Project 4: Goldchase–Two tin cans and a string

This project is an enhancement of project 3, where we add support for sockets so players on other computers can connect to a running game.

Summary

A game instance may be running on either a server computer or a client computer. More than one game instance may be running on both the client and the server. There may only be one server computer but there may be multiple client computers.

The first game instance to start on a server computer will also start a daemon on that server. Likewise, the first game instance to start on a client computer will also start a daemon on the client computer. The daemons manage the communications between computers. It will also now be the responsibility of the the daemons to clean up/remove shared memory once all players have left the game.

Initial game play enhancements

Make three minor modifications to goldchase:

1. Add an additional member to the `gameBoard` structure:

   ```
   int daemonID;
   ```

2. As in the previous project, assign the game's process ID to the `gameBoard` structure in shared memory.
3. Check the contents of the daemonID variable you added above. If it is not equal to zero, then send the daemon process a `SIGHUP` signal.
4. When the game is exiting, after the game's process ID has been set to zero and the player removed from the map, send the daemon process another `SIGHUP` signal. In the previous implementation of Goldchase, the last player would unlink the shared memory and semaphore. The daemon will now be responsible for removing the shared memory and semaphore.

Signal handlers in daemons

SIGHUP **(player has joined or left game)**

1. Socket write: **Socket Player**
2. After the socket write, check the value of the byte you just wrote to the socket. If the byte equals G_SOCKPLR (that is, all of the player bits are off), then there are no more players playing the game. Close and unlink the shared memory and semaphore, the exit the program.

SIGUSR1 **(map has changed)**

1. Create vector pairs

   [`pair.first=(short)offset, pair.second=(unsigned char)mapsquare`]

   of each map square which is different between local copy and shared memory. Update local copy as necessary.
2. If length of vector is > 0, socket write: **Socket Map**

SIGUSR2 **(a message queue message is waiting)**

1. Loop through each message queue. When a message is found in queue, socket write: **Socket Message** protocol.

Socket communications protocol

Socket Message (a message is being sent to a player)

(1) 1 byte: G_SOCKMSG or'd with G_PLR# of message recipient

(2) 1 short: n, # of bytes in message

(3) n bytes: the message itself

Socket Player (a player is joining or leaving the game)

(1) 1 byte: G_SOCKPLR or'd with all active players G_PLR#

Socket Map (a map refresh is required)

(1) 1 byte: 0

(2) 1 short: n, # of changed map squares

(3) for 1 to n:

1 short: offset into map memory

1 byte: new byte value

Socket Client Initialize (the client daemon has connected)

(1) 1 int: the number of map rows

(2) 1 int: the number of map cols

(3) n bytes: (where n=rows X cols) the map

Daemon behaviors

Server start up

1. Connect to shared memory & map via mmap()
2. Call `getpid()` to retrieve the daemon's process id for the `daemonID` field in the `gameBoard` structure in shared memory.
3. Allocate memory (rows * columns) and initialize it with the same values that are in the shared memory map. The server maintains a local copy of the map so that when it receives a SIGUSR1, it can compare the two maps to determine which bytes in the shared memory map were changed by a game process.
4. Start trapping `SIGHUP`, `SIGUSR1`, and `SIGUSR2`
5. Listen on socket for a client to connect

Server receives client connect

1. Socket write: **Socket Client Initialize** (see Socket communications protocol)
2. Socket write: **Socket Player** (see Socket communications protocol)
3. Enter an infinite loop. The loop should block on the read() system call (actually, READ), trying to read a single byte. The contents of that byte will determine which communication is being started (note that all types of messages under **Socket communications protocol** begin with a single byte).

Client startup

1. Read the rows and columns from the socket (see **Socket Client Initialize** protocol).
2. Allocate space (rows * cols) for a local copy of map (not the gameBoard, just the map)
3. Read *n* bytes (where *n*=rows*cols) from the socket into the local copy of map (see **Socket Client Initialize** protocol). The client maintains a local copy of the map so that when it receives a SIGUSR1, it can compare this local copy with the shared memory map to determine which bytes in the shared memory map were changed by a game process.
4. Create a semaphore
5. Create shared memory (`shm_open`, `ftruncate`) just as a new game instance would.
6. Initialize shared memory from your local copy of map, and rows & cols.
7. Call `getpid()` to retrieve the daemon's process id for the `daemonID` field in the `gameBoard` structure in shared memory.
8. Start trapping `SIGHUP`, `SIGUSR1`, and `SIGUSR2`
9. Read **Socket Player** from Socket (handle as per SIGHUP, step 2).
10. Enter an infinite loop. The loop should block on the read() system call (actually, READ), trying to read a single byte. The contents of that byte will determine which communication is being started (note that all types of messages under **Socket communications protocol** begin with a single byte).

The three headings below correspond to the three possible bytes which may read from the socket as per the infinite loop #9 above

On socket read "Socket Player"

Loop through player bits. The player bits are contained in that single byte you read from the socket protocol: *Socket Player (a player is joining or leaving the game)*:

1. If player bit is on and shared memory ID is zero, a player (from other computer) has joined:
    1.1. Create and open player's message queue for read (daemon will receive messages on behalf of player, forwarding them across socket)
    1.2. Set shared memory ID to pid of daemon
2. If player bit is off and shared memory ID is not zero, remote player has quit:
    2.1. Close and unlink player's message queue
    2.2. Set shared memory ID to zero

After looping through the player bits, check the value of that single byte you read in. If that single byte equals G_SOCKPLR (that is, all of the player bits are off), then no players are left in the game. Close and unlink the shared memory. Close and unlink the semaphore. Then exit the program.

On socket read "Socket Message"

After reading this first byte, you need to process the remaining bytes from the socket protocol: *Socket Message (a message is being sent to a player)***.**

1. Determine message recipient
2. Determine message
3. Open, write, and close to message queue of recipient

On socket read "Socket Map" (A 0 which is map update)

After reading this first byte, you need to process the remaining bytes from the socket protocol: *Socket Map (which means a map refresh is required)*

1. Determine number of changed map squares
2. Change local copy and shared memory copy of map
3. Send `SIGUSR1` to each local player

Final game play enhancements

Don't integrate these additional enhancements until after you have all other aspects of the application working:

1. If a hostname *is* provided on the command line, the Goldchase process is assumed to be a **remote process** which interacts with a **client daemon**.
2. If a hostname is *not* provided on the command line, the Goldchase process is assumed to be a **local process** which interacts with a **server daemon**.
3. *After* a Goldchase **local process** has registered its process ID with shared memory, check the daemonID.
    1. If the daemonID is not zero, send a `SIGHUP` signal to the daemon process (this was completed as part of the initial enhancements).
    2. If the daemonID is zero, fork and exec the server daemon.
4. *Before* a Goldchase **remote process** initializes (immediately after startup),
    1. if the `shm_open()` command fails, then the **client daemon** isn't running yet. Fork and exec the **client daemon**. Be sure to provide the client daemon with the process ID of the parent process (the game is the parent, the new client daemon is the child). Once the client daemon is initialized and ready, it should signal its parent that it may continue initializing The parent should proceed as in #2 below.
    2. If the `shm_open()` command succeeds, the the **client daemon** is already running. After the remote process has registered its process ID with shared memory, send a `SIGHUP` signal to the **client daemon** (this was completed as part of the initial enhancements).
5. When a Goldchase **local process** or **remote process** quits, send a `SIGHUP` signal to the daemon process (this was completed as part of the initial enhancements).