CEng 536 Advanced Unix
Fall 2024
HW1
Due: 23/11/2024

1 Description

In this homework you will write an socket based service for matching supplies and demands on a 2D area. Clients connecting to the service can declare amount of items they demand and amount of items they can supply. When there are clients with proximity (on 2D grid) match, the supply is greater than or equal to the demand, they transfer items. There are three kind of items. A supply matches a demand only if items of all kind match. Clients make stream socket (Unix domain or TCP) connections to a server to access the shared map (2D area). Server is a multi-process application starting a worker process per client connection. Clients keep connected until they terminate or send quit command. During a client session, each client can send asynchronous text commands terminated by newline with the following descriptions:

move x y
: Client move to a new location. The former demands of the client is not affected. Assume client is a mobile unit navigating on map and recording the demands. All following demands and supplies of the client will be effective with the new location.

demand nA nB nC
: Declares that the current client demand the following amounts. Each demand is tagged with the current position of the client.

supply distance nA nB nC
: Declares that the current client can supply the following amounts. The client can deliver the items on a client in the given distance. There will be no match if a client demand has more distance than supplier declared value.

watch distance
: Clients wants to be informed on all supplies in a given range, distance is less than or equal to distance. The supplies will not be delivered but the watching client will be informed causing a text messages on clients terminal. If client moves, the watch will still be effective with the declared position, not the new location. A client can watch only one position and distance. A watch requests clears the former watch.

unwatch
: Cancels the current watch if there is one.

mydemands
: List all of demands by current client.

mysupplies
: List all supplies by current client..

listdemands
: List all of demands globally.

listsupplies
: List all supplies globally.

quit
: Quit and close the connection, server closes connection and removes all supplies, demands and watches created by the connection. Client closing the socket without this command also has the same affect.

All distances are Manhattan Distance, sum of absolute value of differences on both dimensions. Both demand and supply commands may cause a a match when issued. Match occurs when:

$d_A \leq s_A$, and $d_B \leq s_B$, and $d_C \leq s_C$, and $\text{dist}_{Manhattan}(d, s) < d_{\text{distance}}$

In case of a match:

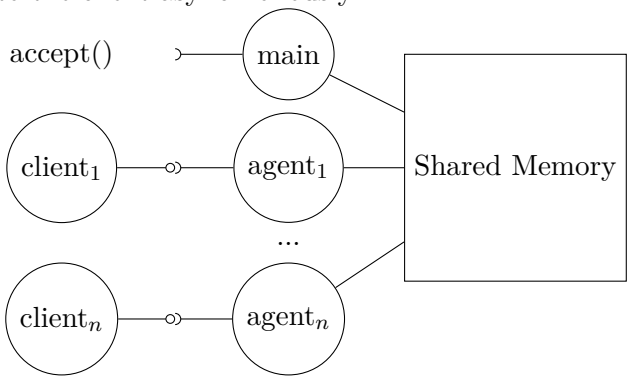
- The demand is removed from the map
- The demand amounts are subtracted from the supply
- If all three amounts of a supply get zero, it is removed from the map
- Both clients are notified about the operation on their connections

The watches do not change supply and demand amounts. Only the watch distance is significant. The watching client gets a notification from all supplies in the proximity regardless of their supply distance and amounts.

2 Architecture

You only need to implement the server side for this homework. Any socket based text client like netcat can be used to take the client role. The server (master in the figure) process listens and accepts the client connections. For each connection, it calls fork() to start a child. All children will use a shared memory area to exchange item and watch information. We are going to call the server side child processes agents. For each client connection, there

will be an agent responsible for getting commands from client and sending notifications to the client asynchronously.



In the agent, you need to block for reading commands from sockets, and shared memory for notifications synchronously. In order to do that, you can have two threads for each agent.

You can use Posix shared memory, System V shared memory, mmap() (best with MAP_ANONYMOUS in this case) for storing your key data structures, list of supplies, demands, and list of watchers. For synchronization you can use any of Posix semaphores, System V semaphores, pthread mutex and condition variables (see pthread_mutexattr_setpshared and pthread_condattr_setpshared, if mutexes or condition variables are allocated in shared memory, they can be used in IPC as well) and signals.

Note that waking up all agents and than agents cherry-pick the notifications is not acceptable. Only the relevant agents affected from the operation should be notified. When an agent is notified, it can query all items and show item changes after a timestamp. Another solution is to implement a notification log in shared memory accessed and cleaned by the agents.

For the event and connection information, you can use linked lists (if you like challenges you can also use spatial trees for faster query) allocated in a fixed array (see the limitations below). The pointers of your data structure, like next of the linked list, will be the index in the array. You can mark free members of the array sentinel values in the field. Note that you need to allocate a large shared memory area and use different offsets for different data structures.

3 Execution

Your code will compile into supdemserv. The following command line arguments are going to be supported:
./supdemserv conn Width Height

conn is the address to be listened by the master process. If it starts with a @, it is assumed to be a Unix path, thus a Unix domain socket.

Otherwise it should be in ip:port format, IP and port of the IPv4 socket address. Width and Height are the dimensions of the map.

4 Limitations

Shared memory does not dynamically grow by default. So we have limitation in the number of demands, number of supplies, number of agents, and number of watches. The following limits are assumed:

- Maximum number of demands and supplies are 10000 each.
- Maximum number of agents are 1000 (you can have at most one watch per agent).
- Total number of active connections is limited by 1000. This limit is put for any internal data structure you need in shared memory per connection, not for rejecting connections.

5 Input/Output

All client commands are single line and easily parsable by scanf() or sscanf(). The message and description strings at the end of the line should not contain the new line character at the end.

When agents send notifications to the client, one of the following outputs are written to client socket (shown in printf syntax. Strings replaced by descriptions.):

```
Your demand at (%d,%d), [%d,%d,%d] is fulfilled by a client at (%d,%d).
Your supply at (%d,%d), [%d,%d,%d] with distance %d is delivered to a client at (%d,%d) [%d,%d,%d].
Your supply is removed from map.
A supply [%d,%d,%d] is inserted at (%d,%d).
```

The output of mydemands and listdemands commands has the following output format (header, than for each row):

```
There are %d demands in total.
  X   |   Y   | A | B | C |
-------+-------+-----+-----+-----+
%7d|%7d|%5d|%5d|%5d|
```

The output of mysupplies and listsupplies commands has the following output format (header, than for each row):

```
There are %d demands in total.
  X   |   Y   | A | B | C |   D   |
-------+-------+-----+-----+-----+-------+
%7d|%7d|%5d|%5d|%5d|%7d|
```

All successfully commands is replied by the agent as ‘OK’. You do not have to handle error cases (map bounds check, input type check, input ranges etc).

6 Submission and Questions

Use ODTUClass for submission. Either submit a single file, supdemserv.c or supdemserv.cpp, or submit a tar.gz including a Makefile (no other fancy archive tools like zip, arj, 7zip etc please). In that case, make command on top of the archive should compile supdemserv binary in the same directory. Your code will be tested on Linux. Single file submissions will be compiled with '-lpthread -lrt' flags. You can ask any homework related questions on course newsgroup or ODTUClass forum.