COMP3301 2021 Assignment 1 - Prometheus Metrics Exporter using Event Driven Programming

- Due: Your lab session in week 5. You must demo in person using your code in svn.
- $Revision: 300 $

1 Prometheus Metrics Collector

This assignment will develop a program to collect metrics about a running OpenBSD system and export them via a Prometheus-compatible HTTP server.

You will be provided with partially-working code, to which you will need to make fixes and improvements. The purpose of this assignment is to demonstrate your skills in reading and understanding existing code as well as fixing and improving it, while exposing you to event-driven programming and many OpenBSD-specific APIs and tools.

This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students, and discuss OpenBSD and its APIs in general terms. You should not actively help (or seek help from) other students with either the correct diagnosis of the coding errors in task 1, or the actual coding of your improvements for the rest of the assignment. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion (outside of the base code given to everyone), formal misconduct proceedings will be initiated against you. If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school web-site: https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism

2 Background

The Prometheus metrics collector is a program which collects metrics. Metrics, in its terminology, are numeric data about a running OpenBSD system. This numeric data can be used to monitor performance or diagnose issues by looking at the history and current value of each metric.

For example, the metrics collector can collect the amount of time the system's CPUs spend in various states (e.g. idle, running kernel code, running user code). This can be used to spot unusual activity on a system or assess how heavily loaded the system is.

In the Prometheus model, there are 4 primary types of metrics:

- Counters: cumulative values which monotonically count the number of times an event has happened, or accumulate the sum of many observations. Counters can only increase or be reset to zero on overflow or restart of the collector. They never decrease (go backwards).
- Gauges: point-in-time measurements, which may increase or decrease over time.
- Histograms: a set of counters which together show an approximate distribution of observations by grouping them into ranges.
- Summaries: a set of counters and gauges which represent digested basic statistics about the distribution of a set of observations (sum, count, min, max etc).

(There is some further information available about these in the Prometheus documentation, as well.)

The collector (the component you will be working on in this assignment) gathers data from the system it is running on. When clients connect to it and request its collected data, it serialises the metrics and sends them to the client.

The interface over which this request is received is the standard HTTP protocol used for accessing websites on the Internet. Clients connect to a chosen TCP port on the machine running the collector and issue an HTTP GET request for the `/metrics` path. They receive a standard HTTP response containing the serialised metrics data. An example of such a request (with the response body truncated):

```
> GET /metrics HTTP/1.1
> Host: foobar:27600
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: obsd-prom-exporter
< Content-Type: text/plain; version=0.0.4; charset=utf-8
< Content-Length: 74983
< Connection: close
<
# HELP io_device_busy_nsec_total IO device busy (service) total time in nanoseconds
# TYPE io_device_busy_nsec_total counter
io_device_busy_nsec_total{device="cd0"} 0
io_device_busy_nsec_total{device="sd0"} 110905274872000
...
```

The serialisation format is readable UTF-8 text with a particular syntax, which looks roughly like this:

```
# HELP io_device_busy_nsec_total IO device busy (service) total time in nanoseconds
# TYPE io_device_busy_nsec_total counter
io_device_busy_nsec_total{device="cd0"} 0
io_device_busy_nsec_total{device="sd0"} 110905274872000
```

In the above example, one metric is present (named `io_device_busy_nsec_total`). It is a counter-type metric with a "help text" description explaining that it represents the sum of all the time that a particular I/O device has spent busy doing work.

You can also see in the example the use of "labels", which allow creating something like "sub-metrics" within one metric. In this case, there are separate counter totals for each different I/O device on the system.

The provided code includes a small framework for declaring and updating metrics and correctly producing this serialisation format.

The client which connects to the collector to read its data is normally Prometheus or OpenTSDB or another compatible system. In this assignment we do not expect you to set up or run either of these – your code will be tested against a basic HTTP client (`curl`) and a simple parser for the Prometheus text metric format.

In the provided code, there are already implemented collectors for a number of key system metrics, such as:

- CPU time spent in various states (user, system, interrupt, idle etc)
- I/O device (e.g. disk) read and write operations, time spent busy
- Network interface transmit and receive operation counts, bytes, errors and drops
- Counts of running processes and threads
- System memory usage (via the UVM subsystem)
- Kernel memory allocator statistics (via `pool(9)`)
- Information from the `pf(4)` firewall

These are exposed as a mix of counters and gauges.

The collector uses an asynchronous I/O reactor loop to process incoming connections and their data. You will need to be familiar with the basics of this concept to fully understand this part of the code. Reviewing the second tutorial session and some of the resources below may be useful.

You should start by reading the remainder of this specification, and then the existing code, to fully understand how it works and what it does. Look up the man pages for functions you find unfamiliar. There are some comments in the code, and these may aid in understanding. The activities will be easiest to complete after first gathering a solid picture of the current code and how it works.

3 Specifications

This assignment consists of 3 major activities:

1. Diagnosing and then fixing an error in the existing program based on a core dump
2. Improving the asynchronous I/O reactor code in the existing program, and changing it to use the `event(3)` library
3. Writing a new metric collector module to add new metrics

The 3 activities build upon one another, so you do not need to submit separate code for all 3. Activity 1 produces some non-code artifacts which you will need to check into your SVN repository alongside the code, as directed. For activities 2 and 3 you will only need to submit one working set of code (by checking it into your SVN repository), which will include your work for both.

3.1 Activity 1: Diagnosing and fixing an error from a core dump

The code you have been provided includes several errors or bugs. In particular, there is a bug which can result in a crash when handling concurrent requests.

On Blackboard->Assessment->Assignment 1, you can find a core dump of the provided code experiencing the crash. The compiled binary which crashed is included in the TAR file as well.

In this first activity, you will need to diagnose the bug which caused the crash in the core dump, write a detailed bug report, and propose a fix (in the form of a patch file).

You do not have to use only the core dump to diagnose the problem: you are free to attempt to reproduce it yourself, and of course you are free to read the code and reason about it to figure out the problem. However, the bug you are required to fix is the one causing the crash in the core file, so you should be sure that it is the issue you fix.

The bug report you produce should follow the outline provided in Appendix A. It should include detailed analysis of the core dump which shows the evidence supporting why your fix is the correct one to make. It should also include details of the steps you took to find the bug and any tools you used. You will not lose marks for using fewer tools or making intuitive deductions, but you do need to provide logical reasoning and evidence for your diagnosis.

Your bug report should be saved in your repository in the same directory as the provided code and your code for the later activities, in a file named `bug-report.txt`.

As well as the bug report, you will need to provide a patch in Unified Diff format (see `diff(1)` and the `-u` option) which fixes the bug. This patch should be able to cleanly apply to the provided code base when run through the command `patch -p1` in the root directory of the code. Your patch does not need to be a minimal patch to be marked correct, but as a hint, we would advise you that there is a very simple possible fix for the issue.

Your patch should be saved in your repository in the same directory as the provided code, in a file named `bug-report.patch`. There is an example Unified Diff patch in Appendix B for your reference.

3.2 Activity 2: Improving the asynchronous I/O reactor

In activity 1, you will learn about a part of the provided code which is prone to error. For activity 2, you will need to improve the code surrounding that error and rework it to use the `event(3)` library.

This will necessarily involve replacing the code which you fixed in activity 1 (which is why you need to produce a separate patch file there).

The current I/O reactor loop in the provided code is very simplistic, and based around using the `poll(2)` API. Hint: it's the loop that contains `poll` in `main.c`. It has a number of sub-optimal behaviours apart from the crash bug you fixed in activity 1.

For this activity, you should replace the current I/O reactor code with code which uses `event(3)`. The new code should be fully asynchronous: all reads and writes should be non-blocking and it should not be possible for one misbehaving client to stop other clients from completing requests.

As well as changing the code to use `event(3)`, you should also add a time-out, which closes client connections which fail to send a complete request to the collector within 30 seconds.

After you have completed this modification, the code should produce the same data and have the same interface as it did before your changes; the only change should be improved concurrent behaviour, and the addition of the 30-second client idle timeout.

You may add additional C source files or header files to the code (and add them to the `Makefile` as needed) to complete this activity, but this is not required.

3.3 Activity 3: Writing a new metric collector module

For the final part of this assignment, you will need to look at the data which is produced by the command `netstat -sp ip` (See netstat man page), and add a new metrics collector module which exports this same data as metrics.

Locating the code and/or APIs responsible for producing this output in `netstat` is part of this activity, as well as figuring out how to integrate that into the existing collector framework in the provided code. Please note that collaborating with other students on the specifics of this particular `netstat` command, how it works, or on how to write a metric collector module to fit into the provided framework is considered collusion.

Your code for collecting these metrics must not call `fork(2)` or any other function which creates a new process (i.e. you may not "shell out" to run `netstat` itself).

The existing metrics collector modules in the code (in files named `collect_*.c`) should also help to provide examples of what you need to do.

You will need to choose appropriate names and metric types for each datum that your new module collects. Metric names must follow the Prometheus metric naming guidelines, and style marks are reserved for this.

You may export fewer metrics than there are data in the output of `netstat -sp ip` by way of using labels, if you think this is appropriate. You should research carefully what each datum represents and think about how it can be useful in understanding the state and performance of the system.

You may add additional C source files or header files to the code (and add them to the `Makefile` as needed) to complete this activity, but this is not required. You will also need to modify the existing `metrics.c` file.

3.4 Writing Style

In activity 1, you will need to produce a written bug report and explanation of your proposed fix. This is to be written in English, with appropriate grammar, punctuation and style for general business writing (think roughly: "would I write this to my supervisor at work?"). You will not lose marks for trivial spelling or grammatical errors which do not obscure the meaning of what you have written, but we recommend taking some care.

3.5 Code Style

Your code is to be written according to OpenBSD's style guide, as per the `style(9)` man page.

3.6 Compilation

Your code for activities 2 and 3 is to be built on an `amd64` OpenBSD 6.9 system. It must compile as a result of running `make(1)` in the root directory of your submitted assignment.

The existing `Makefile` in the provided code is functional as-is, but will need modification as part of your work for this assignment.

Compilation must produce a binary called `prom-exporter`, in the same manner as with the provided `Makefile`.

Note that the existing Makefile ensures the `-Wall` flag is passed to the compiler, as well as a few other warning and error-related flags. We will be compiling your code with these flags forced on, so you should not remove or modify them.

3.7 Required Dependencies

`prom-exporter` must only use the following libraries and the APIs they provide to implement the above functionality.

3.7.1 `libc`

`libc` refers to the ISO or POSIX standard C library provided by UNIX and UNIX like operating systems. `libc` contains the APIs required by `prom-exporter` for basic tasks such as creating network sockets and connections, opening and reading files, and collecting system information.

3.7.2 `libevent`

According to http://libevent.org/, `libevent`:

> ... provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also support callbacks due to signals or regular timeouts.

OpenBSD ships with `libevent` (see `event(3)`). It is this version of `libevent` in the base system that you are required to write the code in activity 2 against.

3.7.3 `http-parser`

The provided code is bundled with a copy of the code for Joyent's `http-parser` (within the directory `http-parser/` in the provided code bundle). This library is used to parse incoming HTTP requests. You should continue to use this code for that purpose, and you must not modify it.

3.8 Restrictions

In Activities 2 and 3:

- The `prom-exporter` must operate as a single process/thread. It may fork once only at startup to daemonise, and must not create any threads.
- The `prom-exporter` may not call `system(3)` or any of the `execv(3)` or `execve(2)` family of functions, nor may it use any other means of executing another process to gather data.
- The final `prom-exporter` code (after activity 2 is completed) must use the `event(3)` library for all I/O multiplexing. It must not call `poll(2)`, `select(2)` or `kqueue(2)` directly, and these functions should not appear in your code.

3.9 Recommendations

The focus of this assignment is reading and modifying existing code, as well as understanding event driven programming. We don't expect you to "reinvent the wheel" if there is common functionality available in OpenBSD already which you can leverage.

It is strongly recommended that the following APIs are used as part of your program:

- `event(3)`
- `sysctl(2)`
- `fcntl(2)` and/or the `SOCK_NONBLOCK` option to `socket(2)`

4 Submission

Submission must be made electronically by committing to your Subversion repository on `source.eait.uq.edu.au`. In order to mark your assignment the markers will check out `a1` from your repository. Code checked in to any other part of your repository will not be marked.

As per the `source.eait.uq.edu.au` usage guidelines, you should only commit source code and Makefiles.

Running the command `ls -laR` in the root of your SVN repository should produce output which looks roughly like this (yours will not be exactly the same as this, of course – this doesn't include any additional files you may add in activity 2, or any files from the pracs):

```
alex@obsd svn$ ls -laR
.:
total 12
drwxr-xr-x  3 alex  alex  512 Aug  8 20:42 .
drwxr-xr-x  4 alex  alex  512 Aug  8 20:42 ..
drwxr-xr-x  3 alex  alex  512 Aug  8 20:47 a1

./a1:
total 208
drwxr-xr-x  3 alex  alex    512 Aug  8 20:47 .
drwxr-xr-x  3 alex  alex    512 Aug  8 20:42 ..
-rw-r--r--  1 alex  alex   1273 Aug  6 15:48 LICENSE
-rw-r--r--  1 alex  alex   1766 Aug  6 17:46 Makefile
-rw-r--r--  1 alex  alex    ??? Aug  8 20:47 bug-report.patch
-rw-r--r--  1 alex  alex    ??? Aug  8 20:46 bug-report.txt
-rw-r--r--  1 alex  alex    ??? Aug  8 20:46 collect_???.c
-rw-r--r--  1 alex  alex   3076 Aug  8 16:30 collect_cpu.c
-rw-r--r--  1 alex  alex   5105 Aug  8 16:31 collect_disk.c
-rw-r--r--  1 alex  alex   6829 Aug  8 16:30 collect_if.c
-rw-r--r--  1 alex  alex   6443 Aug  8 16:31 collect_pf.c
-rw-r--r--  1 alex  alex   6581 Aug  8 16:30 collect_pools.c
-rw-r--r--  1 alex  alex   4688 Aug  8 16:31 collect_procs.c
-rw-r--r--  1 alex  alex   3516 Aug  8 16:30 collect_uvm.c
drwxr-xr-x  2 alex  alex    512 Aug  6 17:29 http-parser
-rw-r--r--  1 alex  alex   3699 Aug  8 16:31 log.c
-rw-r--r--  1 alex  alex   1721 Aug  6 15:36 log.h
-rw-r--r--  1 alex  alex  14404 Aug  8 16:37 main.c
-rw-r--r--  1 alex  alex  14426 Aug  8 20:46 metrics.c
-rw-r--r--  1 alex  alex   5248 Aug  8 16:23 metrics.h

./a1/http-parser:
total 200
```

```
drwxr-xr-x  2 alex  alex    512 Aug  6 17:29 .
drwxr-xr-x  3 alex  alex    512 Aug  8 20:47 ..
-rw-r--r--  1 alex  alex   1077 Aug  6 15:46 LICENSE-MIT
-rw-r--r--  1 alex  alex  75645 Aug  6 17:29 http_parser.c
-rw-r--r--  1 alex  alex  19628 Aug  6 15:46 http_parser.h
```

4.1 Demonstration sessions

In the week following the submission deadline, you will be asked to demonstate your assignment as part of your regular practical session.

Demonstrations will run on an OpenBSD machine provided by course staff, which you and your marking tutor will operate jointly for the demonstration. Your Subversion repository will be checked out from scratch on that machine, and the material that results is what you will be marked on.

At this session we will be running the example commands listed below, as well as examining your code and running additional tests and scripts. We will also be reading your bug reports for activity 1. You may have a chance to clarify any points of confusion or misinterpretation in your bug report, but changes to your patch or code will not be accepted.

5 Testing

For activities 2 and 3, you can test your program using the `curl` command. This can be installed from ports (`pkg_add curl`). Some examples:

5.1 `curl` example 1

A GET on the root URL (`/`) should return a 404 error:

```
$ curl -v http://localhost:27600
*   Trying 127.0.0.1:27600...
* Connected to localhost (127.0.0.1) port 27600 (#0)
> GET / HTTP/1.1
> Host: localhost:27600
> User-Agent: curl/7.78.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< Server: obsd-prom-exporter
< Connection: close
<
* Closing connection 0
```

5.2 `curl` example 2

A GET on the metrics URL (`/metrics`) should return collected metrics:

```
$ curl -v http://localhost:27600/metrics
*   Trying 127.0.0.1:27600...
* Connected to localhost (127.0.0.1) port 27600 (#0)
> GET /metrics HTTP/1.1
> Host: localhost:27600
> User-Agent: curl/7.78.0
> Accept: */*
>
```

```
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Server: obsd-prom-exporter
< Content-Type: text/plain; version=0.0.4; charset=utf-8
< Content-Length: 73113
< Connection: close
<
# HELP io_device_busy_nsec_total IO device busy (service) total time in nanoseconds
# TYPE io_device_busy_nsec_total counter
io_device_busy_nsec_total{device="cd0"} 0
io_device_busy_nsec_total{device="sd0"} 7759505000
# HELP io_device_written_bytes_total Count of bytes written to an I/O device
# TYPE io_device_written_bytes_total counter
io_device_written_bytes_total{device="cd0"} 0
io_device_written_bytes_total{device="sd0"} 345798656
# HELP io_device_read_bytes_total Count of bytes read from an I/O device
# TYPE io_device_read_bytes_total counter
io_device_read_bytes_total{device="cd0"}  0
io_device_read_bytes_total{device="sd0"}  477447168
# HELP io_device_write_ops_total Count of write operations completed by an I/O device
# TYPE io_device_write_ops_total counter
io_device_write_ops_total{device="cd0"} 0
io_device_write_ops_total{device="sd0"} 23403
...
```

5.3 `curl` example 3

A POST to the metrics URL (`/metrics`) should return a 404 error:

```
$ curl -v -XPOST http://localhost:27600/metrics
*   Trying 127.0.0.1:27600...
* Connected to localhost (127.0.0.1) port 27600 (#0)
> POST /metrics HTTP/1.1
> Host: localhost:27600
> User-Agent: curl/7.78.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< Server: obsd-prom-exporter
< Connection: close
<
* Closing connection 0
```

5.4 Testing concurrency

You can test with multiple `curl` commands in parallel to verify concurrent behaviour (run them in separate shells).

You can also use tools such as `nc` to make a raw TCP connection to the correct port and type out a basic HTTP 1.0 request yourself. This is often simpler for triggering edge cases in concurrent behaviour (since you can control the timing of connection and request and do things like close the connection early):

```
$ nc -v localhost 27600
Connection to localhost (127.0.0.1) 27600 port [tcp/*] succeeded!
GET / HTTP/1.0
```

```
HTTP/1.0 404 Not Found
Server: obsd-prom-exporter
Connection: close

$
```

You are also free to use any other tool you wish to attempt to verify the correct operation of your server code after the modifications in activity 2.

5.5 Testing for Activity 3

In activity 3 you will add a new metrics module. The easiest way to test its operation is to use `curl` to fetch the metrics endpoint and then look through the output for the new metrics you have added.

You can check their values against the actual output of the `netstat` command listed in the activity.

6 Appendix A: Bug Report Template

The following is the format in which you should provide your bug report for Activity 1:

```
Summary
-------
A short one-sentence summary of the issue.

Analysis
--------

Paragraphs of text. This should be a narrative which describes how the bug
was first found, what analysis was performed to determine the cause of the bug,
showing the evidence along the way.

Include evidence as indented blocks of command output or code (see example
below).

Proposed fix
------------

Summarise the changes that need to be made to the code and why they will fix
the problem.
```

6.1 Example

```
Summary
-------
prom-exporter leaks memory and sockets after successful requests

Analysis
--------

A prom-exporter instance running in production was observed using a lot of
memory in "top":

      PID USERNAME PRI NICE  SIZE   RES STATE     WAIT      TIME    CPU COMMAND
    19133 _promexp   2    0   87M   35M sleep/1   poll     29.6H  0.00% obsd-prom-export
```

A core dump was taken of the process to inspect its state, by sending it
the SIGABRT signal.

In `gdb`, we can see the current `malloc` state of the process:

```
(gdb) print malloc_readonly.mopts
$1 = {malloc_pool = 0x6a19ad1c000, malloc_mutexes = 8, malloc_mt = 0,
  malloc_freecheck = 0, malloc_freeunmap = 0, def_malloc_junk = 1,
  malloc_realloc = 0, malloc_xmalloc = 0, chunk_canaries = 0,
  internal_funcs = 1, def_maxcache = 64, malloc_guard = 0,
  malloc_canary = 1181946964}
(gdb) print $1.malloc_pool[1]
$2 = (struct dir_info *) 0x6a17539f740
(gdb) print *$2
$3 = {canary1 = 860545812, active = 0, r = 0x6a15060a000,
  regions_total = 256, regions_free = 230, rbytesused = 26,
  func = 0x6a19ac4872e "free", malloc_junk = 1, mmap_flag = 0, mutex = 1,
  chunk_info_list = 0x6a17539f780, chunk_dir = 0x6a17539f7e0,
  delayed_chunks = 0x6a17539f960,
  rbytes = 0x6a17539f9e0 "..", cache = 0x6a17539fa00, canary2 = 3434421483}
(gdb) ...
   <cut for brevity>
```

It looks like all the memory usage is made up of chunks of size 64-128 bytes.
The current size of the "struct req" is 88 bytes, which would fit in this
bucket.

Looking at the reqs list:

```
(gdb) set $req = reqs
(gdb) while ($req != 0)
     >print *$req
     >set $req = $req->next
     >end
$10 = {id = 1, done = 1, next = 0x6a0d0e45280, prev = 0x0, registry = 0x6a0d0e68ca0,
  raddr = {sin_len = 16 '\020', sin_family = 2 '\002', sin_port = 32426,
    sin_addr = {s_addr = 29468864}, sin_zero = 0x6a0d0e452a8 ""}, pfdnum = 1,
  sock = 4, wf = 0x6a19ad12740, parser = 0x6a0d0e2ef20, resp = RESP_NOT_FOUND}
$11 = {id = 2, done = 1, next = 0x6a0d0e45380, prev = 0x6a0d0e45180, registry = 0x6a0d0e68ca0,
  raddr = {sin_len = 16 '\020', sin_family = 2 '\002', sin_port = 32512,
    sin_addr = {s_addr = 29468864}, sin_zero = 0x6a0d0e452a8 ""}, pfdnum = 2,
  sock = 5, wf = 0x6a19ad12740, parser = 0x6a0d0e2ef20, resp = RESP_NOT_FOUND}
...
$1011 = { ... }
^C
```

The operation had to be cancelled due to taking too long to print them all.
Modifying the loop to increment a counter instead:

```
(gdb) set $req = reqs
(gdb) set $count = 0
(gdb) while ($req != 0)
     >set $count = $count + 1
```

```
         >set $req = $req->next
         >end
    (gdb) print $count
    $1012 = 351328
```

This represents roughly 30MB of struct reqs, which would account for most of the process' unusually high RSS.

All of the struct reqs on the list appear to have the "done" flag set to 1, and a unique file descriptor.

In main.c around line 410, we call the http-parser on the request (which will then call our callbacks such as on_url). After it returns, we check for error conditions, but we do not currently check for the "done" flag being set. This allows finished requests to accumulate on the list for as long as their underlying socket remains open.

Proposed fix

An if() block should be added to this part of main.c to handle checking the req->done flag:

```
        if (req->done) {
            free_req(req);
            continue;
        }
```

7 Appendix B: Example Unified Diff

```
--- a/main.c
+++ b/main.c
@@ -216,7 +216,7 @@
        settings.on_headers_complete = on_headers_complete;
        settings.on_message_complete = on_message_complete;
        settings.on_url = on_url;
-       settings.on_body = on_body;
+       /* The on_body callback is optional. */
        settings.on_header_field = on_header_field;
        settings.on_header_value = on_header_value;

@@ -481,13 +481,6 @@
 }

 static int
-on_body(http_parser *parser, const char *data, size_t len)
-{
-       //struct req *req = parser->data;
-       return (0);
-}
-
-static int
 on_header_field(http_parser *parser, const char *hdrname, size_t hlen)
 {
```

```
//struct req *req = parser->data;
```