CS631 - Advanced Programming in the UNIX Environment
–
Dæmon Processes, Shared Libraries

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

Dæmon processes

So... what’s a dæmon process anyway?

BSD Daemon Copyright 1988 by Marshall Kirk McKusick
All Rights Reserved.

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
- resource leaks eventually surface
- 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`
- relay information via *event logging*, often done using *e.g.*, `syslog(3)`

BSD Daemon Copyright 1988 by Marshall Kirk McKusick
All Rights Reseived.

(Shared) Libraries

Linking and Loading

Executable and Linkable Format

Compilers produce, and linkers and loaders operate on *object files*. Just like other files, they have specific formats such as *e.g.*, assembler output (`a.out`), Common Object File Format (`COFF`), `Mach-O`, or `ELF`.

- *executable* – just what it sounds like (*e.g.*, `a.out`)
- *core* – virtual address space and register state of a process; debugging information (`a.out.core`)
- *relocatable file* – can be linked together with others to produce a shared library or an executable (*e.g.*, `foo.o`)
- *shared object file* – position independent code; used by the dynamic linker to create a process image (*e.g.*, `libfoo.so`)

Executable and Linkable Format

```
$ cc -Wall -c main.c
$ hexdump -C main.o | head -2
```

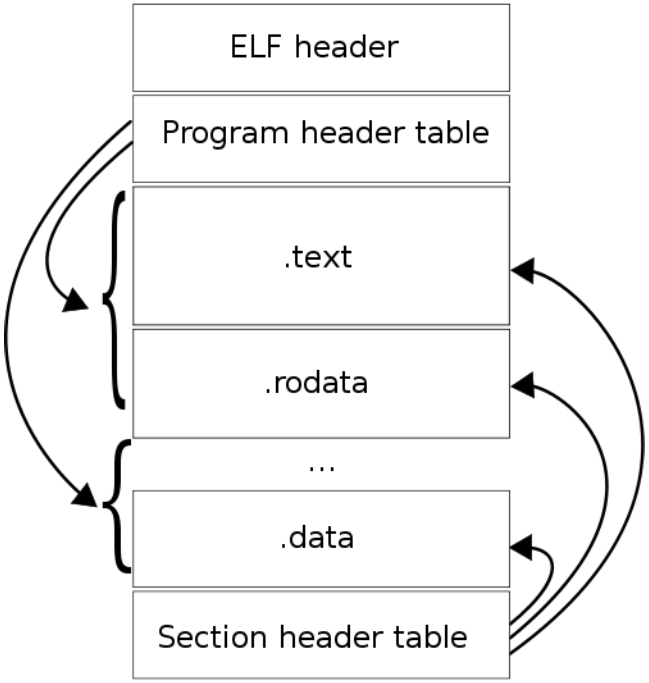
```
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
00000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00
```

```
$ file main.o
```

```
main.o:  ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
not stripped
```

Executable and Linkable Format

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



More details: `https://stevens.netmeister.org/631/elf.html`
`https://www.thegeekstuff.com/2012/07/elf-object-file-format/`

Executable and Linkable Format

```
typedef struct {
        unsigned char   e_ident[ELF_NIDENT];    /* Id bytes */
        Elf64_Half      e_type;                 /* file type */
        Elf64_Half      e_machine;              /* machine type */
        Elf64_Word      e_version;              /* version number */
        Elf64_Addr      e_entry;                /* entry point */
        Elf64_Off       e_phoff;                /* Program hdr offset */
        Elf64_Off       e_shoff;                /* Section hdr offset */
        Elf64_Word      e_flags;                /* Processor flags */
        Elf64_Half      e_ehsize;               /* sizeof ehdr */
        Elf64_Half      e_phentsize;            /* Program header entry size */
        Elf64_Half      e_phnum;                /* Number of program headers */
        Elf64_Half      e_shentsize;            /* Section header entry size */
        Elf64_Half      e_shnum;                /* Number of section headers */
        Elf64_Half      e_shstrndx;             /* String table index */
} Elf64_Ehdr;
```

Executable and Linkable Format

```
$ hexdump -C a.out | head -2
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
00000010 02 00 3e 00 01 00 00 00 e0 07 40 00 00 00 00 00
```

```
$ readelf -h a.out
ELF Header:
 Magic:                7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class:                ELF64
 Data:                 2's complement, little endian
 Version:              1 (current)
 OS/ABI:               UNIX - System V
 ABI Version:          0
 Type:                 EXEC (Executable file)
 Machine:              Advanced Micro Devices X86-64
 Version:              0x1
 Entry point address:  0x4007e0
 ...
```

Linking and Loading

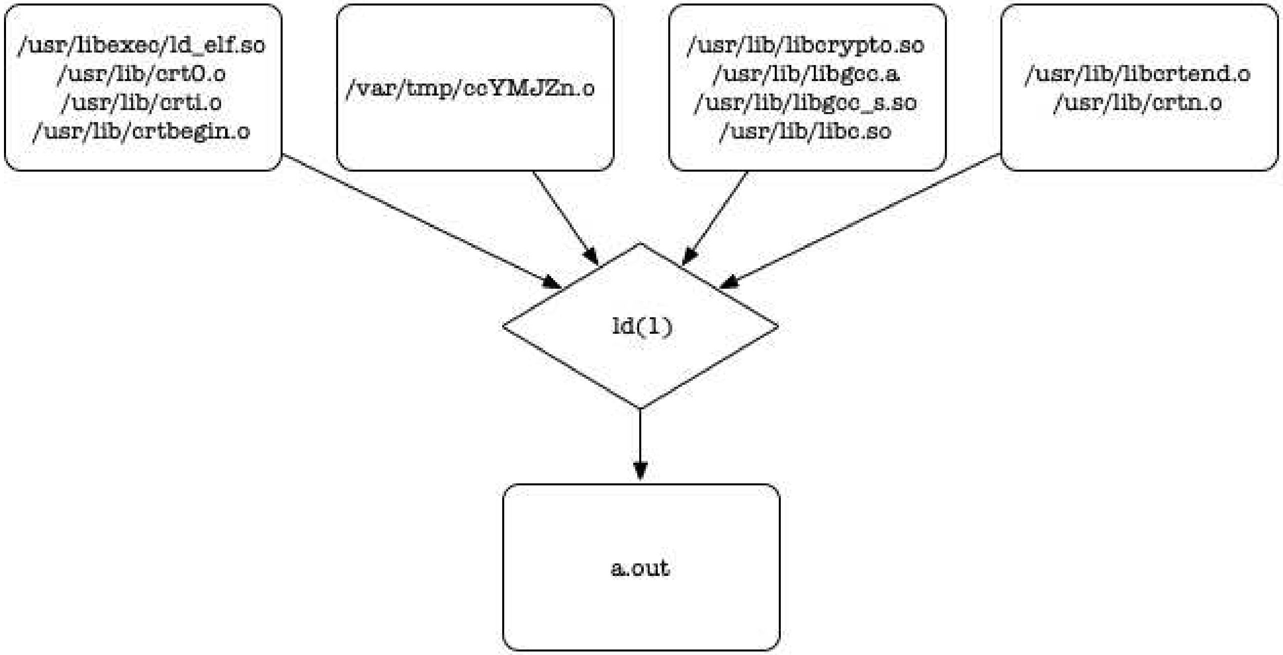
Let’s revisit our lecture on Unix tools and the compile chain:

The compiler chain or driver usually performs preprocessing (*e.g.*, via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ cc -c hello.c
$ ld hello.o
[...]
$ ld hello.o -lc
[...]
$ cc -v hello.o
[...]
$ ld -dynamic-linker /usr/libexec/ld.elf_so            \
        /usr/lib/crt0.o /usr/lib/crti.o                \
        hello.o -lc /usr/lib/crtn.o
$ file a.out
$ ./a.out
```

Linking and Loading

A linker takes multiple *object files*, resolves symbols to *e.g.*, addresses in *libraries* (possibly relocating them in the process), and produces an *executable*.


/usr/libexec/ld_elf.so
/usr/lib/crt0.o
/usr/lib/crti.o
/usr/lib/crtbegin.o

/var/tmp/ccYMJZn.o

/usr/lib/libcrypto.so
/usr/lib/libgcc.a
/usr/lib/libgcc_s.so
/usr/lib/libc.so

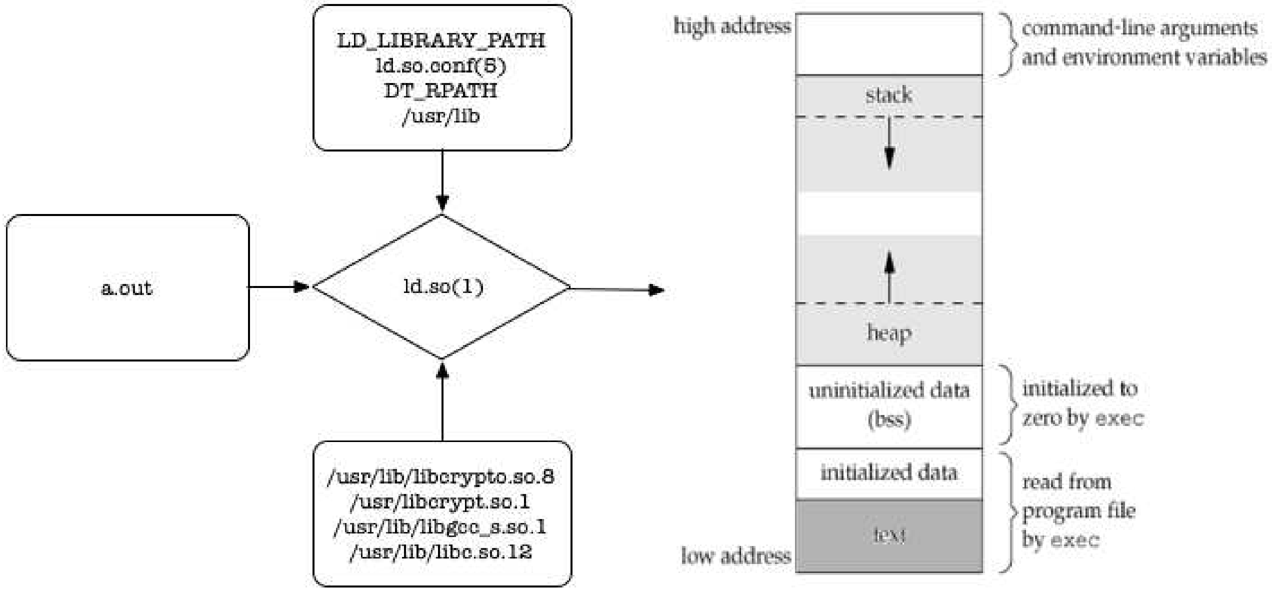
/usr/lib/libcrtend.o
/usr/lib/crtn.o

ld(1)

a.out

Linking and Loading

A loader copies a program into main memory, possibly invoking the *dynamic linker* or *run-time link editor* to find the right libraries, resolve addresses of symbols, and relocate them.



Executable and Linkable Format

```
00000020 40 00 00 00 00 00 00 00 70 14 00 00 00 00 00 00
00000030 00 00 00 00 40 00 38 00 06 00 40 00 1e 00 1b 00
00000040 06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00
00000070 08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00
00000080 90 01 00 00 00 00 00 00 90 01 40 00 00 00 00 00
00000190 2f 75 73 72 2f 6c 69 62 65 78 65 63 2f 6c 64 2e
000001a0 65 6c 66 5f 73 6f 00 00 07 00 00 00 04 00 00 00
```

Translated:

- There are 6 program headers, starting at offset 64.
- The size of the `ehdr` is `0x38`.
- Program Header Table (`PHDR`) with segment permissions read + exec of size 0x40.
- Next header at offset `0x38 + 0x40 = 0x78`:
- `PT_INTERP` with read permissions at offset `0x0190`
- `0x190` contains the interpreter `/usr/libexec/ld.elf_so`

Executable and Linkable Format

```
$ readelf -l a.out

Elf file type is EXEC (Executable file)
Entry point 0x400520
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000150 0x0000000000000150 R E    8
  INTERP         0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000017 0x0000000000000017 R      1
      [Requesting program interpreter: /usr/libexec/ld.elf_so]
```

Linking and Loading

Compare:

```
$ ld
        /usr/lib/crt0.o /usr/lib/crti.o                \
        hello.o -lc /usr/lib/crtn.o
$ readelf -l a.out
$ ./a.out
```

To:

```
$ ld -dynamic-linker /usr/libexec/ld.elf_so            \
        /usr/lib/crt0.o /usr/lib/crti.o                \
        hello.o -lc /usr/lib/crtn.o
$ readelf -l a.out
$ ./a.out
```

Executable and Linkable Format

```
$ hexdump -C /lib/libc.so | head -2
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
00000010 03 00 3e 00 01 00 00 00 70 b7 03 00 00 00 00 00
```

```
$ readelf -h /lib/libc.so
ELF Header:
 Magic:                7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class:                ELF64
 Data:                 2's complement, little endian
 Version:              1 (current)
 OS/ABI:               UNIX - System V
 ABI Version:          0
 Type:                 DYN (Shared object file)
 Machine:              Advanced Micro Devices X86-64
 Version:              0x1
 Entry point address:  0x3b770
 ...
```

Shared Libraries

What is a shared library, anyway?

- contains a set of callable C functions (*i.e.*, implementation of function prototypes defined in `.h` header files)
- code is position-independent (*i.e.*, 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 (*i.e.*, implementation of function prototypes defined in `.h` header files)
- code is position-independent (*i.e.*, 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

```
#include <openssl/rand.h>
int main(int argc, char **argv) {
        int i; unsigned char data[NUM];

        if (RAND_bytes(data, NUM) == 0)
                err(EXIT_FAILURE, "Unable to generate random data: %s\n",
                                strerror(errno));
        for (i=0; i<NUM; i++)
                printf("%02X", data[i]);
        printf("\n");
        exit(EXIT_SUCCESS);
}
$ cc -Wall -c rand.c
$ cc -Wall rand.o
rand.o: In function ‘main’:
rand.c:(.text+0x1c): undefined reference to ‘RAND_bytes’
$ cc -Wall rand.o -lcrypto
```

Shared Libraries

How do shared libraries work?

- at *link time*, the linker resolves undefined symbols
- contents of object files and *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

Understanding object files

```
$ cc -Wall ldtest1.c ldtest2.c main.c
$ nm a.out
                 U _libc_init         # undefined
00000000004007a0 T _start             # defined in the Text segment
                 U atexit             # undefined
0000000000600ea0 B environ            # defined in th BSS segment
                 U exit               # undefined
0000000000400990 T ldtest1            # defined in the Text segment
00000000004009b4 T ldtest2            # defined in the Text segment
00000000004009d8 T main               # defined in the Text segment
                 U printf             # undefined
$ ldd a.out
a.out:
        -lc.12 => /usr/lib/libc.so.12
```

See also: `objdump -t a.out`

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
$ nm libldtest.a

ldtest1.o:
0000000000000000 T ldtest1
                 U printf

ldtest2.o:
0000000000000000 T ldtest2
                 U printf
$ objdump -x libldtest.a
```

Statically Linked Shared Libraries

```
$ cc -Wall main.c libldtest.a
$ mv libldtest.a /tmp/
$ cc -Wall main.c libldtest.a
$ cc -Wall main.c -lldtest
$ cc -Wall main.c -L/tmp -lldtest -o a.out.dyn
$ cc -static main.o -L/tmp -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

Dynamic libraries:

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

Dynamically Linked Shared Libraries

```
$ cc -Wall -c -fPIC ldtest1.c 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
ld: cannot find -lldtest
$ mv /tmp/libldtest.a lib
$ cc -static -Wall main.o -L./lib -lldtest
$ ./a.out
[...]
$ 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 # note: order of directories matters
$ 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 2>/dev/null
[...]
$
```

Dynamically Linked Shared Libraries

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

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 rand.c -lcrypto
$ cc -Wall -rdynamic dlopenex.c
$ ./a.out
```

Reading

Half of this lecture in a single graphic: `https://is.gd/v8eVFI`

- `ld.so(1)`, `elf(5)`
- `https://www.bell-labs.com/usr/dmr/www/man51.pdf`
- `https://www.iecc.com/linker/`
- `https://stevens.netmeister.org/631/elf.html`
- `https://is.gd/TbAYGh`
- `https://is.gd/H3Oj0B`
- `https://is.gd/v45CcV`
- `https://is.gd/MxHrj6`
- `https://is.gd/HLXZOg`
- `https://is.gd/hdCIej`