```
else{
	// Check the data structure and see if any descriptors are ready to read
	for (int i = 0; i < NUM_OF_PLAYERS; i++) {
		if (FD_ISSET(streams_[i]->connectionNumber(), &fdread)){
			printf ("Message form player#%d\n",i);
			return i;
		}

	// If none were set we get here. This means that ten microseconds have
// passed, and no one did anything. So we return some value to say no   /
	// one moved.
	return NO_MOVES;
```

To find out more about file descriptors (and UNIX in general), read some of W. Richard Steven's great reference book Advanced Programming in the UNIX Environment. Skim some of the introductory sections.

```
int
Client::Listen() {

  int status = 0;       // select error value
  int highfd = -1;      // the highest file descriptor we need for select
  fd_set fdread;        // The data structure we pass fd into select with
  int fd = 0;           // A temporary

  timeval tv;           // a predefined struct to tell select how long to wait

                  // Set our time-limit to ten microseconds
  tv.tv_usec = 10;      // micro-seconds
  tv.tv_sec = 0;        // seconds

  FD_ZERO(&fdread);  // Initialize our fd_set

// This for loop not only adds every NetworkStream's fd to the fd_set but also
// finds out which one is highest

  for (int i = 0; i < NUM_OF_PLAYERS; i++) {
    fd = streams_[i]->connectionNumber();   // Get fd
    FD_SET(fd, &fdread);                              // Put it in the fdset
    if (fd > highfd)                                  // Which is highest
      highfd = fd;
  }

 // Here we call select. I am doing a c style check on the function call.
 // I assign it to status so that I can keep the error value around if I need it.
 // select errors are all < 0 If things are cool it holds the number of ready file
 //descriptors (as if when more than one are ready)
  if ((status = select(highfd+1,&fdread, 0,0,&tv)) < 0)
    perror("Eat me"); // This is a systems function that will print errors
```

< we continue the else statement on the next page >

System calls such as `open(), read(), write(), close(), and seek()` deal with file descriptors, and therefore can be used on all of the types of objects that the file system abstracts. See the man pages for each of these commands for a detailed description of what they do and how to use them.

Example:

```
- int ffd = open("/u/jal/foo", O_RDONLY);
    // ffd is set to a value which refers to a file
    // system object which happens to be a file on an
    // NFS mounted disk

 - int tfd = open("/dev/pts/1", O_RDONLY);
 // tfd is set to a value which refers to a file
 // system object which happens to be a terminal on
 // my computer

    const char *msg = "Hello!\n";
  - write(ffd, msg, strlen(msg));
    // prints "Hello!" to /u/jal/foo
    write(tfs, msg, strlen(msg));
    // prints "Hello!" to someone's terminal :-)
```

Select:

`select()` will assist you by telling you which file descriptors are ready for reading or writing at any particular time. select() deals with sets of file descriptors (or fd_sets). While most file system calls work on one object at a time, `select()` will tell you which of a number of objects are ready to have `read()` or `write()` applied to them.

Below is an example of a client's use of select. Remember streams_ is an array of NetworkStreams. The function Client::Listen() returns which stream in the array has a message or NO_MOVES of none of them does.

Appendix A

Unix Implementation

File Descriptors:

The UNIX file system provides a useful abstraction for several system objects which all support a common `read()` and `write()` interface. These items include files, queues, pipes, sockets, and terminals.

Examples:

- Your terminal is read and written to with file system calls.

- The NFS mounted `/u/jal/foo` is read and written to with the same file system calls.

- Input/output redirection is abstracted by the file system, i.e. `ls > foobar` runs the ls command in an environment where all output commands go to a file rather than the terminal.

All of these items are represented by an abstract object called a "file descriptor". A file descriptor is an integer whose value is greater than or equal to zero, thus a negative number is an invalid file descriptor. This integer value is your "identifier" which tells the file system what object you actually want to work on. file descriptors are also secure object references.

Example:

- The default output object is referred to by file descriptor 1. Whenever you call `printf()` or use a `cout` stream, internally it is calling `write()` with fd = 1.

- To redirect output, the file system can merely *replace* the object pointed to by file descriptor 1 (usually your terminal) with another object (like a file or a socket, etc).

Balance of computation:

If you picture your world as a distributed database then the clients and active agents (the AA's) are all operating on the database at one time. Whether they do it on one machine or do it spread throughout the network is called the Balance of computation.

#1 One way of building your game is to have all AA's working on one server (in a star topology) then they can have their decisions sent out when the server sends data to all of the clients. This could also slow down the server (and slow down the game).

#2 If your AA's act in a deterministic manner you can do what they did in Netspace. (thank you scd) That is have all of the AI computed on each client. No one needs to send AA data across the net and since the game is in synch and the AA's are deterministic they will act in the same manner. Easy.

#3 Very complex AA's are similar to players and can even be treated by the server as such. You can implement them as robotic players that the server treats the same way as it would a human player. Simply implement them as a new type of client that computes its behavior rather than reading it from a human user and then either start the client as you would a normal one or have the server automatically start up the robot players as part of the game startup.

The idea is to make the design and implementation of your game simple, with as little net traffic as possible.

Network Topology:

A quick review:

Star: Safest and simplest. It involves having each client process send its data to a central process which then relays it out to the other clients.

Round-Robin: We advise against this type of design for the assignment. Since our game must be constantly interactive it would not be feasible.

Fully-connected: Everyone has their own world state and is connected to everyone else. Synchronization worlds as outlined above. Fast, but may result in more net traffic than you want.

Synchronization:

This concept deals with making sure everyone is advancing at the same tim - in synch. It overlaps with consistency and also deals with other things. We need things to be in synch so that the users can affect each other quickly and consistently.

Methods of synchronization:

#1 Blocking as noted above insures that everyone moves together, used naively it can slow and halt the game.

#2 The server can choose to wait a given amount of time before sending information to everyone. If it receives input, it sends it to the clients; if not, it tells the clients that no one moved. In this manner it makes sure all the clients receive data at regular intervals.

#3 If the design is a client-to-client topology then the server can be used to tell each client when to update frames. No information need be passed; just a message that the clients know means "I can update my visual now." This way they can exchange information between each other, and get the game moving faster.

#4 Each client could connect to the others and update their frames at given time intervals. While clocks may be off they are generally close enough so that the user will not notice.

#5 Anything you can think of. If you have a good idea run it past the net members and see if it makes sense.

#2 Don’t block on either Network or User Input

Server:
game play
  listen to clients (whether they sent any or not) if
    they sent none just tell the clients that no one moved)
  process data
  send all data out to all players

Client
game play loop
  draw display
  check if the user typed anything, if not keep truckin
  send user data - if no move send no move.
  get data from server, if no data check user.
  process data - only if data is there
    move objects - if you have to

Now this model is fast! No one waits for anyone. Data is sent back and forth whether or not anyone moved and everyone receives updates often.

But we have a problem. At these rates someone on a slow machine would be swamped and in trouble. Their machine would not be processing as much as everyone else and would be missing frames or processing them slowly. The game could easily advance away from them and they would never know.

We will discuss this more in the section for synchronization.

Summary

The idea to understand is that Blocking involves creating a bottleneck, that nothing can get past, and not blocking could mean uncontrolled chaos.

What we need is a way of losing the bottleneck while insuring everyone stays in time with each other.

#1 Game play blocks on both Network and User Input.

Server:

```
game play loop
  wait for data from all players, blocking on each NetworkStream
  process data
  send all data out to all players
```

Client:

```
game play loop
  draw display
  block on user input - doesn't continue until key pressed.
  send user data
  get data from server (blocking)
  process data
    move objects
```

Now examining this set-up there are good points and bad. The good is that this game insures complete consistency. By waiting for everyone before sending out information the server insures it got all the signals and that everyone else will receive them. Everyone will be in sync as each frame advances when the server sends the information.

This model is also horrible because it waits for every player to press a key before advancing a frame. If a player does not move then the game grinds to a halt. The client will not send data to the server, and the server will not advance frames because it hasn't heard from everyone.

Blocking:

What is blocking? Well if you play football you understand it is one of the most important things on the planet. If you hack net-code you know it is almost as important.

A process "blocks" if it halts until some particular event occurs which then wakes the process up so that it may proceed. Keep in mind that no other computation can take place in a given process while it is blocking.

Examples:

Blocking on stdin means waiting for a key to be pressed.

Blocking on a socket (or networkStream) means waiting for a message to be sent from the other side of that stream.

So how should I block? In our game (and in most applications) there are two extreme choices. All of our situations will begin in the same manner. I have chosen to demonstrate this with a star for no reason but simplicity, most of the principles cross over to other topologies.

Server startup:

command line option tells number of players, n
  create a socket on a well-known port
  waits for n players to connect
    share initial game information with player (map, player ID, position)

Client Startup

command line specifies server machine
startup
  create a socket on well-known port to server machine
  share initial game information with server (map, player ID, position)

At this point every client has connected to the server and has all of the initial info. Now we shall get to the blocking.

VisualMaze:

Files: In directory /course/gs007/study/systems/code
VisualMaze.h VisualMaze.cc

VisualMaze is a wrapper around the curses library. It allows the user to easily visualize the maze world and draw symbols on the screen with no difficulty.

The visual calls are simple and documented in the VisualMaze.cc file. So we only explain the I/O functions here.

`GetInput():` Curses will capture all of your keystrokes once initialized. There is no need to use either `cin` or `stdin`. When the program needs to check for user input you call this function. If the user has hit a key that key will be returned. If the user has not, one of two things can happen. It will either block (which means wait until it receives input) here that means stopping your game until someone presses a key. Or it will return the value `ERR`.

`DontBlock(bool choice)` is the function which decides whether `GetInput` blocks or not. It defaults to blocking mode. If called and passed a TRUE GetInput will no longer block. If passed `FALSE` it will block.

`GetConnection()` returns the file descriptor for keyboard input. You may wish to use select() on both this file descriptor and your networking file descriptors as a means of controlling blocking as described later.

NetworkHost is very simple just construct it with the name of the machine you want to talk to. Note: Do not use SunLab abbreviation, like `9a` instead of `cslab9a`. They will not work. It is used to construct a NetworkStream.

NetworkStream is where all the action is. This class is what you will use to communicate between processes. To connect to another machine construct one with the target machine's NetworkHost (see above) and its well-known-port number, that you have already selected.

To read from the NetworkStream call `read(void* buf,size_t nbytes)`, it will return the number of bytes it has read. `write` is the same process except it returns the number of bytes written.

```
// writing to a another machine!
stream.write("A", sizeof("A"));

// reading from that machine
char buff;

stream.read(&buff, 1);
//Now buff contains the char A.
```

As mentioned in class ints work a little differently. We use stream operators:

```
int x = 0;

// Reading an int sent from another machine
stream >> x;

// Writing an int to another machine
stream << x;
```

Maze-Net

Due Date: February 15, 1996 at 6:00
Handin: To be announced in newsgroup

Specifications:

Multiple processes (clients) are able to startup and communicate over the net allowing them to interact in a maze world visualized using VisualMaze.

There must be at least three processes with at least three players moving within the maze and having their user's actions mirror on every other client. Game play should not halt if the user chooses not to move in a given time interval; the screen should still update if other players move.

Network Package:

Files: In directory /course/gs007/study/systems/code
Network.h Network.cc

Network is a wrapper around an implementation of TCPIP sockets. It has everything you could ever need for this project (and most others) except `select`. It has three classes:

NetworkServer - manages server sockets
NetworkHost - Internet Address
NetworkStream - actual socket type class (works like a file)

Network Server is constructed with a port, and a char*. The port is the well-know port (a number higher than 5000 but four digits) and the char* is used for debugging purposes Network may print for you.

There is only one function you really need to know about with Network-Server. This is `waitForConnection()` which does exactly what it says. Once called it blocks until another process makes a socket connection to that machine and port number (until it gets a call). Once it receives its call it returns a fully working socket-structure, NetworkStream (which must be deleted when the game is over.)