**

    learning in C.

- **Testing It:**
  - At Linux prompt, type “myshell”, which starts the interactive command environment
    - Currently, this is as far as I've gotten, and it works for most but not all command and IO redirection scenarios.
  - In “myshell”, run “tsp_p < input_graph | tspsort > tsp_p.out”
    - “tsp_p” computes all possible paths through a graph (G) of vertices (V) and edges (E).
    - a path is defined as an ordered list of nodes from a starting point, S, visiting all other nodes in V exactly once, and then returning back to S.
    - a path distance is defined as the sum of all edge weights between pairs of nodes in G visited along a specific path
    - each path along the route with its overall distance will be the output to stdout by “tsp_p” as it is discovered
    - the result is “piped” by “myshell” to stdin of “tspsort”, which collects all paths and orders them from shortest to longest distance
  - In “myshell”, run “tsp_t < input_graph | tspsort > tsp_t.out”, with the same goals with the thread version as above with the process version
- **Basic Shell Details (in C on Linux--this is the part of this project that is mostly done):**
  - **Goal:** to implement a basic shell called "myshell", to learn how a command interpreter works. It's a simple parser to interpret commands, that uses fork/exec functions to create and execute these commands. I/O redirection and the use of pipes was required.
  - **"Myshell" specification**
    - similar to the "bash" shell, with the following requirements:
      - **1. Execution file:** the execution file called "myshell".
      - **2. Shell prompt:** If the input to the "myshell" program is from a terminal, you should print "myshell>" as the prompt for the user to type in the shell command. If the input is from a file, you should not print any prompt, as this will help when it comes to testing your projects. NOTE: You can use the "isatty" function to determine whether the input (via a specific file descriptor) is from a terminal or not.
      - **3. Command execution and shell termination:** "myshell" reads its input from stdin, parsing the command line to determine how to execute the shell request. Termination of "myshell" occurs either when a user types "ctrld", or when the program issues an "exit()" call due to some error/termination condition.
        - **Description of the valid command line formats.**
          - Cmd - A single command that identifies the name of a program file.
          - cmd; cmd - A sequence of commands: "myshell" should execute them one after another.
          - cmd > output_file - Redirect stdout of the cmd to the file with the name "output_file".
          - cmd < input_file - Redirect stdin of the cmd to the file with the name "input_file.
          - cmd 1> output_file - Redirect stdout of the cmd to the file with the

        name "output_file".
        - cmd 2> output_file - Redirect stderr of the cmd to the file with the name "output_file".
        - cmd &> output_file - Redirect both stdout and stderr of the cmd to the file with the name "output_file".
        - cmd1 | cmd2 | cmd3 - Pipeline the output of the command before each "|" symbol to the input of the command after the same "|" symbol.
        - cmd & - "myshell" should let cmd execute in the background, so that the shell can interpret further input commands.
    - **4. Special Input:** "ctrlc" is the default key sequence to send a "SIGINT" signal to a process. In "myshell", you should intercept this signal, and send it to all "foreground" processes to stop them instead of stopping "myshell".
    - **5. Prevent "zombie" children:** when a command running in the background finishes, it must inform the "myshell" (parent) process via a "SIGCHLD" signal. "myshell" can then keep track of which child processes are still active.
- **Basic Shell operation:**
    - You should use the "fork" syscall to create each child of "myshell" and then one of the "exec*" functions to execute the command in the corresponding child's address space. You should also use either the "wait" or "waitpid" function to determine the exit status of a terminating child.
    - For output redirection and piping between processes: to redirect a file descriptor to stdin, stdout, or stderr you can use the "dup2" function.
- **Code developed in a series of stages:**
    - 1. write and test a parser to read the command lines – this works
    - 2. get the simple commands to work – this works
    - 3. get I/O redirection to work – this works
    - 4. get pipes to work – this works for ever some cases but not with 2nd command sent to an output file instead of stdout
    - 5. get "&" to work – not implemented yet
- **Parsing shell strings:** in this project, you only need to implement a simple command line parser, in which we use "space" as the delimiter between command "tokens". You can use support libraries such as the Gnu readline library for command line parsing.
- **Traveling Salesman Multi-Process Version:**
- **Problem:**
    - The problem to be addressed is a parallel implementation of the Travelling Salesman Problem (TSP) on an input graph, G={V,E}, comprising a set of nodes or vertices, V, and a set of edges, E. Your main program (called "tsp_p") is to be invoked as follows:
        - $ tsp_p
    - It then reads an input graph from stdin. You can assume the input graph, G has |V|=n vertices representing different cities. The graph itself can be represented as an n*n distance matrix, D, such that D[i,j] is the distance between city i and city j. The diagonal of the matrix has distance 0 for each D[i,i] . You can assume all distances are in integers over some predefined range [0..MAX_DISTANCE] and the number of cities, n, is limited to a maximum value MAX_NODES. We will likely not test

your program on values of n much above single digits but you should try to design your code to be as efficient as possible, so that it can scale to large input graphs. Not all cities need to have a direct path between them. In this case you can assume there is no edge connecting them. In the input graph you can assume a value for D[i,j] = 1 if there is no edge directly connecting cities/nodes i and j. For all pairs of cities with direct paths, or edges between them, then your distance matrix must contain a valid value. You can assume MAX_DISTANCE is set to 255.

  - For each node in G, you should consider it the starting point of all possible paths to all other nodes. You should create a separate process (n in total) for each starting node and compute all paths from that starting point to all other nodes and back again to the starting point.
  - The output of "tsp_p" is a list of paths along with overall distance. A path is defined as an ordered sequence of nodes, from a starting point S visiting all other nodes exactly once and then returning back to S. The paths should be output along with their distances to stdout.
- **The Big Picture of the Multi-process, Traveling Salesman Problem:**
  - 1. **The first step is for the parent process to the input graph from stdin**, and perform appropriate error checks, to verify its correctness.
  - 2. **The parent process then creates a shared memory segment which is big enough for communications between parent and child processes.** Using the "fork" syscall, the parent creates n child processes.
  - 3. **A child process uses one of the exec functions to map a program file called "find_paths" into its address space.**
  - 4. **The "find_paths" program will take a starting node, S, in the input graph and calculate the distances of all possible paths from that node to all other nodes and back again to S. Aside from S, all other nodes along a valid path should only be visited once.**
  - 5. The **parent process waits until all of the child processes have finished, and then outputs the resultant set of paths and their distances stdout.**
  - 6. **Each child will send back its paths to the parent via shared memory.** You need to establish sufficiently large shared memory between each child and the parent. This memory region does not need to be big enough to store all paths but could instead form a communication channel to exchange paths as they are produced. Note that each process has a unique starting point. Since there are n possible nodes (or cities in the input graph) then each of the n child processes has a different one of these nodes as its starting point.
- **Traveling Salesman, Multi-threaded Version:**
  - **Problem:**
    - The problem to be addressed is the same as in "tsp_p", except you will use multiple threads to calculate paths through a graph instead of separate child processes. Your multithreaded program should be called "tsp_t".
  - **The Big Picture of the Multi-threaded traveling salesman version:**
    - The first step is for "tsp_t" to read an input graph as before, from stdin, and perform appropriate error checks, to verify its correctness.
    - Once the "tsp_t" program has read the input graph, it will create a series of threads to help solve the Travelling Salesman Problem. To create threads, you cannot use any supporting libraries for thread creation. Instead, you must write a

function called my_thr_create(), as follows:

    - void my_thr_create(void (*func) (int), int thr_id);
    - The first argument is a pointer to a function for the thread to execute, while the second argument is a unique integer thread ID that you create for each new thread. NOTE: To establish the concept of a thread, you will either need to establish a "ucontext_t" for each thread having a given ID, so that you can use setcontext/swapcontext to save/restore each thread's state, or you can use (sig)setjmp/(sig)longjmp/sigaltstack to implement the functionality of the context handling functions. More points will be given to a solution that avoids the use of set/swapcontext/etc functions.
    - Since the main thread and all subthreads share the process memory space, there is no need to use separate shared memory to store all discovered paths.
    - When all subthreads have finished their "find_paths" operations, the main thread will then output the resultant set of paths to stdout.

- **Hints Given Were:**
  - To create your threads you will need to understand how context control works. This is documented in the GNU document on System V contexts.
  - *** A more portable approach to implementing threads, that requires only setjmp/longjmp, sigaltstack and signal handling will be given extra credit ***