CS631 - Advanced Programming in the UNIX Environment
–
HTTP, Dæmon processes, System Logging, Shared Libraries

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
http://www.cs.stevens.edu/~jschauma/631/

HTTP

`http://www.cs.stevens.edu/~jschauma/631/f13-final-project.html`

Hypertext Transfer Protocol

RFC2616

HTTP

HTTP is a request/response protocol.

The Hypertext Transfer Protocol

HTTP is a request/response protocol:

1. client sends a request to the server
2. server responds

The Hypertext Transfer Protocol

HTTP is a request/response protocol:

1. client sends a request to the server
   - request method
   - URI
   - protocol version
   - request modifiers
   - client information
2. server responds

HTTP: A client request

```
$ telnet www.google.com 80
Trying 2607:f8b0:400c:c02::93...
Connected to www.google.com.
Escape character is '^]'.
GET / HTTP/1.0
```

The Hypertext Transfer Protocol

HTTP is a request/response protocol:

1. client sends a request to the server
   - request method
   - URI
   - protocol version
   - request modifiers
   - client information
2. server responds
   - status line (including success or error code)
   - server information
   - entity metainformation
   - content

HTTP: a server response

```
HTTP/1.0 200 OK
Date: Mon, 22 Oct 2012 03:08:18 GMT
Content-Type: text/html; charset=ISO-8859-1
Server: gws

<!doctype html><html itemscope="itemscope"
itemtype="http://schema.org/WebPage"><head><meta content="Search the
world's information, including webpages, images, videos and more. Google
has many special features to help you find exactly what you're looking
for." name="description"><meta content="noodp" name="robots"><meta
itemprop="image"
content="/images/google_favicon_128.png"><title>Google</title><script>
window.google={kEI:"oriEUNmMGMX50gH6kYGwBw",getEI:function(a){var
b;while(a&&!(a.getAttribute&&(b=a.getAttribute("eid"))))a=a.parentNode;return
b||google.kEI},https:function(){return
window.location.protocol=="https:"},kEXPI:"25657,30316,39523,39977,40362
```

The Hypertext Transfer Protocol

Server status codes:

- **1xx** – Informational; Request received, continuing process
- **2xx** – Success; The action was successfully received, understood, and accepted
- **3xx** – Redirection; Further action must be taken in order to complete the request
- **4xx** – Client Error; The request contains bad syntax or cannot be fulfilled
- **5xx** – Server Error; The server failed to fulfill an apparently valid request

HTTP: A client request

```
$ telnet www.cs.stevens.edu 80
Trying 155.246.89.84...
Connected to tarantula.srcit.stevens-tech.edu.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 04 Apr 2011 02:16:14 GMT
Server: Apache/2.2.9 (Debian) DAV/2 SVN/1.5.1 PHP/5.2.6-1+lenny9 with Suhosin-Patch m
Last-Modified: Wed, 17 Nov 2010 19:25:54 GMT
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>SRCIT wiki page</title>
<meta http-equiv="REFRESH"
content="0;url=http://www.srcit.stevens.edu/wiki"></HEAD>
<BODY>
</BODY>
</HTML>
```

HTTP - more than just text

HTTP is a *Transfer Protocol* – serving *data*, not any specific text format.

- `Accept-Encoding` client header can specify different formats such as *gzip*, *Shared Dictionary Compression over HTTP (SDCH)* etc.
- corresponding server headers: `Content-Type` and `Content-Encoding`

HTTP - more than just static data

HTTP is a *Transfer Protocol* – what is transferred need not be static; resources may generate different data to return based on many variables.

- CGI – resource is *executed*, needs to generate appropriate response headers
- server-side scripting (ASP, PHP, Perl, ...)
- client-side scripting (JavaScript/ECMAScript/JScript,...)
- applications based on HTTP, using:
  - AJAX
  - RESTful services
  - JSON, XML, YAML to represent state and abstract information

Writing a *simple* HTTP server

- parse command-line options, initialize world, ...
- open socket
- run as a dæmon, loop forever
  - accept connection
  - fork child to handle request
- upon `SIGHUP` re-read configuration, restart

Writing a *simple* HTTP server

Processing requests consists of:

- reading request from socket
- parsing request
  - valid syntax?
  - type of request (GET, HEAD, POST)?
  - determine pathname
    - $\sim$ translation
    - translate relative into absolute pathname
- generate server status response
- handle request

Writing a *simple* HTTP server

Processing requests consists of:

- handling regular file request
  - `stat(2)` file
  - `open(2)` file
  - `read(2)` file
  - `write(2)` to socket
  - `close(2)` file
  - terminate connection
  - exit child handler
- handling CGI execution
  - setup environment
  - setup filedescriptors (stdin/stdout)
  - fork-exec executable

Client-Server Model

- two categories of of servers
  1. iterative
  2. concurrent

Client-Server Model

- two categories of of servers
  1. iterative
     1.1. wait for client request to arrive
     1.2. process the client request
     1.3. send the response back to the client
     1.4. go back to 1.1
  2. concurrent

Client-Server Model

- two categories of of servers
  1. iterative
     1.1. wait for client request to arrive
     1.2. process the client request
     1.3. send the response back to the client
     1.4. go back to 1.1
  2. concurrent
     2.1. wait for client request to arrive
     2.2. start a new server to handle this client's request
     2.3. go back to 2.1

Dæmon processes

So... what's a dæmon process anyway?

Dæmon characteristics

Commonly, dæmon processes are created to offer a specific service.

Dæmon processes usually

- live for a long time
- are started at boot time
- terminate only during shutdown
- have no controlling terminal

Dæmon characteristics

The previously listed characteristics have certain implications:

- do one thing, and one thing only
- no (or only limited) user-interaction possible
- consider current working directory
- how to create (debugging) output

Writing a dæmon

- fork off the parent process
- change file mode mask (umask)
- create a unique Session ID (SID)
- change the current working directory to a safe place
- close (or redirect) standard file descriptors
- open any logs for writing
- enter actual dæmon code

Writing a dæmon

```
int
daemon(int nochdir, int noclose)
{
        int fd;

        switch (fork()) {
        case -1:
                return (-1);
        case 0:
                break;
        default:
                _exit(0);
        }

        if (setsid() == -1)
                return (-1);

        if (!nochdir)
                (void)chdir("/");

        if (!noclose && (fd = open(_PATH_DEVNULL, O_RDWR, 0)) != -1) {
                (void)dup2(fd, STDIN_FILENO);
                (void)dup2(fd, STDOUT_FILENO);
                (void)dup2(fd, STDERR_FILENO);
                if (fd > STDERR_FILENO)
                        (void)close(fd);
        }
        return (0);
}
```

Dæmon conventions

- prevent against multiple instances via a *lockfile*
- allow for easy determination of PID via a *pidfile*
- configuration file convention `/etc/`*`name`*`.conf`
- include a system initialization script (for `/etc/rc.d/` or `/etc/init.d/`)
- re-read configuration file upon `SIGHUP`

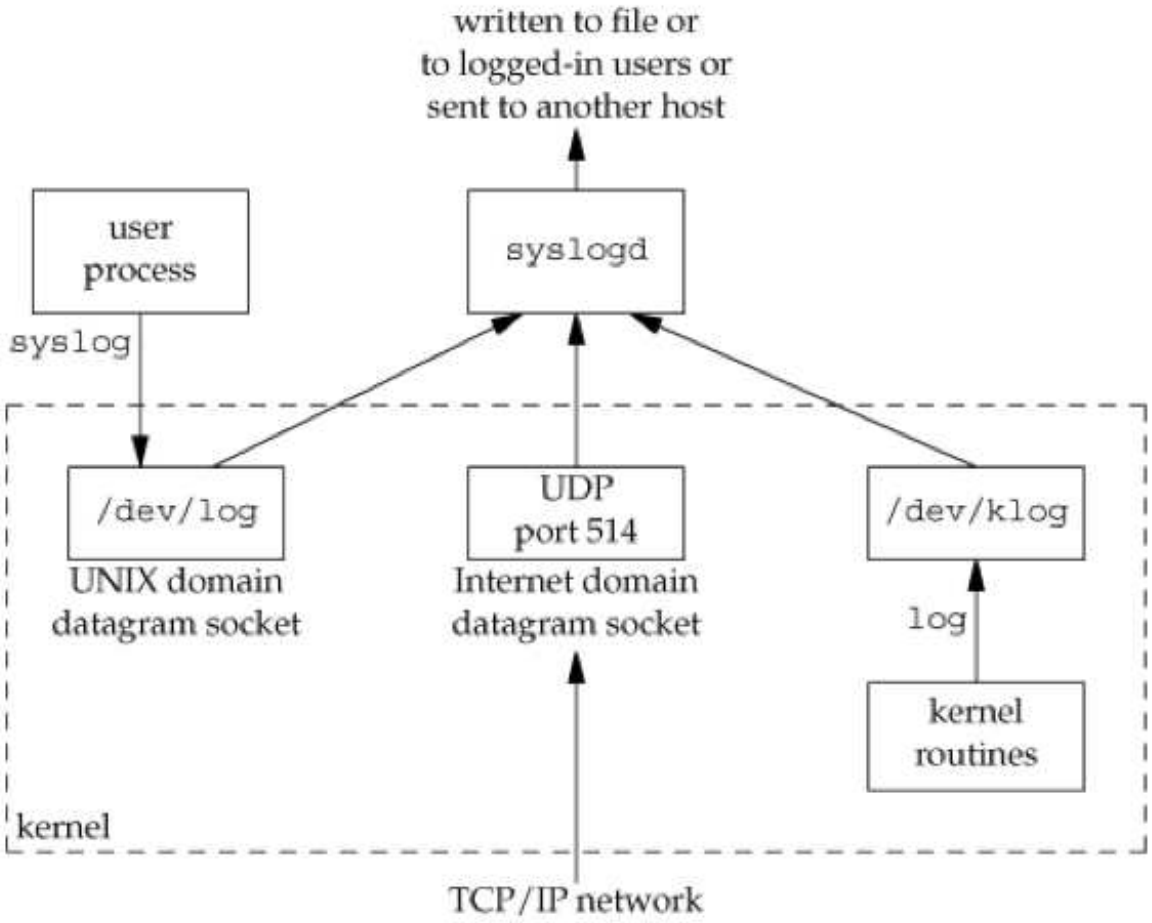
Logging

A central logging facility

There are three ways to generate log messages:

- via the kernel routine `log(9)`
- via the userland routine `syslog(3)`
- via UDP messages to port 514

A central logging facility



syslog(3)

```
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);
void syslog(int priority, const char *message, ...);
```

`openlog(3)` allows us to set specific options when logging:

- prepend *ident* to each message
- specify logging options (`LOG_CONS | LOG_NDELAY | LOG_PERRO | LOG_PID`)
- specify a *facility* (such as `LOG_DAEMON`, `LOG_MAIL` etc.)

`syslog(3)` writes a message to the system message logger, tagged with *priority*.
A *priority* is a combination of a *facility* (as above) and a *level* (such as `LOG_DEBUG`, `LOG_WARNING` or `LOG_EMERG`).

Shared Libraries

What is a shared library, anyway?

Shared Libraries

What is a shared library, anyway?

- contains a set of callable C functions (ie, implementation of function prototypes defined in `.h` header files)
- code is position-independent (ie, code can be executed anywhere in memory)
- shared libraries can be loaded/unloaded at execution time or at will
- libraries may be *static* or *dynamic*

Shared Libraries

What is a shared library, anyway?

- contains a set of callable C functions (ie, implementation of function prototypes defined in `.h` header files)
- code is position-independent (ie, code can be executed anywhere in memory)
- shared libraries can be loaded/unloaded at execution time or at will
- libraries may be *static* or *dynamic*

```
$ man 3 fprintf
$ grep " fprintf" /usr/include/stdio.h
```

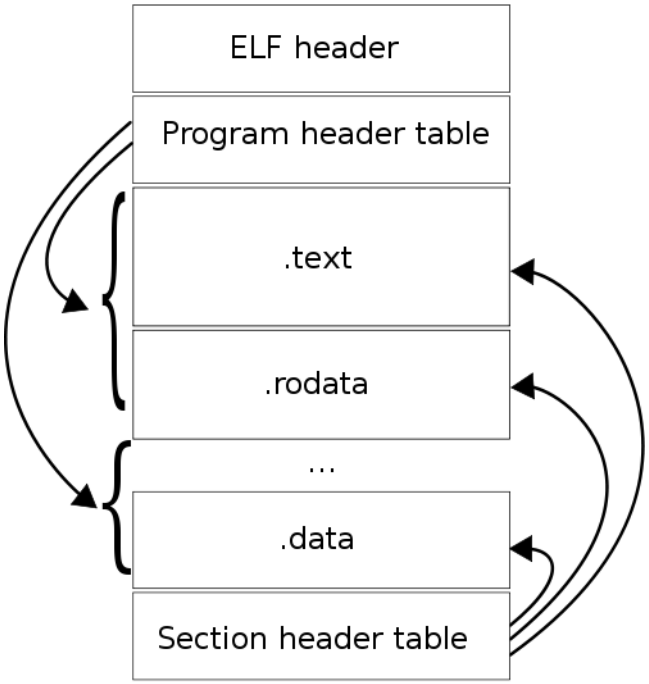
Shared Libraries

How do shared libraries work?

- contents of *static* libraries are pulled into the executable at link time
- contents of *dynamic* libraries are used to resolve symbols at **link time**, but loaded at **execution time** by the *dynamic linker*
- contents of *dynamic* libraries may be loaded at **any time** via explicit calls to the dynamic linking loader interface functions

Executable and Linkable Format

ELF is a file format for executables, object code, shared libraries etc.



More details: `http://www.cs.stevens.edu/~jschauma/631/elf.html`
`http://www.thegeekstuff.com/2012/07/elf-object-file-format/`

Understanding object files

```
$ cc -Wall -c ldtest1.c ldtest2.c main.c
$ readelf -h ldtest1.o
[...]
$ cc *.o
$ readelf -h a.out
[...]
$ ldd a.out
[...]
$ readelf -h /lib/libc.so.6
[...]
$ readelf -s a.out | more
[...]
$ objdump -d -j .text a.out | more
[...]
$ nm -D a.out | more
[...]
$
```

Statically Linked Shared Libraries

Static libraries:

- created by `ar(1)`
- usually end in `.a`
- contain a symbol table within the archive (see `ranlib(1)`)

Statically Linked Shared Libraries

```
$ cc -Wall -c ldtest1.c
$ cc -Wall -c ldtest2.c
$ cc -Wall main.c
[...]
$ cc -Wall main.c ldtest1.o ldtest2.o
$
```

Statically Linked Shared Libraries

```
$ cc -Wall -c ldtest1.c ldtest2.c
$ ar -vq libldtest.a ldtest1.o ldtest2.o
$ ar -t libldtest.a
$ cc -Wall main.c libldtest.a

$ cc -Wall -c main.c
$ cc main.o -L. -lldtest -o a.out.dyn
$ cc -static main.o -L. -lldtest -o a.out.static
$ ls -l a.out.*
$ ldd a.out.*
$ nm a.out.dyn | wc -l
$ nm a.out.static | wc -l
```

Dynamically Linked Shared Libraries

Explicit loading of shared libraries:

- `dlopen(3)` creates a handle for the given library
- `dlsym(3)` returns the address of the given symbol
-

```
$ cc -Wall setget.c
$ cc -Wall -rdynamic dlopenex.c -ldl
$ ./a.out
```

Dynamically Linked Shared Libraries

Dynamic libraries:

- created by the compiler/linker (ie multiple steps)
- usually end in `.so`
- frequently have multiple levels of symlinks providing backwards compatibility / ABI definitions

Dynamically Linked Shared Libraries

```
$ rm *.o libldtest*
$ cc -Wall -c -fPIC ldtest1.c
$ cc -Wall -c -fPIC ldtest2.c
$ mkdir lib
$ cc -shared -Wl,-soname,libldtest.so.1 -o lib/libldtest.so.1.0 ldtest1.o ldtest2.o
$ ln -s libldtest.so.1.0 lib/libldtest.so.1
$ ln -s libldtest.so.1.0 lib/libldtest.so
$ cc -static -Wall main.o -L./lib -lldtest
[...]
$ cc -Wall main.o -L./lib -lldtest
[...]
$ ./a.out
[...]
$ ldd a.out
[...]
```

Dynamically Linked Shared Libraries

Wait, what?

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./lib
$ ldd a.out
[...]
$ ./a.out
[...]
$ mkdir lib2
$ cc -Wall -c -fPIC ldtest1.2.c
$ cc -shared -Wl,-soname,libldtest.so.1 -o lib2/libldtest.so.1.0 ldtest1.2.o ldtest2.
$ ln -s libldtest.so.1.0 lib2/libldtest.so.1
$ ln -s libldtest.so.1.0 lib2/libldtest.so
$ export LD_LIBRARY_PATH=./lib2:$LD_LIBRARY_PATH
$ ldd a.out  # note: no recompiling!
[...]
$ ./a.out
[...]
```

Dynamically Linked Shared Libraries

Avoiding `LD_LIBRARY_PATH`:

```
$ cc -Wall main.o -L./lib -lldtest -Wl,-rpath,./lib
$ echo $LD_LIBRARY_PATH
[...]
$ ldd a.out
[...]
$ ./a.out
[...]
$ unset LD_LIBRARY_PATH
$ ldd a.out
[...]
$ ./a.out
[...]
$
```

Dynamically Linked Shared Libraries

But:

```
$ cc -Wall -fPIC -c evil.c
$ cc -shared -Wl,-soname,libldtest.so.1 -o lib3/libldtest.so.1.0 \
        ldtest1.o ldtest2.o evil.o
$ export LD_PRELOAD=./lib3/libldtest.so.1.0
$ ldd a.out
[...]
$ ./a.out
[...]
$
```

Dynamically Linked Shared Libraries

```
$ export LD_DEBUG=help # glibc>=2.1 only
$ ./a.out
[...]
$ LD_DEBUG=all ./a.out
[...]
```

Homework

Same as last week:

`http://www.cs.stevens.edu/~jschauma/631/f13-hw3.html`

`http://www.cs.stevens.edu/~jschauma/631/f13-final-project.html`