Linux multithreaded server

I have some doubts regarding concurrency of posix threads in multiprocessor machine. I have found similar questions in SO regarding it but didnt find conclusive answer.

Below is my understanding. I want to know if i am correct.

1. Posix threads are user level threads and kernel is not aware of it.
2. Kernel scheduler will treat Process( with all its threads) as one entity for scheduling. It is the thread library that in turn chooses which thread to run. It can slice the cpu time given by the kernel among the run-able threads.
3. User threads can run on different cpu cores. ie Let threads T1 & T2 be created by a Process(T), then T1 can run in Cpu1 and T2 can run in Cpu2 BUT **they cant run concurrently**.

Please let me know if my understanding in correct

Since you marked your question with "Linux" tag I'm going to answer it according to standard pthreads implementation under linux.

1) Posix threads are user level threads and kernel is not aware of it.

No this is certainly not correct. The Linux kernel and the pthreads libraries work together to administer the threads. The kernel does the context switching, scheduling, memory management, cache memory management, etc.. There is other administration done at the user level of course but without he kernel, much of the power of pthreads would be lost.

2) Kernel scheduler will treat Process( with all its threads) as one entity for scheduling. It is the thread library that in turn chooses which thread to run. It can slice the cpu time given by the kernel among the run-able threads.

No, the kernel treats each process-thread as one entity. It has it's own rules about time slicing that take processes (and process priorities) into consideration but each sub-process thread is a schedulable entity.

3) User threads can run on different cpu cores. ie Let threads T1 & T2 be created by a Process(T), then T1 can run in Cpu1 and T2 can run in Cpu2 BUT they cant run concurrently.

No. Concurrent executing is expected for multi-threaded programs. That's why synchronization and mutexes are so important and why programmers put up with the complexity of multithreaded programming.

One way to prove this to you is to look at the output of ps with -L option to show the associated threads. ps usually wraps multiple threaded processes into one line but with -L you can see that the kernel has a separate virtual process-id for each thread:

ps -ef | grep 20587

foo 20587 1 1 Apr09 ? 00:16:39 java -server -Xmx1536m ...

versus

ps -eLf | grep 20587

foo 20587 1 20587 0 641 Apr09 ? 00:00:00 java -server -Xmx1536m ...

foo 20587 1 20588 0 641 Apr09 ? 00:00:30 java -server -Xmx1536m ...

foo 20587 1 20589 0 641 Apr09 ? 00:00:03 java -server -Xmx1536m ...

...

I'm not sure if Linux threads still do this but historically pthreads used the clone(2) system call to create another thread copy of itself:

Unlike fork(2), these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.

This is different from fork(2) which is used when another full process is created.

On my Linux server, I need to synchronize multiple scripts, written in BASH and PHP, so that only one of them is able to start a system-critical job, which is a series of BASH/PHP commands, that would mess things up if performed simultaneously by two or more scripts. From my experience with multithreading in C++, I'm familiar with the notion of mutex, but how do I implement a mutex for a bunch of scripts that run in separate processes and, of course, aren't written in C++?

Well, the first solution that comes into mind would be making sure that each of the scripts initially creates a "lock flag" file to let other scripts know that the job is "locked" and then deletes the file after it's done with the job. But, as I see it, the file writing *and* reading operations are required to be *completely atomic*to let this approach work out with a 100% probability, and the same requirement would apply to any other synchronization method. And I'm pretty sure that file writing/reading operations are not atomic, they are not atomic across all existing Linux/Unix systems at least.

So what is the most flexible and reliable way to synchronize concurrent BASH *and* PHP scripts?

I'm not a PHP programmer, but the documentation says it provides a portable version of [flock](http://www.php.net/manual/en/function.flock.php) that you can use. The first example snippet looks pretty close to what you want. Try this:

<?php

$fp = fopen("/tmp/lock.txt", "r+");

if (flock($fp, LOCK\_EX)) { // acquire an exclusive lock

// Do your critical section here, while you hold the lock

flock($fp, LOCK\_UN); // release the lock

} else {

echo "Couldn't get the lock!";

}

fclose($fp);

?>

Note that by default flock waits until it can acquire the lock. You can use LOCK\_EX | LOCK\_NB if you want it to exit immediately in the case where another copy of the program is already running.

Using the name "/tmp/lock.txt" may be a security hole (I don't want to think hard enough to decide whether it truly is) so you should probably choose a directory that can only be written to by your program.

 I have the following ideas of how to implement a multithreaded TCP server in LINUX.

1. Have a listener thread that creates, binds, listens and accepts connections, then spawns a new pthread to listen on the connection for incoming data and close connection after an idle timeout period. If the number of active threads is currently 2, new connections are instantly closed to ensure only 2 are allowed.
2. Do not spawn new threads from the listener thread, instead use select() to detect incoming connection requests as well as incoming modbus connects on active connections (similar to the approach in Beejs guide).
3. Create 2 listener threads each of which creates a socket (same IP and port number) which can block on accept() calls, then close the socket fd and deal with the connection. Here I am (perhaps naively) assuming that this will only allow max of 2 connections which I can deal with using blocking reads.

I have been using C++ for a long time but I am fairly new to Linux development. I would really welcome any suggestions as to which of the above approaches is best (if any) and if my inexperience with Linux means that any of them are really really bad ideas. I am keen to avoid fork() and stick to pthreads as incoming modbus requests are going to be queued and read off a main controller loop periodically. Thanks in advance for any advice.

The third alternative won't work, you can only bind to the local address once.

I would probably use your second alternative, unless you need to do a lot of processing in which case a combination of the first to alternatives might be useful.

The combination of the two first alternative I'm thinking of is to have the main thread (the one you always have when a program starts) create two worker threads, then go a blocking accept call to wait for a new connection. When a new connection arrives, tell one of the threads to start working on the new connection and go back to block on accept. When the second connection is accepted you tell the other thread to work on that connection. If both connections are open already, either don't accept until one connection is closed, or wait for new connections but close them immediately.

I am writing a server in linux that is supposed to serve an API.

Initially, I wanted to make it Multi-threaded on a single port, meaning that I'd have multiple threads working on various request received on a single port.

One of my friends told me that it not the way it is supposed to work. He told me that when a request is received, I first have to follow a Handshake procedure, create a thread that is listening to some other port dedicated to the request and then redirect the requested client to the new port.

Theoretically, it's very interesting but I could not find any information on how to implement the handshake and do the redirection. Can someone help?

What your friend told you is similar to passive FTP - a client tells the server that it needs a connection, the server sends back the port number and the client creates a data connection to that port.

But all you wanted to do is a multithreaded server. All you need is one server socket listening and accepting connections on a given port. As soon as the automatic TCP handshake is finished, you'll get a new socket from the accept function - that socket will be used for communication with the client that has just connected. So now you only have to create a new thread, passing that client socket to the thread function. In your server thread, you will then call accept again in order to accept another connection.

If I'm not wrong in interpreting your responses, once I create a multithreaded server with a main thread listening to a port, and creates a new thread to handle requests, I'm essentially making it multithreaded on a single port?

Consider the scenario where I get a large number of requests every second. Isn't it true that every request on the port should now wait for the "current" request to complete? If not, how would the communication still be done: Say a browser sends a request, so the thread handling this has to first listen to the port, block it, process it, respond and then unblock it.

By this, eventhough I'm having "multithreads" , all I'm using is one single thread at a time apart from the main thread because the port is being blocked.

Comments

Two (or more) connections can be active to the same local port at the same time - there's absolutely nothing wrong with this, and it's the usual way things work. One connection doesn't "block" any of the others - once created, they're completely independent.

Hi. Thanks for the comment. I completely get the point you are making and in fact, is what I'm getting confused with. If more than one connections are actively using one port, implying that two different clients are connecting on the same port, how would reads/writes be done correctly and how is information sent to the right client? I've tried reading some books/articles on sockets and server programming, but realized that I'm searching for a solution in the wrong place. Can anyone suggest a neat-small-quick guide on this topic

From the application's point of view, the two connections are represented by different sockets - different file descriptors. From the transport's (TCP/IP) point of view, the two connections are identified by different (remote IP, remote port, local IP, local port) tuples (one or both of the remote parts of the address will be different)

What your colleague is suggesting sounds like the way FTP works. This is not a good thing to do -- the internet these days is more or less used for protocols which use a single port, and having a command port is bad. One of the reasons is because statefull firewalls aren't designed for multi-port applications; they have to be extended for each individual application that does things this way.

I have linux c++ multithreaded application. Now it's tested on production servers and have segfault. Problem is that I cannot reproduce that error on any of my test servers and have no access to production servers. I have no dump or any other useful information. Only line: segfault at 0000000046bf0fb8 rip 000000000048ac6b rsp 0000000046bf0fa0 error 6

I would like to ask community can I get from such line some information which will help decrease area of possible places where I should search. I cannot run debug build on production because of its slow speed. What can I add to release which help me debug? This bug looks like multithread bug, and hard to reproduce. But I'm not sure, because application works with a lot of different emails from MTA.

Platform: Linux

Compiler line: g++ -O3 -D\_REENTRANT

Thank you.

upd.: Thanks for your answers. I can include debug information. I'd like to know base methods of debugging release builds. For example I have dump and release version. How should I continue. What should I read about that? Can you explain in few words how you debug your application if possible? Thank you.

I've been reading the gdb manuals recently, and they recommend leaving the debugging symbols in e.g.g++ -g.

Since you don't have access to the production server maybe include some basic logging functionality that'll output a data to a text file. You should be able to narrow down roughly where the error occurs, depending on which data has been output to your log file.

As Andy mentioned, leave the debugging symbols in when you make your release builds.

If this makes the size of the finished executable unacceptably large, then you can make a copy of the final executable and run it through strip to remove the debug symbols. This way you have *two*executables that are identical except one has debug symbols and the other does not. Put the one without symbols on the production server. When it segfaults, debug against the copy of the executable that still contains debug symbols.

You can (and should) build release executables with debug information. If you don't want to distribute executables containing the debug information, then you can [separate](http://stackoverflow.com/questions/866721/how-to-generate-gcc-debug-symbol-outside-the-build-target/866731#866731) the debug information and install it later for debugging. That's what we do in our application.

I have a program launching about 12 threads on 8-core server. some of the threads are waiting for critical data. where it's using a recv in loop, the recv will block until data comes. However when there is data, it needs to process it asap.

one thing I noticed is that from time to from when the connection is quiet, the thread doesn't have much activity. the thread maybe put to sleep (suspect?), then when data comes in, it needs to wake up first, therefore wasting time. I'm wondering if there is anyway to set so the thread will not be put to sleep and wake up later? thanks!

It's not just your threads, there are other threads too. Within a large range it does not matter how many thread you actually have. Other threads are likely to get choosen by the scheduler while your thread is waiting on the recv. So when there is data to let your recv return your thread will get ready to run and may be choosen by the scheduler for execution. If all cores are occupied by threads of equal of higher priority than your thread and if their time slice not just luckely ends, your thread would have to wait for the cpu. However, in order to get your thread scheduled **immedeately** after the recv has data, you should raise your threads priority. In this case the recv return will make your thread "runnable" and the scheduler will swtich to it before it considers any other threads of lower priority. If neccessary it will even stop a thread of lower prioriy of your thread.

Comments

That's the way these preemptive systems work - the thread calls recv() and blocks. recv() returns when there is data, (or conn closed, or error). Time is not wasted - while the recv() thread is 'sleeping', the CPU it would have used is available for other things, eg. running the network stack so that you get your data as fast as possible

i see. as I have more threads than cores, will it cause contention? or should I only launch 8 threads on the 8 core server?

If you're just waiting for requests, processing them in a CPU-bound way, then responding, there's probably no point in more threads than cores. But if you are receiving a request, waiting on a non-CPU resource (say, local disk or sending a request to another server and waiting for a response), then more threads should help. Of course, you can always measure and know for sure in your environment

There appear to be several options available to programs that handle large numbers of socket connections (such as web services, p2p systems, etc).

1. Spawn a separate thread to handle I/O for each socket.
2. Use the [select](http://linux.die.net/man/2/select) system call to multiplex the I/O into a single thread.
3. Use the [poll](http://linux.die.net/man/2/poll) system call to multiplex the I/O (replacing the select).
4. Use the [epoll](http://linux.die.net/man/4/epoll) system calls to avoid having to repeatedly send sockets fd's through the user/system boundaries.
5. Spawn a number of I/O threads that each multiplex a relatively small set of the total number of connections using the poll API.
6. As per #5 except using the epoll API to create a separate epoll object for each independent I/O thread.

On a multicore CPU I would expect that #5 or #6 would have the best performance, but I don't have any hard data backing this up. Searching the web turned up [this](http://lse.sourceforge.net/epoll/index.html) page describing the experiences of the author testing approaches #2, #3 and #4 above. Unfortunately this web page appears to be around 7 years old with no obvious recent updates to be found.

So my question is which of these approaches have people found to be most efficient and/or is there another approach that works better than any of those listed above? References to real life graphs, whitepapers and/or web available writeups will be appreciated.

I think this is a solved problem and the answer is here - <http://www.kegel.com/c10k.html>

Comments

 agree with @Seun - I have read the C10K document numerous time, each time spending more time with it than before, in order to really grok the issue. Well, I can't say that author tells of their preference for either approach, and in fact it's not clear whether they intended to. The person asking this question however, wants an opinion from someone who would know better, not a menu of options (which C10K looks like, aside from references which indeed may provide further clues.)

Speaking with my experience with running large IRC servers, we used to use select() and poll() (because epoll()/kqueue() weren't available). At around about 700 simultaneous clients, the server would be using 100% of a CPU (the irc server wasn't multithreaded). However, interestingly the server would still perform well. At around 4,000 clients, the server would start to lag.

The reason for this was that at around 700ish clients, when we'd get back to select() there would be one client available for processing. The for() loops scanning to find out which client it was would be eating up most of the CPU. As we got more clients, we'd start getting more and more clients needing processing in each call to select(), so we'd become more efficient.

Moving to epoll()/kqueue(), similar spec'd machines would trivially deal with 10,000 clients, with some (admitidly more powerful machines, but still machines that would be considered tiny by todays standards), have held 30,000 clients without breaking a sweat.

Experiments I've seen with SIGIO seem to suggest it works well for applications where latency is extremely important, where there are only a few active clients doing very little individual work.

I'd recommend using epoll()/kqueue() over select()/poll() in almost any situation. I've not experimented with splitting clients between threads. To be honest, I've never found a service that needed more optimsation work done on the front end client processing to justify the experimentation with threads.

The program is a client server socket application being developed with C on Linux. There is a remote server to which each client connects and logs itself as being online. There will be most likely be several clients online at any given point of time, all trying to connect to the server to log themselves as being online/busy/idle etc. So how can the server handle these concurrent requests. What's a good design approach (Forking/multithreading for each connection request maybe?)?

personally i would use the event driven approach for servers. there you register a callback that is called as soon as a connection arrives. and event callbacks whenever the socket is ready to read or write.

with a huge amount of connections you will have a great performance and resource benefit compared to threads. But i would also prefere this for a smaler count of connections.

i only would use threads if you really need to use multiple cores or if you have some request that could take longer to process and where it is too complicate to handle it without threads.

i use [libev](http://software.schmorp.de/pkg/libev.html) as base library to handle event driven networking.

I'd like to develop a multithreaded UDP server in C/Linux. The service is running on a single port x, thus there's only the possibility to bind a single UDP socket to it. In order to work under high loads, I have n threads (statically defined), say 1 thread per CPU. Work could be delivered to the thread using epoll\_wait, so threads get woken up on demand with 'EPOLLET | EPOLLONESHOT'.

As you are only using a single UDP socket, there is no point using epoll - just use a blocking recvfrom instead.

Now, depending on the protocol you need to handle - if you can process each UDP packet individually - you can actually call recvfrom concurrently from multiple threads (in a thread pool). The OS will take care that exactly one thread will receive the UDP packet. This thread can then do whatever it needs to do in handle\_request.

However, if you need to process the UDP packets in a particular order, you'll probably not have that many opportunities to parallalise your program...

I have been in an assumption that multithreading gives a lot of flexibility and power to make efficient programs. But I am wrong. Multithreading is sometimes not desirable as it will create serious impact on the way the actual program behaves.

**My question?**

*I am not sure of when to use and when not to use multithreads in applications. Please update this thread and give your opinions on multithreading and suggestions on its usage scenarios.*

***when to use and when not to use multithreads***

Lets look at a simple problem: Sorting.

Now we can design a sorting algorithm that, given an element in a dataset (array), looks at that entire dataset and computes where the given element belongs in the final sorted output. It's straightforward enough to do. Just count how many elements before the chosen element are less-than-or-equal to the chosen element, and how many elements after it are less-than the chosen element.

The downside is that's it's O(N^2). For large N, computation times are huge.

But **each element can be computed (sorted) independently.** This algorithm threads very nicely, with little need for cross-communication (semaphores/mutexes) between threads.

Contrast this with quicksort, a far more efficient O(NlogN) algorithm. But how would you thread quicksort? Probably some sort of work-queue, with mutexing and semaphores and whatnot and threads waiting around for work.

**That sitting around waiting for mutexes to clear, or semaphores to be tripped, is inefficient. With widespread inefficiencies, you lose the advantage of threading.**

You may have 4 or 6 cores, but with the inefficiencies involved you may only see a 2x or 3x speedup. (Or less. Or more...)

Worse still is when you don't properly mutex (or use semaphores) (e.g. improper thread boundaries), and you wind up with unreproducible race conditions that are all but impossible to track down. You can say something will only go wrong 1 in a million times. But on modern computers, that will happen far too frequently. It only takes a few such bugs to render an application useless.

Then there are the hidden "gotchas" like **livelock**. (Your philosophers are starving.) One of the classic livelock's I've seen is where folks unlock a mutex in one thread and subsequently reacquire (re-lock) it before any other thread gets a chance to run.

**Starvation** is another issue. For instance, when one thread uses a lot of CPU time and another yields (intentionally or otherwise) itself far too frequently.

Still another involves thread stacks, which may become quite limited. With lots of threads, it's easy to run out of stack space. Especially under 32-bit OS's with their 4gig address space limitation per process (across all threads in the process).

These things can be dicey to figure out. There's a substantial time hit to implementing multithreaded code correctly, and even more so for debugging it!

In your case, you mention making huge numbers of http/udp requests of servers. On the order of 10,000. That's actually not that large a number.

Presumably for each request, you are looking at a timeout of some sort if it doesn't respond promptly. You'll want to be able to receive the data back. A simple *select()* statement in a single thread ought to cut it.

Depending on your system, you may want/need to increase the buffering size on each socket. (Just how big are the messages you are getting back? You mention UDP -- How are you dealing with out-of-sequence data, messages spread across multiple packets, or dropped packets?)

For that matter, have you attempted to measure how many UDP packets you can send per second from your system? How many *write()* calls you can make per second?

How about your receiving bandwidth? IIRC, UDP handles bandwidth shortages by dropping packets. If you are fetching webpages, some of those could be quite large. At 10,000 concurrent requests, you could be looking at a significant bandwidth hit.

Let me ask you this: What do you think multi-threading is giving you? Extra CPU processing power via multiple cores? How much work are you doing on each message you receive? (Code it up and time it! That's not difficult to measure. You'll probably want some metrics anyway in your system.) What's your mean time to send/receive to/from a server? 0.1 seconds? 0.2? 0.5?

Say it's 0.3 seconds. That's 300,000 microseconds. So with 10,000 messages coming in, you would be looking at 30 microseconds per message. (Or less after factoring in overhead.) Not a whole lot of time, even by modern computing standards.

FYI, typical linux kernels run in 10,000 microsecond slices. Though of course you can yield sooner if desired...

Running with 100 threads isn't magically going to create more CPU time. Quite to the contrary, you'll be eating up time with context swaps. Though if you had 4 or 6 cores, running with 4/6 threads would allow you to max out all the cores. Or perhaps 8/12 threads to max out performance while waiting for disk accesses or network responses.

Even so, you're still not dealing with a whole lot of time per message. On the order of 120/160 microseconds. (Or less after factoring in overhead.)

Depending on exactly what you are doing, it might be easier to build a single-threaded system and run 4 (or 6) copies of it. (One on each core.)

If you are thinking you want a single massive internal datastore, well then you're back into thread contention, mutexes, semaphores, and waiting... And you don't have a lot of time between messages to wait...

Comment

That's slow. You can use *select()* to block until data comes back in, or until you timeout. You can have a great many sockets in a single thread/process if you want (Though you may be able to send to multiple destinations from a single socket. And then determine where the responses came from when receiving. IIRC, *recvfrom()* has a *from* argument, and *recvmsg()* has similar abilities within *msg*. I believe UDP supports *from*, but I could be mistaken. It's been a while

I am interested in learning how to write extremely efficient network server software and I don't mind getting my hands dirty with pointers, sockets and threading. I'm talking a server being able to handle thousands of concurrent connections. There is not much processing for each client, but a little.

**Do you know of any code examples for really efficient network servers?**

Optionally points for small, well documented code that is cross-platform as well.

Have a look at [nginx](http://nginx.net/), [lighttpd](http://www.lighttpd.net/) and [varnish](http://varnish.projects.linpro.no/) for some popular high performance http servers.

BTW, I am currently working on combining edge-triggered epoll with multithreading (plus user-level swapcontext-style threads/fibers) - see <http://svn.cmeerw.net/src/nginetd/trunk/> for some work-in-progress code (although this one is written in C++).

You'll find a lot of good references and discussion about building highly scalable network servers on Dan Kegel's [The C10K problem](http://www.kegel.com/c10k.html) page.

I am trying to design a multithreaded web server in C using Pthreads and i am having a problem in accepting more incoming connections without serving them.

I want to put the file descriptor of each recieved connection in a buffer to be pocessed by a thread, Im using default accept(2) for accepting clients connections.

should i be using select ? any suggestion ?

A common way of doing multi-threaded servers is to create a new thread right after you accept a new connection, and pass the new socket to that thread. Something like this:

int main(int argc, char \*argv[])

{

/\* ... \*/

int client\_socket = accept(server\_socket);

pthread\_create(&thread, NULL, my\_connection\_handler, (void \*) client\_socket);

/\* ... \*/

}

void \*my\_connection\_handler(void \*argp)

{

int socket = (int) argp;

write(socket, "Hello!\r\n", 8);

close(socket);

return NULL;

}

Comments

Just be warned that while this is the simplest way and the way you should usually learn first, the performance is terrible. You have to create a new thread and wait for it to get the CPU before you can start servicing a new connection. And worse, to do a little bit of work on each of 100 connections requires 100 context switches. Also, if the threads wind up accessing common data structures, you get bad behavior from the synchronization on those data structures. – [David Schwartz](http://stackoverflow.com/users/721269/david-schwartz)

DavidSchwartz So if i used a fixed size pool of worker threads, and synchronize the shared memory using mutex or semaphores, it will start receving connectoin when ever one is passed to a worker thread

CodeRed: That's a common way to do it. It's still not great for many of the reasons I explained above -- it requires lots and lots of context switches and the large number of thread (one per client) makes synchronization inefficient. (For example, a bunch of data comes in, a bit for each of 80 clients, so all 80 threads wake only to wind up blocking on the same synchronization object which each of the 80 threads get in turn. You wind up with hundreds of context switches for no reason

I am writing application in Linux using C, pthreads and sockets.

This will be client-server application, server will have N+2 threads, where N - number of active clients, one thread for accepting new connections and creating threads for clients and last one will be accepting user input.

I will be using linked list to save some data that will be relevant to my application, with every client there will be associated one node in my list. Those client threads will update information that is stored in their nodes with some interval, could be one second, could be two minutes, it will dynamically change.

Now here is the problem, if user requests it, the information stored in linked list needs to be written to standard output. Of course during writing I should acquire mutex. I am worried that one mutex for whole list will hinder the performance.

I was thinking about associating mutex with every node, but it will complicate removal of some specified node (firstly, I would need to make sure that the 'stdout writer' thread won't be traversing the list, I would also need to acquire mutex of my node and the previous one to change the pointer that points to the next node and so on - either I would need to traverse all the way to the previous or I would need to make double linked list).

So I am wondering if the solution that involves multiple mutexes is even better with much more complicated code, conditions and all of this locking, waiting and unlocking.

You are right that having a per-node mutex will make code more complex. That's a tradeoff you will have to decide the value of. You can either have a single lock for the entire list, that might cause lock contention, but the code is largely not impacted by the presence of the lock and thus easier to write, or you can have more locks with considerably less opportunity for contention, leading to better performance, but the code is harder to write and get correct. You could even have something in the middle by having a lock per group of nodes - allocate a few nodes together and have a lock for that group - but then you'll have issues with tracking a free list and the potential for fragmentation.

You'll need to consider the relative frequency of add operations, delete operations, and full-list iterations, as well as others (reorganization, searching, whatever else your application will require). If add/delete are extremely frequent, but walking the list is once every third blue moon, the single lock could easily be appropriate. But if walking the list (whether for a full dump of the data, or to search or something else) is very common, the more granular approach becomes more attractive. You might even need to consider reader/writer locks instead of mutexes.

Comment

And what about this, instead of physically removing nodes I would marked them as 'unused', this way during the writing of the data and removal there won't be any problem because I won't need to access any other nodes, if the reader wants to traverse the list he would lock it, also list addition would only occur if the list is not being traversed by the reader. – [Andna](http://stackoverflow.com/users/1044110/andna)

You could do that as well - but consider whether it's worth it. Adding a new node would become searching for an unused node and then allocating one if there are no unused nodes. Traversing the list would have to be smart enough to skip over unused ones. Deleting would be somewhat easier, as you wouldn't have to adjust links, but you would still need to use a lock just to mark the node unused, so you might as well pull it out of the list anyway. All around, I think marking nodes as unused makes the code more complicated for not much gain. – [twalberg](http://stackoverflow.com/users/1253222/twalberg)

Ok, I think I will use read-writer locks, as far as I understand I can guarantee that writer will have permission over reader to access the list. – [Andna](http://stackoverflow.com/users/1044110/andna)

I am implementing a small database like MySQL.. Its a part of a larger project..

Right now i have designed the core database, by which i mean i have implemented a parser and i can now execute some basic sql queries on my database.. it can store, update, delete and retrieve data from files.. As of now its fine.. however i want to implement this on network..

I want more than one user to be able to access my database server and execute queries on it at the same time... I am working under Linux so there is no issue of portability right now..

I know i need to use Sockets which is fine.. I also know that i need to use a concept like Thread Pool where i will be required to create a maximum number of threads initially and then for each client request wake up a thread and assign it to the client..

As for now what i am unable to figure out is how all this is actually going to be bundled together.. Where should i implement multithreading.. on client side / server side.? how is my parser going to be configured to take input from each of the clients separately?(mostly via files i think?)

If anyone has idea about how i can implement this pls do tell me bcos i am stuck here in this project...

Thanks.. :)

If you haven't already, take a look at [Beej's Guide to Network Programming](http://beej.us/guide/bgnet/output/html/multipage/index.html) to get your hands dirty in some socket programming.

Next I would take his [example of a stream client and server](http://beej.us/guide/bgnet/output/html/multipage/clientserver.html) and just use that as a single threaded query system. Once you have got this down, you'll need to choose if you're going to actually use threads or use select(). My gut says your on disk database doesn't yet support parallel writes (maybe reads), so likely a single server thread servicing requests is your best bet for starters!

In the multiple client model, you could use a simple per-socket hashtable of client information and return any results immediately when you process their query. Once you get into threading with the networking and db queries, it can get pretty complicated. So work up from the single client, add polling for multiple clients, and then start reading up on and tackling threaded (probably with [pthreads](https://computing.llnl.gov/tutorials/pthreads/)) client-server models.

As an alternative to multithreading, you might consider event-based single threaded approach (e.g. using poll or epoll). An example of a very fast (non-SQL) database which uses exactly this approach is redis.

This design has two obvious disadvantages: you only ever use a single CPU core, and a lengthy query will block other clients for a noticeable time. However, if queries are reasonably fast, nobody will notice.

On the other hand, the single thread design has the advantage of automatically serializing requests. There are no ambiguities, no locking needs. No write can come in between a read (or another write), it just can't happen.  
If you don't have something like a robust, working MVCC built into your database (or are at least working on it), knowing that you need not worry can be a huge advantage. Concurrent reads are not so much an issue, but concurrent reads and writes are.

Alternatively, you might consider doing the input/output and syntax checking in one thread, and running the actual queries in another (query passed via a queue). That, too, will remove the synchronisation woes, and it will at least offer *some* latency hiding and some multi-core

Comments

yes actually for now i am not much bothered about MVCC and i dont have also one in my database code.. for now i think it would be nice as you mentioned running actual queries in one thread and syntax checking in another... Also about event based i'm looking into it.. may be it can just serve the purpose

Without MVCC, you must use reader-writer locks. Naively implemented (database-level or table-level locking) this will negate most if not all benefits of multithreading unless you have mostly reads with ony very few updates. Row level locking is better, but obviously more hassle, too (and, no panacea either). So, in that case, having everything serialized may indeed save some headaches. Do you, by any chance, already use Boost? If you do, you can look into asio for a no extra cost portable solution (though hand-coding epoll is really easy... only downside is it's not portable)

With a single file descriptor, is there any performance difference between select, poll and epoll?

The main difference between epoll vs select or poll is that epoll scales a lot better when run in a single thread. I don't know how this would compare to using a multithreaded server using select or poll. Look at this <http://monkey.org/~provos/libevent/libevent-benchmark2.jpg>

The reason for this(as far as I can tell) is that when you are using select or poll you must loop through all the connected sockets to determine which ones have data to be read. When you are using epoll, it keeps a seperate array which contains references only to sockets which have data to be read. This saves you lots of loop cycles, and the difference becomes more and more noticeable the more sockets that are connected.

Another thing to look into if performance ever becomes a major issue is io completion ports(windows only) and kqueue(FreeBSD only). It's also important to remember that epoll is linux only. In most cases select or poll will work just fine.

**In the case of a single file descriptor, select and poll are more efficient than epoll** due to being much simpler. (epoll has some overhead which doesn't make itself useful with only a single socket)

Im designing a multi cleint web server using multithreads in C using Pthreads, i have a Masterthread that is in a loop doing the listening, when there is a connection it spawns a new thread doing a function to serve, while the Masterthread continue listening for connection.

as i have notice from debuging, its serving only one connection at a time, the Accept() system call is waiting for that connection to close, then it will spawn the next connection in the queue.

Its acting like if its a single-thread web server

You observation is not correct, accept(2) does not wait for established TCP connection to complete. What you see is probably thread dispatch artifact - the main thread is preempted by the newly spawned thread and does not get to run until that one completes.

In general, thread-per-connection is a workable strategy, but you might want to create a pool of threads before accepting connection, instead of starting a new one every time

How to interrupt a thread performing a blocking socket connect?

I have some code that spawns a pthread that attempts to maintain a socket connection to a remote host. If the connection is ever lost, it attempts to reconnect using a blocking connect() call on its socket. Since the code runs in a separate thread, I don't really care about the fact that it uses the synchronous socket API.

That is, until it comes time for my application to exit. I would like to perform some semblance of an orderly shutdown, so I use thread synchronization primitives to wake up the thread and signal for it to exit, then perform a pthread\_join() on the thread to wait for it to complete. This works great, unless the thread is in the middle of a connect() call when I command the shutdown. In that case, I have to wait for the connect to time out, which could be a long time. This makes the application appear to take a long time to shut down.

What I would like to do is to interrupt the call to connect() in some way. After the call returns, the thread will notice my exit signal and shut down cleanly. Since connect() is a system call, I thought that I might be able to intentionally interrupt it using a signal (thus making the call return EINTR), but I'm not sure if this is a robust method in a POSIX threads environment.

Does anyone have any recommendations on how to do this, either using signals or via some other method? As a note, the connect() call is down in some library code that I cannot modify, so changing to a non-blocking socket is not an option.

Try to *close()* the socket to interrupt the *connect()*. I'm not sure, but I think it will work at least on Linux. Of course, be careful to synchronize properly such that you only ever *close()* this socket once, or a second*close()* could theoretically close an unrelated file descriptor that was just opened.

**EDIT**: *shutdown*() might be more appropriate because it does not actually close the socket.

Alternatively, you might want to take a look at [pthread\_cancel()](http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_cancel.3.html) and [pthread\_kill()](http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_kill.3.html). However, I don't see a way to use these two without a race condition.

I advise that you abandon the multithreaded-server approach and instead go event-driven, for example by using [epoll](http://linux.die.net/man/4/epoll) for event notification. This way you can avoid all these very basic problems that become very hard with threads, like proper shutdown. You are free to at any time do anything you want, e.g. safely close sockets and never hear from them again.

On the other hand, if in your worker thread you do a non-blocking *connect()* and get notified via epoll\_pwait() (or *ppoll()* or *pselect()*; note the *p*), you may be able to avoid race conditions associated with signals.