CS631 - Advanced Programming in the UNIX Environment
–
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/

Create an ssh key pair.

```
$ ssh-keygen -f ~/.ssh/cs631apue -q
$ mv ~/.ssh/cs631apue.pub ~jschauma/tmp/cs631/${USER}.pub
$ chmod a+r ~jschauma/tmp/cs631/${USER}.pub
```

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`

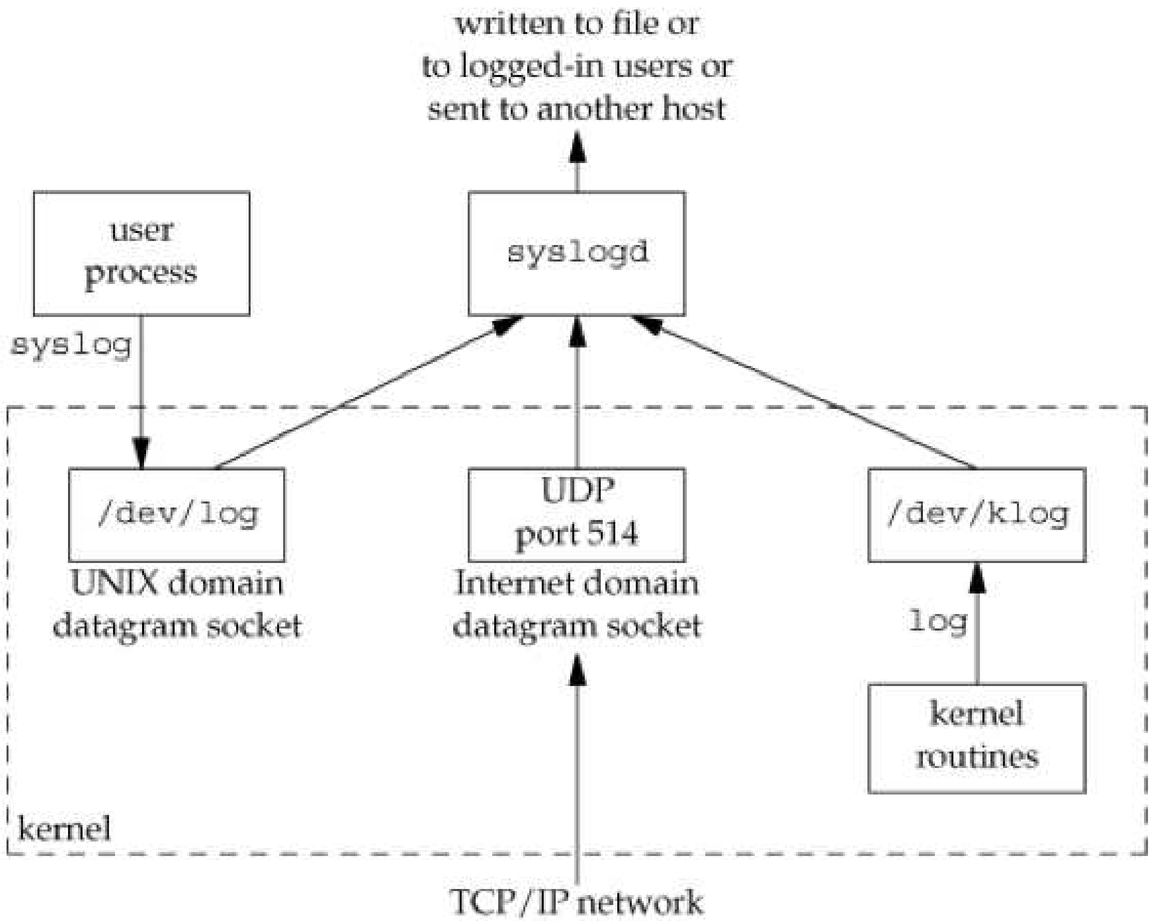
Logging

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`).

Let’s write a shared library, `libgreet`.

```
NAME
     greet, hello, getgreeting, setgreeting  hello world library

LIBRARY
     Greetings Library (libgreet, lgreet)

SYNOPSIS
     #include <greeting.h>

     void greet(void);

     void hello(const char * friend, const char * greeting);

     char * getgreeting(void);

     int setgreeting(const char * greeting);

DESCRIPTION
     The greet, family of functions allows you to easily greet your users.

     The greet() function simply prints the current greeting, followed by a
     newline character (\n) to stdout.

     The hello() function prints greeting prefixed with friend, a colon (:)
     and a space to stdout.

     The getgreeting() function returns the current greeting.

     The setgreeting() function sets the default greeting to use when calling
     greet().
```

Let's write a shared library, `libgreet`.

```
#include <greet.h>
#include <stdio.h>

int main(void) {
        greet();
        if (setgreeting("Howdy!") != 0) {
                fprintf(stderr, "Unable to set greeting!\n");
        }
        greet();
        hello("world", getgreeting());
        return 0;
}
$ cc -Wall hello.c -lgreet
$ ./a.out
Hello!
Howdy!
world: Howdy!
```

Let’s write a shared library, `libgreet`.

`https://www.cs.stevens.edu/~jschauma/631/f15-libgreet.html`

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
```

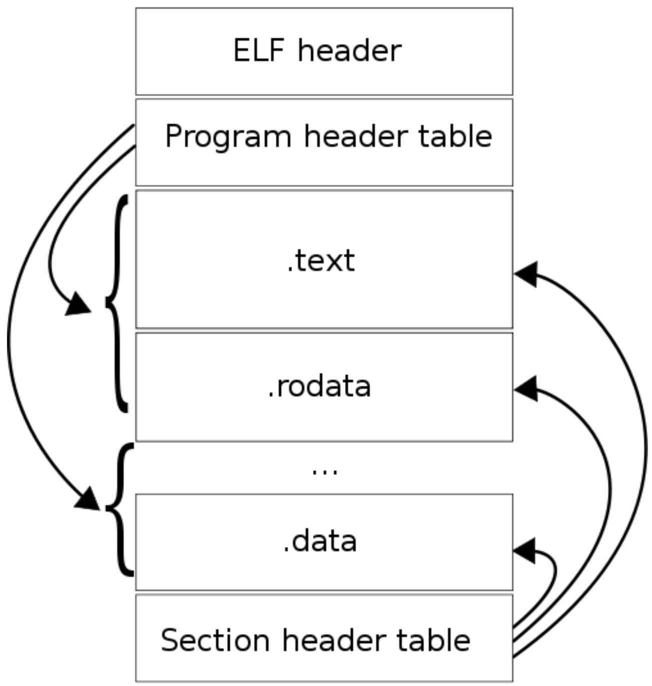
Shared Libraries

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

Turn your `greet.c` code into a shared library, `libgreet.so`, such that you can:

```
cc -Wall hello.c -I./libgreet \
        -L./libgreet -Wl,-rpath,./libgreet -lgreet
```

And of course: work on your final project.