HW3: Sweethand

Assignment written by Julie Zelenski

Your next assignment will be a chance to explore the issues in writing programs that create multiple threads and synchronize their actions. Before starting, familiarize yourself with the thread package— see our handout or check out `/usr/class/cs107/include/thread_107.h` and look over the examples we have gone through in lecture. Remember the thread package is specific to the Solaris Sparcstations, so you'll need to do your work on `epic`, `elaine`, `myth`, or `saga` machines.

Due: Monday, November 13th at 11:59 p.m.

Your task is to put your fine coding skills and new-found appreciation of concurrency to use in developing a marvelous little utility for keeping tabs on the usage of the leland workstations by CS107 students. Imagine you're about to sit down to do some serious coding and you need to decide which of the machines to use for your attack. "Elaine-best" and so on gets you to the least loaded machine, but you want something a little more specific. You want to find the machine that has the least, or hopefully no, CS107 students logged on so you can completely monopolize its resources. Thus enters **sweethand**…

You may know about the "sweetfinger" command that finds if a particular user is logged in on any of the leland workstations. Our improvement to this is to extend the search to look for not just one user, but a whole list of users, think of it as a "handful of fingers" if you will. If you have a list of all the 107 students and put sweethand to checking up on their status, you can get a full picture of classmate activity across all machines.

Threads are well-suited to this sort of distributed activity. Most of the time taken by fingering is spent waiting for a network response while the CPU is idle, so it is especially effective to have other threads running in the meantime making requests of their own. If it takes ~3 seconds for a host to respond to a finger request, a single-threaded program that sequentially sends out 20 finger requests, waiting for each response before sending the next, will take over a minute to complete. If we dispatch 20 threads in parallel, each of which makes one request and waits for the results concurrently, the whole program can complete in little more than 3 seconds total. How snappy!

We have written all of the network communication code that is needed, so there is no need to fear that aspect of the program, just call our routines at the right time and instead focus your energies on managing the threads and synchronization. Here is a sketch of the components of sweeethand:

Dispatcher

The dispatcher is the main controlling thread of the entire operation. It is responsible for first scanning the hostnames from a file, then pulling the real names from a file, doing the Whois lookup described below and spawning the username thread for each. The dispatcher is also responsible for making sure the socket threads and scoreboard thread exit after all the results have been computed.

Whois lookup

The list of people to finger is given in a file containing real names, e.g. “Jerry Cain”. One of the tasks for the dispatcher is to scan each real name from the file and convert from full name to leland username using our provided Whois function. Once a name has been converted, the dispatcher spawns a username thread to coordinate the task of fingering that user on all the hosts. The dispatcher then can move on scanning and translating the next name, allowing the Whois lookups and the username threads to operate concurrently.

Username thread

A username thread is responsible for setting up all of the finger requests for one username. It iterates over all hosts, arranging for a finger request on each host. To arrange for a finger request, the username thread will acquire the rights to use of one the sockets (described below) and then coordinate with a socket thread to initiate the network finger request. After the username thread does one host, it moves on to the next hostname, allowing all the finger requests to operate in parallel. (The socket threads actually do the fingering work). When all finger requests for the user are done, the username thread exits.

Finger request

A finger request is the act of sending a single username to a single host to get information about that user on that host. The request is sent out over the network using our provided network function. If all goes well, the count of the number of times that user is logged in on the host will be returned. If experiencing network problems or machine downtime, the request times out and returns an error response, in which case, you will re-send the request. A finger request is attempted four times before giving up. Finger requests are sent by the socket threads (described below).

Host address lookup

To make a finger request, you must translate the human-readable hostname to its network address. We provide the LookupHostAddress function that will contact the necessary network name server to obtain the address for a host. Since the lookup is somewhat expensive, you will translate each hostname only once and cache the address to be repeatedly used later.

Sockets

A "socket" is a communication channel, a connection to the Internet that is used to send and receive messages. Opening the socket establishes the network connection and makes it available for sending requests. An opened socket is only available to exactly the thread that opened it— it is a fatal error for Thread B to try to send over

a socket opened by Thread A. The socket should be closed when no longer needed. We provide the necessary open and close functions. There is a limit on the number of simultaneously open sockets.

Socket threads

Rather than repeatedly opening and closing sockets, a socket should be opened once when first needed and then kept open and re-used until no longer needed at the end. However, since an opened socket is available only to the thread that opened it, this necessitates that a thread must be dedicated to managing that socket for the lifetime of the program. Thus when a socket is needed, a separate "socket thread" is spawned that opens the socket and manages activity on that socket. When the username thread wants to make a finger request, it will need to acquire exclusive access to a socket and then coordinate with the socket thread to send the desired request on its behalf. Since all finger requests are made by the socket threads, the bulk of the real work is handled by these threads.

Scoreboard

The “scoreboard” data structure tracks the accumulated table of results of users per hosts. As finger requests are made and data comes in, the counts are accumulated in this table. It can take a while to do the entire cross-product of on all 107 students for all hosts. To gain a sense of the progress as the work goes on, a scoreboard display thread periodically prints out a progress summary. The scoreboard thread also prints out the final sorted results at the end.

Here are the more specific details about what should happen in the program:

Scanning all hosts

The first thing the dispatcher thread does is read the leland hostnames one by one from a file (using the Scanner, of course) and store them in an array for easy access later. The number of hostnames is guaranteed to be 200 or less so you can statically allocate the array to this fixed maximum size. After reading the host names, the dispatcher starts up scoreboard thread that will provide the running commentary on the ongoing progress and then the dispatcher moves on to the task of scanning and translating the names.

Real name to username translation

After the initial setup, the dispatcher thread opens the users file to read the names to finger. It reads in each real name and converts it to leland username by calling our provided function:

```
bool Whois(const char *realName, char username[], int usernameLen);
```

Whois looks up the given real name, and writes the matching leland username into the passed-in character array. The length of the array should be at least 9 characters long to hold any username. The function returns true if it found a match and wrote the username into the array, false otherwise.

If the real name was successfully converted, the dispatcher spawns a new username thread to look up this user on all the hosts. The username is passed to the new thread which is responsible for fingering that user on all hosts. The dispatcher does not wait for the new username thread to finish before moving on, it immediately goes back to scanning and converted more names, and so on.

Note that you can create a thread from within a thread and since `RunAllThreads` has already been called at that point, the new thread will immediately be added to the ready queue. This means that the newly created thread can start executing right away, potentially even waylaying the parent thread (the one that created the new thread). Also note that the parent doesn't block, waiting for that thread to finish, it competes for the processor along with everyone else.

The bulk of the dispatcher's work is scanning the real names, converting them, and dispatching their username threads. After that is done, the dispatcher has nothing to do until all of the fingering activity completes. At that point, dispatcher is then responsible for notifying the scoreboard and socket threads that it is the appropriate time to exit.

Username thread

One username thread is spawned by the dispatcher for each username to be fingered. This thread is responsible for arranging for that user to be fingered on each of the hosts. Arranging for a finger request means translating the host name to its network address and coordinating with a socket thread who does the actual finger operation. The username thread arranges for the user to be fingered on all hosts, then waits until all requests complete so it can exit.

Hostname to address translation

Before setting up a finger request for a host, the username thread must obtain the host's network address. This is straightforward if a previous username thread has done the translation and stored the result. If not, the username thread calls our provided function:

```
int LookupHostAddress(const char *hostname);
```

This function will contact the network name server to obtain the Internet address for the host and return it. If no such named host is found, the function returns –1 as an error code. Since doing this translation is expensive, the hostname should be looked up exactly once. The first username thread that needs the translation is responsible for storing the result so that it is accessible for the later threads who need it. Make sure that if two usernames threads both come looking for an as-yet-untranslated host that only one of them does the lookup, not both accidentally.

If a hostname is not known to the name server or there was some other error in translating the name, that host is marked unreachable (see section on unreachable hosts below) and no finger requests will ever be sent to it.

Finger requests

Once the host address has been determined, the username thread needs to arrange for a finger request on that host. To do so, the username thread must first gain control of one of the sockets. If all open sockets are in use, but we have not yet reached the maximum number of opened sockets, the username thread can spawn a new socket thread that will open another socket. If all open sockets are in use and the maximum number of sockets are already opened, the username thread must wait for an in-use socket to become available.

This act of acquiring a socket is an interesting synchronization point. The username thread wants exclusive access to a socket, any socket, and should not wait for a particular socket. The username thread should efficiently block when no sockets are available. When a socket becomes free, the username thread should snap it up. It may be that the username thread will have to search the set of sockets to find an available one. If many sockets are available at once, multiple threads might be looking for a socket at the same time. Make sure that two threads cannot assign themselves the same socket in that case.

Once a socket has been acquired, the username thread needs to provide information to its socket thread about the username and host address since it is the socket thread that must actually send the request over the opened socket. How to communicate this information safely between threads requires a bit of careful planning, but is similar to the clerk-manager inspection from the store example from lecture.

After setting up the request for the socket thread to perform, the username thread doesn't wait for the results, it goes on to arrange the request for the next hostname and so on, until all requests have been initiated. This makes it possible for all the finger requests to go on in parallel. The username thread waits for all of the requests to complete so that is can report back to the dispatcher that this username has been handled. The username thread then exits.

Socket threads

There are a fixed number of sockets (64), identified by number from 0 to 63. No sockets are open at the beginning of the program and there are no socket threads running. When a username thread needs a socket to make a network request, it assigns itself exclusive access to an available socket number. If that socket is not yet opened, a new socket thread is spawned to open and control that socket, otherwise the username thread coordinates with the already-running socket thread.

To open a socket, the socket thread calls our provided function:

```
void OpenSocket(int socketNum);
```

Once it has opened its socket, the socket thread's task is largely to sit in a loop, waiting for a username thread to initiate a finger request. At that time, the socket thread gets information from the username thread about the username and host address to finger. Then the socket thread calls our provided function to make a network finger request:

```
int SendFingerRequest(const char *user, int hostAddress,
                      int socketNum, bool isResend);
```

This function uses the opened socket to send a request to the given host address to get information about the given user. The isResend parameter allows you to indicate whether this is the first attempt or a re-send of this request. If it is a re-send, the timeout is increased slightly to allow for slower responding hosts. The host address should be a result previously returned from the host lookup function. The socket should be a valid socket number in range for the number of sockets, it must be an open socket and must have been opened by the same thread that is making the request.

The function returns the count of the number of times that the user is logged on to the given host and can be zero if the user is not logged on to that host. The socket thread is responsible for updating the count in the result table if the user was found to be logged on. For example, if we finger "xinh" on "epic5" and find that she is logged on 3 times, 3 is added to the user count for the "epic5" entry in the table.

In case of some sort of network error (host down, timeout, etc.), our function returns -1. On error, the socket thread should re-send in an attempt to get a valid response. After four unsuccessful tries, the socket thread gives up on this request. See the section on unreachable hosts below for what do when finger requests to a host repeatedly time out.

After a successful attempt or four unsuccessful ones, the socket thread completes the finger request and notifies the proper username thread. At that point, the socket thread goes back to waiting for another username thread to initiate another request. One tricky thing to work out is how each socket thread learns that all fingering activity is completed so it knows to close its socket and exit. It is the dispatcher who is responsible for coordinating the shutdown of the socket threads, but it will require a little thinking to see how to do so.

To close a socket, the socket thread calls our provided function:

```
void CloseSocket(int socketNum);
```

Simultaneous finger requests to the same host

Given the whims of the scheduler and our design, it is possible that all 64 sockets could be in use making simultaneously finger requests to the same host for different users. Rather than swamp any poor host with such a deluge of requests, you should maintain the constraint that at most 3 finger requests can be ongoing for one host at a time. Before a username thread claims a socket to start a finger request, it should check this constraint. If there are already 3 requests ongoing for the host, the username thread should efficiently block until one of the other requests completes and only then acquires the socket and initiate the finger request.

Unreachable hosts

If three different finger requests to the same host all fail (e.g. if `philips@epic9` fails all four attempts and then in a later finger request `kevinlee@epic9` fails in four attempts and then `bchatham@epic9` fails similarly), the host is marked as unreachable. Once a host has been marked unreachable, no new finger requests are sent to it for the duration of the program. Thus the username threads just ignore the host from now on. If earlier finger requests to a host were successful and then later it stops responding and is marked unreachable, you can maintain any previously counted users for that host or reset the count to zero, whichever is more convenient for you. Any host that failed to get a valid address in the lookup translation is marked unreachable from the beginning and no finger requests are ever sent to it.

Scoreboard thread

The last thread in the program is the scoreboard thread. The scoreboard thread is in charge of printing a regular status update as well as printing the final result at the end. Approximately every 5 seconds, the scoreboard thread prints a quick update of these statistics:

```
[May 1 11:22:59 AM] 812 finger requests completed (averaging 192 per sec)
```

The scoreboard sits in a loop, sleeping for 5 seconds, and then prints. The status line contains the current date and time (there is some sample code in the starting file to show you how to do this), the total number of finger requests made so far (i.e. the number of calls to our SendFingerRequest function, both successful and not) and the average requests per second (i.e. the number of total finger requests divided by the elapsed time). This last measure is a fairly rough approximation, since the scoreboard thread isn't guaranteed to print at exactly 5-second intervals and it starts a bit before the finger requests begin, etc., but it gives some idea of the overall throughput your program is achieving. Note that the throughput will vary widely depending on the machine load and network traffic.

After all finger requests have completed, the scoreboard prints out the total accumulated results:

```
-------------------- Hostname ---- Count -----------------
                     epic5           4
                     elaine10        3
                     myth3           3
                     saga16          2
                     saga18          2
                     elaine34        1
                     myth5           1
                     saga3           1

                     TOTAL          17
  [Unreachable hosts: elaine23 elaine29]
-------------------------------------------------------------
```

Each line has a hostname and the total count of cs107 users logged on to that hostname. The entries should be sorted in order of decreasing frequency, ties are broken by sorting alphabetically. At the end of the table, the names of the hosts that have been marked as unreachable are printed.

After sorting and printing the final table, the scoreboard thread exits. Probably the only interesting issue for the scoreboard thread is sorting out how the dispatcher thread can communicate to the scoreboard thread that all requests have completed so it knows to print the final results.

Global data structures

There is a quite of bit of shared data that you need to manage: the socket activity, the request results, the host addresses, etc. Much of this data will need to be global since it is shared by several threads. We haven't specified what form we expect your data to take, where the cached addresses are stored, where the error counts or results are tracked, but once you start to think about the problem, we are sure you can make good choices on your own. Be sure to organize all the necessary data sensibly and clearly using well-chosen identifiers.

Creating and managing semaphores

Given the presence of multiple threads simultaneously accessing the shared data, you are going to need several semaphores to coordinate access to this data and communicate between threads. It is recommended that you name each semaphore in a manner consistent with its usage to aid in readability. A binary semaphore used to provide mutually exclusive access should probably have a name of "somethingLock". A general semaphore used as a counter usually would have a name similar to how you would name an integer variable, e.g. "numSomethings". A semaphore used to rendevous between threads might have a name indicating the message that is being sent: "packetArrived", "finishedSearching", etc. Be sure to clearly comment how each semaphore is being used: what it is initialized to, who signals it when, who waits on it, etc. to document the flow of control throughout the program.

Don’t be stingy with creating semaphores where needed. Re-using the same semaphore for unrelated actions or to control access to too much data will unnecessarily serialize independent activity. At one extreme, if you have just one semaphore to lock access to all global data, then any action taken by a thread to access it will block out any other threads’ use. This will create one bottleneck that all threads have to go though, undesirably serializing activity that could and should be completely independent (such as one thread updating the host table while another is assigning itself a socket while a third is looking up a network address and so on). For example, consider the host data, likely kept in an array of structs with one entry per host. If one shared lock controlled access to the entire array it would lead to unnecessary constriction when several threads wanted to update information for different hosts. A per-host lock would be more appropriate here.

On the other hand, you might feel compelled to go overboard and have an independent semaphore for every piece of data and more. The presence of redundant or useless semaphores usually doesn't cause incorrect behavior, but it is important to consider what things actually need concurrency protection and what things do not. For example, you shouldn’t needs lock around local stack variables, since each thread has its own stack and thus has distinct local variables. If you have a lock that protects access to the entire struct, you probably don't need an inner per-field lock within to control access to a particular field. If some piece of data is only ead by multiple threads and never changed, a lock shouldn’t be needed. Like any other resource, you should be reasonable about your use of semaphores. If you have two pieces of closely related data that are never accessed independently, you can use just one semaphore to monitor both rather than maintaining two separate locks. And often in the context of some sort of iteration, you can repeatedly re-use semaphores instead of discarding ones no longer in use and creating new ones.

The program

Your program should take two arguments, the first is the file containing the real names, the second is the file with the hostnames. There are also optional “-t” and “-n” flags that can appear in the first argument slot to enable tracing in the network and/or thread library: These may be helpful for your debugging.

```
% sweethand –n names_107 hosts_all
```

After initializing your semaphores and setting up your global state, you create the sole dispatcher thread and call `RunAllThreads` to start it off. The dispatcher runs the show from there, spawning the scoreboard thread and the username threads, who in turn initiate the socket threads. When all threads finish, your main should print "`All done`" to announce your success!

Use symbolic constants for the parameters so you can easily change the number of sockets, the interval to print scoreboard updates, etc. It is easiest to debug with smaller

numbers early on. For example, you might work on successfully handling the one socket version as a first step.

Although we include the full files with the real names and hostnames, **we highly recommend that you do most of your development on small test files.** Edit a pair of test files to contain just a few users and few hosts so that you have something reasonable to work with. Then work your way up to running the full version to test the scalability of your program when it is further along. The full set of 25,000 finger requests takes several minutes to run. Although that works out to a lot of requests a second, it's still pretty awkward to work with when you're busy in development.

A suggested sequence of attack

Remember that your best strategy for handling a complex program is to proceed in stages and thoroughly test each piece before moving on. You are handicapping yourself if you try to write the entire program and then debug the whole thing at once. However, in a concurrent program often the pieces depend upon each other in a way that makes it difficult to sort out how you can develop in incremental stages. Here is a sketch of a sequence of steps we might recommend you take to complete the assignment successfully:

- Before you implement anything, take time to think carefully about your strategy. Sketching things out on paper, brainstorming to make sure your design handles all the cases, and settling on an approach that is correct and robust before you start coding can save you a lot of grief later. At each step, think carefully about what data you will need and how it would be best to organize and manage it.
- Familiarize yourself with what we provide. The starting program illustrates a sequential finger loop for one user across all hosts. This gives you basic information about how the username and hostnames are handled, the usage of the network library functions, and so on. Read more about the network function specifications from the `network.h` file in `/usr/class/cs107/include`. Compile the starting program and see what it does. Then run the compiled version in `hw3_sample` to see what your goal is.
- Start with the dispatcher thread and get it to scan all the hostnames in the host array. Next, scan in the names and do a `Whois` lookup on each (without any fingering).
- Extend the dispatcher to start up a new username thread for each username. The username threads don't need to do anything yet, perhaps just sleep for a bit, then check back in with the dispatcher.
- Have the dispatcher start up the scoreboard thread and have it run printing periodic "hello" messages until all username activity has completed.
- Extend the username threads to iterate over all hostnames just printing "finger username@HostAddress" after having done the hostname to host address translation. Make sure that the translation is done only once for each host!

- Now get socket assignment working, so that each username thread can efficiently and reliably gain exclusive access to a socket when needed. If the socket isn't yet open, arrange for the socket thread to be started.
- Now you're at the toughest part— coordinating the finger request between the username thread and socket threads. First, get the username thread to set up the socket thread to print the "finger username@HostAddress" and sleep for a bit to see that the right request is being communicated. Install the constraint that at most 3 simultaneous requests can be ongoing to each host.
- Now make the socket thread actually do the finger request, calling our function. At first, don't bother with storing the data, just print the result, then go back and extend it to record the counts in the table. Be careful to avoid conflicts on the shared data!
- Finish off by extending the scoreboard thread to print the proper periodic statistics and sort and print the final results.

Debugging and tracing

Most debuggers have problems working on multi-threaded applications and gdb is no exception. It can get confused in the presence of multiple threads, may not reliably stop at breakpoints, and can behave somewhat erratically about listing your source and printing the value of variables. There is an alternate debugger available (dbx) that is a bit smarter about working with threads if you'd like to try it. Unfortunately, just about every command in dbx is slightly different than its counterpart in gdb (break is stop in, delete is clear, etc.) so it is bit of pain to switch between the two. It has a decent man page, though, if you're game for checking it out.

A few known quirks of gdb to watch for: at times it will complain about trouble reading files ("premature end of symbol table" or "too many open files"), these messages can be ignored. Also gdb may print harmless "new LWP" messages as threads are created. If gdb seems to stop with a "SIGLWP" message at those times instead of continuing on, you need to be sure you have the .gdbinit file from the starting project directory which tells gdb to not bother stopping.

You may need to resort to inserting fprintf(stderr,...) to find out what your program is doing. You may leave debugging print statements in your code as long as it doesn't interfere with the readability and they are turned off in your final version. No printing beyond what is described in the assignment should be included when we run your program. You might find better to print to stderr than stdout— stderr gets flushed immediately to your terminal, while stdout may buffer output which can be confusing when debugging.

There is some debugging trace support in the thread package. If you give each thread an identifiable name when you create the thread, you'll be able to obtain the current thread name with a call to ThreadName and print it out. You can turn on thread tracing by

setting the debugging trace flag when you initialize the thread library. This will print out a running commentary on the threads as they wait, signal, and block on the various semaphores. Be warned though, the output can be prodigious.

The functions `ListAllThreads` and `ListAllSemaphores` may come in handy when you're trying to figure out the state of the world while in the debugger, so be sure to call to them when you are stopped and need more information. As mentioned in the suggestions for the factory warmup, calling these functions works most reliably if the program stopped by hitting a breakpoint. If the program was interrupted via Control-C, the functions may fail to execute because the context is not properly set up.

There is also some rudimentary trace support in the networking package as well. If you initialize the network package with the trace flag on, it will print send, receive, and timeout activity on the various socket connections to help you visualize what is going on.

General hints and guidelines

- Some of our examples declare semaphores as local variables and pass them as arguments to the threads and others set up the semaphores as global variables. We wanted to show you variety in the ways you can approach sharing data and semaphores between threads. If you can safely and easily avoid making things global, we prefer that, but globals are okay, too.
- It will be hard to avoid having at least some global variables in your programs— it is characteristic of multi-threaded programs that data is made globally visible to allow multiple threads to access it. However, don't indiscriminately promote variables from local to global. Only make data global that needs to be used by more than one thread.
- Be careful to organize any global data in sensible ways. Don’t create of parallel arrays or lots of bits and pieces of related info scattered randomly. Create structures with good names and logical organization to keep things tidy. For example, all of the data associated with a particular host entry could be organized into one structure.
- The `PROTECT` macro is for guarding system calls like `random`. It should not be used for other global variables, they should be guarded with their own semaphores. Using the library lock on simple variables is overkill and will inhibit concurrency through the whole program.
- There is a sample compiled version of sweethand in the hw3_sample directory. Run it a few times on different machines to get a sense of the runtime variations. For example, the throughput and time to completion can vary significantly due to the network activity. Also the hosts marked unreachable may change between runs. A host that is down or nonexistent will, of course, always be unreachable, but a live host that is temporarily slow to respond can be marked as unreachable. The timeout on our finger request function is set to be fairly short to help keep things moving, but it does mean it may assume unreachable a bit too rashly sometimes.
- Note that successfully running a concurrent program a few times doesn't at all guarantee its correctness, you must design for safety among multiple threads with a careful and logical strategy. Folks in previous quarters have said this program worked

best when they took time to map out their thread coordination arrangements before trying to implement anything. Think of each thread contention point as a little puzzle you have to solve and work through the solution logically on paper before you attempt it into code.

- Be sure to test your program on both a multiprocessor (saga) and uniprocessor (epic, elaine, myth). This may help you to spot lurking bugs that are only likely on one type of hardware.
- It is essential that your commenting include documentation about how you are synchronizing data access and coordinating among threads. If a function expects that a particular lock must already be held by calling thread, it should state so in its usage comment. If a thread waits on the activities of another thread, your comments should show the connection and explain the dependency. In rare instances, it may be correct or at least harmless to access a global variable without the protection of a lock. Any such unprotected access to shared memory must be documented with justification to receive full credit.
- In previous quarters, the concurrency programs turned in have been somewhat of a step backward in terms of previously solid areas like decomposition, data structures, and readability— lots of haphazard data organization, repeated code, overly long functions, etc. This quarter we have been very pleased with the quality of your submissions in these areas, and I'd like this to continue. Don't abandon good program design skills while learning new concepts!
- We expect your program to conform to our design specifications, especially with regard to which thread does what task. The given design was chosen so that you have to deal with a variety of concurrency issues, and give you a chance to learn how to work through and solve problems in the concurrent domain. Some of our choices may not lead to optimal concurrency in the final result, but that's okay.
- You'll need a working Scanner to read the names and hosts from the file. If you're not happy with yours, grab the scanner.o we distributed for hw1b testing and use that instead. We will not grade your scanner or allocate any points for it in this assignment.
- Folks in the past have been tempted to bring in the DArray and Hashtable to help manage the data structures. As handy as the ADTs are, I know using them as a client is far from trouble-free. I didn't seem right for pointer troubles to interfere with your learning about the concurrency issues. For this program, just go ahead and stick with simple fixed-size normal C arrays and use linear searches (lfind / lsearch are okay if you want), qsort, etc. as needed to work on the arrays.
- Freeing memory will be somewhat less important than it was on the earlier programs. We expect that you to be careful about your usage of memory and make a reasonable attempt to avoid leaks, but we won't nitpick over it.
- We're going to ignore Purify this time around. The thread library doesn't always interact well with Purify and since you won't use lots of dynamic memory or pointers in this program , I don't think you will miss Purify too much. (Now if only there was a "Purify" that checked your runtime use of semaphores...)

Getting Started

The starting project is in the leland directory `/usr/class/cs107/assignments/hw3`**.** This directory contains the buggy factory simulation program, a skeleton sweethand.c file, and a makefile that builds both projects. You want to make your own copy of the project directory to work on. You can also get the starting files via anonymous ftp to ftp.stanford.edu or from the class Web site `http://cse.stanford.edu/classes/cs107/`. Be sure to grab all the files, including the hidden dot-files in the directory that set up the debuggers to be a little more gracious about the thread usage.

Electronic submission

Electronically submit your entire project directory using the leland submit script. Maricia, Miler, and Erin all request hardcopies from their students, but Karen andEric prefer to grade the electornic versions. Make sure you get a hardcopy to your TA (or to me—I'm happy to get it to them) within 24 hours of electronic submissions.

Sample Sweethand Output

This is an excerpt of the ouput from sweethand with network tracing on (-n) using only 5 sockets. The trace messages printed by our network library are listed in columns according to the socket on which the activity is taking place and prefixed by the socket number in brackets. This is to help you follow what is going on and where. When a request is made on a socket, the network trace first prints the user@host on that socket column, then it will note when sending and when either a response is received or it times out. If successful, it prints the count for that user on that host.

From this you can get an idea of the interspersing of send / receive and how when a request takes a long time to process, a lot of other work can get done in the meantime.

```
elaine26% sweethand -n names_ta hosts_small
[S0] ajmiller@epic10
[S0] Send request
            [S1] ajmiller@myth5
            [S1] Send request
                           [S2] ajmiller@saga3
                           [S2] Send request
                                       [S3] ajmiller@elaine25
                                       [S3] Send request
                                                  [S4] ajmiller@elaine14
                                                  [S4] Send request
                                                  [S4] Response received
                                                  [S4] ajmiller@elaine14 = 2
                                                  [S4] ajmiller@epic5
                                                  [S4] Send request
[S0] Response received
[S0] ajmiller@epic10 = 0
[S0] ajmiller@myth1
[S0] Send request
                                                  [S4] Response received
                                                  [S4] ajmiller@epic5 = 0
                                                  [S4] ajmiller@saga7
                                                  [S4] Send request
                           [S2] Response received
                           [S2] ajmiller@saga3 = 0
                           [S2] bchatham@epic10
                           [S2] Send request
                           [S2] Response received
                           [S2] bchatham@epic10 = 0
                           [S2] gmh@epic10
                           [S2] Send request
                                                  [S4] Response received
                                                  [S4] ajmiller@saga7 = 0
                                                  [S4] kevinlee@epic10
                                                  [S4] Send request
            [S1] Response timed out
                                       [S3] Response timed out
                                       [S3] ajmiller@elaine25
                                       [S3] Re-send request
            [S1] ajmiller@myth5
            [S1] Re-send request
[S0] Response timed out
[S0] ajmiller@myth1
```

```
[S0] Re-send request
                          [S2] Response received
                          [S2] gmh@epic10 = 0
                          [S2] ajmiller@treex
                          [S2] Send request
                                                  [S4] Response received
                                                  [S4] kevinlee@epic10 = 1
                                                  [S4] bchatham@myth5
                                                  [S4] Send request
                          [S2] Response received
                          [S2] ajmiller@treex = 0
                          [S2] xfaz@epic10
                          [S2] Send request
          [S1] Response timed out
          [S1] ajmiller@myth5
          [S1] Re-send request
                          [S2] Response received
                          [S2] xfaz@epic10 = 0
                          [S2] xinh@epic10
                          [S2] Send request

[May  4 01:30:07 AM] 14 requests completed (averaging 2 per sec).

          [S1] Response received
          [S1] ajmiller@myth5 = 0
                                                [S4] Response received
                                                [S4] bchatham@myth5 = 0
```

… Intevening part of log removed for brevity …

```
---------------------- Hostname ----- Count ----------------------
                        elaine14        2
                        epic10          1
                        saga7           1

                        TOTAL           4
       [Unreachable hosts: none]
-----------------------------------------------------------------
All done!
```