CS 194-24 Lab 1: `httpd`

Palmer Dabbelt, Vedant Kumar

February 5, 2014

Contents

For this lab you will be extending a simple HTTP server with a number of performance and feature enhancements. The official HTTP specification is available online as RFC2616. You should treat this much like Intel's ISA manual: it contains the ground truth about HTTP, but in a very high level of detail.

As this is a class in operating systems we won't be too worried about the details of HTTP (much like how you don't need to dwell on the details of Intel's ISA) but will instead be using this HTTP server as a method of driving your exploration into Linux.

High performance web servers tend to push operating systems to the limit that web other applications do: specifically they mix large amounts of IO with large amounts of parallel computation (databases are similar in this regard, but there would be too much code required there before anything interesting happens). One of the major uses of the Linux kernel is to run web servers, and as such Linux has many interesting system calls designed to support them. One of the best ways to learn about Linux's userspace API is, therefore, to write your own web server!

This current assignment is somewhat of a warm-up: it's your first assignment as a group, possibly your first shot at writing an interesting POSIX application, and almost certainly your first time reading an RFC. As such, we won't be requiring you to do too much kernel hacking here – that would just be overkill! You should instead view this assignment as a way to learn about the POSIX programming environment.

This assignment isn't just busy-work, though: For the remainder of this class you will be using this web server as a method of driving a number of improvements to Linux. That means you'll want to take your time getting familiar with the POSIX API calls you use in for this assignment. There will ideally be a performance competition at the end of the class as a means of driving your implementations.

1 Setup

For this assignment you'll need to setup your git repository to talk to your group's repo instead of talking to your personal repository. Luckily you can do this with a minimum of effort. Note that if you've gone and made any changes to the build system then you'll lose them during this process – you should consult with your group members about exactly how you want to merge those changes.

First, you'll want the link to your private repository from the git repository in your VM

```
$ git push personal
$ git remote rm personal
```

You'll then want to reset your git repository to match the public repository. This way you'll start out with a clean slate.

```
$ git fetch origin
$ git checkout release
$ git reset --hard origin/master
$ git checkout master
$ git reset --hard origin/master
```

You'll then want to connect your group's repository to your VM

```
$ git remote add group ssh://git-cs194-24@cs194-24.cs.berkeley.edu/student-group-projects/student-grou
```

Finally, you'll want to populate the branches in your group repo – if you're not the first person to push then it'll simply say "everything up to date", that's perfectly fine.

```
$ git fetch group
$ git push group master
$ git push group release
```

At this point you should have a nice, clean, populated git repo that contains the sources for your group to work on. You may want to tell git to push and pull from your group repository by default (as opposed to origin, which you don't even have write access to). You can do this by running the following commands:

```
$ git branch --set-upstream master group/master
$ git branch --set-upstream release group/release
```

2 Automated Testing

This lab will contain your first foray into the world of automated testing (at least, the first one in this course). We will be providing support code for two automated testing frameworks for this course: Cucumber (for integration testing) and CUnit (for unit testing). You're welcome to use whatever testing frameworks you want as long as they're capable of running in the provided virtual machine, but you'll be stuck writing your own support code.

2.1 CUnit

CUnit is a lightweight unit testing framework that's quite amenable to kernel testing: it's written in C and linkable as a static library. Unfortunately, we forgot to actually install CUnit in the provided virtual machine image so you're going to have to install it yourself. Run the following commands to install the CUnit testing framework

```
$ su
Password: root
# apt-get install libcunit1 libcunit1-dev
```

Example CUnit testing code is provided in the source distribution. Running

```
$ make check
```

will build some test harnesses and proceed to run the CUnit tests. Note that the provided tests are for the memory allocator and therefor do NOT run on your modified kernel (or inside QEMU). This is fine because these unit tests only run in userspace and don't touch any hardware.

2.2 Cucumber, Capybara and Mechanize

The Capybara ruby library contains extensions that allow for easy testing of web applications from within Cucumber. We've provided some example Capybara code that tests the functionality of Google from within Cucumber.

In order to run Capybara, you'll need to install some missing dependencies. You can do this from within your own VM by running the following commands

```
$ su
# apt-get install ruby-dev cucumber bundler
# cd /home/user/cs194-24
# bundle
```

Note that this will take a long time because it's got to compile some source code. At this point running Cucumber will run some tests that speak to Google and parse the results that come back. Look at the provided feature for more information.

3 Distributed Code

We've distributed some code along with this assignment that's designed to get you started. All this code should be available inside your git repository once you've followed merging steps above (it's in the "public/master" branch).

3.1 HTTP Server

The core of the provided code is a simple HTTP server, which is located at `httpd/`. The provided code implements a single-threaded HTTP server that only serves static files. You'll be significantly extending this code for the remainder of the class, so you should be sure that you understand it well.

`httpd` uses a number of tricks that are common to large C codebases that provide some of the niceties of object-oriented design without requiring any language features. You should familiarize yourself with this sort of design because Linux uses many of these principles.

3.2 palloc

`httpd/palloc.*` contains a pool-based memory allocator. This memory allocator in a simplified clone of `talloc`, the memory allocator used in Samba. The `palloc.h` header file should contain enough documentation that you can figure out how the API works. If you're not familiar with pool-based memory allocation, the `talloc` documentation is quite good, you should go ahead and read it

`https://talloc.samba.org/talloc/doc/html/libtalloc__tutorial.html`

`palloc` implements a simplified version of the `talloc` API. Specifically: `palloc` does not support context stealing, contexts with multiple parents, or slab allocation (what `talloc` confusingly calls "memory pools"). There are a few interesting performance improvements that you might want to add to `palloc`: slab allocation and doubly-linked lists.

Part of the reason that we've provided you with this memory allocator is that it's particularly amenable to testing. You should write tests for `palloc`. There is one (intentional) bug in `palloc` – there's probably many more unintentional bugs (though hopefully we shook most of those out last semester)!

3.3 mm_alloc

`httpd/mm_alloc.*` is a clone of the `malloc(3)` interface. The provided code contains stubs which call libc `malloc`. One of your tasks in this lab is to implement your own memory allocator in `mm_alloc.c` from scratch. This is crucial for several reasons, the most pressing of which is that `palloc` relies on `mm_alloc` internally. The other reasons are that it exposes you to POSIX interfaces, forces you to reason about memory, and poses interesting algorithmic challenges.

If your system needs to deal with high memory pressure (which webservers often do), improving your memory allocator can have a dramatic impact on performance. That's why custom allocators are ubiquitous (e.g in CPython, C++ Boost, and Linux), so it's worth understanding them well.

3.4 Web Clock

We've provided a simple CGI application inside the `webclock` directory. This is essentially just a dummy application: it reads the current time, does some useless work, and then prints out the time it read earlier. The idea is that this simulates a real, CPU-intensive webapp without all the complexity of actually doing work.

Lifting the requirement of the webapp doing actual, useful work provides a few advantages: it's simple to provide a scheme for caching of results, and the application becomes trivially parallel.

The trouble with doing dummy work is that it's trivial to cheat by simply not doing the dummy work. `Don't cheat`. The purpose of this application is to allow you to simply test your caching code –

as you won't be modifying the Linux scheduler for this assignment a dummy should be OK. I'll provide a proper application along with the next assignment, once I've figured out exactly how I want you to modify the scheduler.

4 Tasks

The core of this assignment consists of a number of enhancements you'll be making to the provided `httpd` codebase.

4.1 Parallel Client Processing

The provided `httpd` code only handles a single client connection at a time. Clients tend to have a high-latency, low-bandwidth connection to the HTTP server, which means that a single client cannot fully utilize the server. The provided `httpd` will spend the vast majority of its time simply blocking on `write()` calls waiting for data to be sent over the network to the client.

In order to get reasonable performance, all interesting HTTP servers handle more than a single client connection in parallel. Each client request is completely independent, which makes parallelizing a web server fairly simple. There are quite a few approaches to handling client requests in parallel.

Below is a list of increasingly complex (and efficient) methods of handling client requests in parallel. For full credit you'll need to use asynchronous IO, but along the way you'll at least end up with an implementation of thread pools along the way.

4.1.1 Process per client

The simplest way to parallelize a web server is to call `fork()` for each client, which will start a new process for every connection. This has the advantage of pushing all resource management into the kernel (as `httpd` calls `exit()` after every client connection), which makes writing the web server very simple.

The big disadvantage of a `fork()`-based server is that starting a new process for each client request adds a significant amount of overhead. Modern web servers do not `fork()` for each client request because of the added overhead.

Creating a process per client also has the disadvantage that if many client connect, then many processes will be created. This can swamp the operating system's scheduler, causing no very little work to be done at all.

4.1.2 Thread per client

A simple optimization to creating a process per client is to create a thread per client. The advantage is that threads are significantly faster to create than processes are. As threads don't automatically clean up when they terminate, a bit more work is required of the HTTP server to clean up after each client request.

This still has all the disadvantages of creating a process per client, it's just slightly faster.

4.1.3 Thread Pooling

One way to eliminate the overhead of starting a thread for every client request is to instead start a pool of worker threads when the HTTP server starts up. Each of these threads will run for the entire lifetime of the `httpd` process, serving many client requests. This fixes the problems described above that are inherent to creating a schedulable entity for each client connection.

The trouble with using a thread queue is that you must still start up more threads then are host CPUs in order to be able to parallelize IO with processing. This means that extra scheduler overhead is added because the kernel must context switch between many threads, each of which uses a small amount of CPU time and then blocks on an IO call.

4.1.4 Asynchronous IO

One way of lessening the performance of context switching between many threads on a single CPU is to use non-blocking IO. A server that uses non-blocking IO will start many long-lived threads once when it starts up, much like a server that uses thread pooling. The difference is that a non-blocking server will use a single thread per CPU, and will schedule multiple clients onto each thread in userspace by using the non-blocking IO calls and something like `epoll()` to wait for new events.

Essentially a non-blocking web server consists of a userspace implementation of cooperative multi-tasking on top of many kernel threads.

4.2 CGI

The provided `httpd` can only serve static content, which makes it fairly uninteresting. A simple standard for serving dynamic content is known as CGI (Common Gateway Interface). The CGI interface is pretty straight-forward: when a clients requests a file from a web server that has the “execute” bit set, the web server will instead run said file with that process’s standard out piped back to the client. The nice thing about CGI is that it provides a very strict separation between the CGI binary and the web server, thus making the server very extensible.

You’ll want to make sure that CGI works properly, as that’s one of the functions we’ll be using for the remainder of the class.

4.3 Caching and Memory Allocation

One significant improvement that HTTP/1.1 introduces is the ability to cache the results of HTTP requests. For full credit, you must implement HTTP caching of both static and dynamic requests. A description of HTTP/1.1 caching along with references to the RFC can be found online
`http://www.kaizou.org/2009/02/http-caching-explained/`

4.3.1 Browser Caching

Enabling a HTTP/1.1 compliant browser to is fairly simple. HTTP/1.1 adds some cache-specific headers that any compliant browser will add to every request it makes. Your `httpd` must look at those headers and decide what sort of request it should return: it will either say that the client should use its cached response, or that the client should invalidate its cache and use a new response.

`webclock` provides the necessary headers such that clients can cache its requests, but it does not look at client’s headers in order to avoid unnecessary computations. Your solution should not require modification of `webclock`, but you should attempt to avoid doing extra work whenever possible – in other words, your HTTP server should parse the browser’s HTTP caching headers and abort `webclock` when it discovers that the response can be satisfied from the browser’s cache.

For full credit, you must design a scheme that allows browsers to cache static content (ie, HTML files read from the filesystem). Note that this “static” content can change (as the files in the filesystem can change), it’s just called static because it’s not the result of running a program for every request.

4.3.2 `httpd` Caching

In addition to simply passing along the cache headers to the browser, your server itself can contain a cache of frequently requested URLs. This would allow the server to satisfy common requests directly out of its cache rather than requiring it to access disk or run a CGI script.

In order to get full credit, you must support caching both static (ie, on disk) and dynamic content. Note that this is orthogonal from supporting browser caches, but requires quite a bit of the same knowledge.

4.3.3 `palloc` and `mm_alloc`

You are responsible for testing and debugging `palloc`. The stub routines in `mm_alloc.c` should help with this. Since you will be using `palloc` extensively, it's best to do this early on.

Your version of `malloc()` must use `brk()` or `mmap()` (sparingly!) to request memory from the kernel. The main job of `malloc()` is to carve up this memory into reasonably-sized pieces as needed. Your `free()` routine is responsible for deallocating memory and for releasing it back to the kernel as memory pressure decreases. These routines *must be thread safe.* Also note that any heap-allocated data structures you use in your allocator must use memory *allocated by your allocator.* You will be graded on performance, correctness, and the amount of memory you reserve in order to satisfy allocations. The man pages for `malloc`, `mmap`, and `brk` are your best resource. This pthreads tutorial is also useful:

`https://computing.llnl.gov/tutorials/pthreads/`

Regarding memory layout: let `char*` $p = $ `malloc`$(size)$, then $p - 4 \equiv 0 \bmod 4$ in the required layout;

```
---------------------------------------------
| uint32_t size | <size> zero-filled bytes |
---------------------------------------------
(p - 4)         p
```

Consider using a simple data structure to manage memory, like a free interval tree. These let you represent every memory allocation as an extent `(start, length)`. The leaves of the tree correspond to regions of unused memory. If N bytes are requested, it should be possible to scan the interval tree for $(> N)$-sized pieces of memory in $O(\log n)$ time, so long as it's properly balanced.

Deallocating memory can cause fragmentation, which reduces the amount of available contiguous memory and increases lookup times. A simple way to reduce fragmentation is to create separate memory zones for allocations within pre-set size ranges. Each zone would have a separate lookup structure and a pool of backing memory. Slab allocation is based on similar ideas, and can be useful if you're using some sort of linked list for your lookup structure:

`http://en.wikipedia.org/wiki/Slab_allocation`

4.4 Sub-process `clone()` limits

If you look closely at `webclock`, you'll see that it attempts to start up many threads to do its CPU-intensive work. This is intended to simulate an application that reads the number of CPUs present in the machine and attempts to use all of them. This is a good strategy when you're just running a single program, but is bad when you're trying to run instances of the application at once – if the web server and the application both attempt to run n instances on n CPUs then you'll end up with n^2 schedulable entities with only n CPUs, which can be very inefficient on large machines.

Many systems attempt to prevent this thrashing by asking each child application to launch fewer processes. The trouble is that Linux does not provide a way of enforcing these limits, so the child application can still do whatever it wants. One nice feature to add to Linux would be a method of enforcing these limits in the kernel, this way the HTTP server could say "create this child process, but only allow it to make 2 schedulable entities".

Implement this by modifying the `prctl(2)` interface in `linux/include/uapi/prctl.h`. Issuing `prctl(PR_SET_THREADLIMIT, <Limit>, ...)` should set the calling process's thread limit, i.e it says "only allow my subprocesses to create `<Limit>` schedulable entities". Issuing `prctl(PR_GET_THREADLIMIT, &<Limit>, &<Used>, ...)` should retrieve the thread limit and the number of threads currently managed by the process (always ≥ 1). You may need to modify the `clone()` and `exit()` system calls to keep track of how many entities have been created.

5 Schedule and Grading

As this assignment is fairly open-ended, we'll have weekly check points to ensure that you're on the right track. The assignment lasts for three weeks. The first two deadlines are soft: feel free to submit early if you're looking for feedback, if you do I'll give you a day of leeway on incorporating my suggestions; while the final deadline is hard: no late submissions will be accepted. Assignments are all due on Thursday nights.

Additionally, the documentation you provide during your first two weeks is considered to be a work-in-progress. I'm expecting that you'll make some changes to your design during implementation and that you'll need to change your tests to accommodate interesting cases you discover while trying to make things work. That said, you'll be penalized for any major design changes. In other words: your design document needs to be a good-faith effort and reasonably resemble what you build, otherwise you haven't really done a design document. We're trying to get you into the habit of test-driven development, which means your tests should be good enough to describe how your system should work before you start implementation. To enforce this, a similar penalty policy exists for your tests: if your final set of tests don't look anything like your earlier tests, then we'll take off points as you haven't really built proper tests the first time. That said, I wouldn't worry about it – nobody lost points for this last semester.

If you miss a checkpoint deadline then you can submit working code up until the final deadline 75% credit. Note that all the checkpoints require you to submit partial functionality so you'll probably have to end up doing the work anyway in order to do the next checkpoint.

5.1 Checkpoint 1

For the first checkpoint, the following code features will be due

- (5 points) Tests for `malloc`, running against libc's `malloc` (not your own implementation).
- (5 points) A simple version of the `prctl` where calling the `prctl` with 0 disables `clone()`, and calling it with any non-0 value allows unlimited `clone()` calls.
- (5 points) Tests for `palloc` that detail how you'll extend it to be thread-safe, but no implementation. These tests must compile and fail.
- (5 points) A single-threaded, blocking HTTP server that can call CGI scripts.
- (5 points) A design document that details how you will implement malloc, threaded palloc, asynchronous IO, and the full prctl.

5.2 Checkpoint 2

- (5 points) An HTTP server that uses asynchronous IO and is multi-threaded, but doesn't have a cache yet.
- (5 points) A thread-safe version of `palloc` that passes your test suite.
- (5 points) A `malloc` implementation that passes your test cases, as well as works with your `palloc` implementation.
- (10 points) The sub-process clone limits `prctl()` working.
- (5 points) Updates to your design document that cover any changes you've made, as well as how you're going to implement thread-safe caching.

5.3 Final Handin

- (20 points) HTTP caching implemented and working with your previous HTTP work.

6 Design Document

The design document should contain a high level description of how you're planning on implementing each of the features listed above. At this point your plan should contain an assignment of work to each member of your group, along with your best estimate of how long each member's tasks will take.

In addition to your design document, you'll need to submit a test suite that covers all the features you will be implementing for this project. This should consist of Cucumber code. You can submit this by pushing to the release branch and including the hash in your design document.

7 Evaluation

At the end of your third week you'll have to submit your code along with an evaluation. Your evaluation must list any known bugs in your implementation along with some benchmark of what sort of performance impact each feature has. The final code submission deadline is Thursday 2/27 at 9pm, and your evaluation is due by 12pm on Friday.