PWN College

Session 4

Atousa Ahsani

Main Reference: https://pwn.college/

Fundamentals

Computer Architecture

Assembly Code

Introduction to Binary Files

Linux Process Loading

Linux Process Execution

Cat Lifecycle

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

A process is created.

- **fork** and (more recently) **clone** are system calls that create a nearly **exact copy** of the **calling process**.
- Later, the **child** process usually uses the ***execve*** syscall to replace itself with another process.

- Example:
  - You type ***/bin/cat*** in bash.
  - Bash **forks** itself into the old **parent** process and the **child** process.
  - The **child** process ***execves*** /bin/cat, becoming /bin/cat.

Cat is loaded.

- **Can we load it?**
  - Executable Permission
- **What to load?**
  - Kernel checks: Shebang, binfmt_misc, dynamically linked ELF, statically linked ELF, other formats.
- **Loading dynamically linked ELFs**
  1. The **program** and its **interpreter** are loaded by the **kernel**.
  2. The interpreter **locates** the **libraries**.
     - LD_PRELOAD, /etc/ld.so.preload, LD_LIBRARY_PATH, rpath, /etc/ld.so.cache (/etc/ld.so.conf, /lib and /usr/lib)
  3. The **interpreter** loads the **libraries**.

Where is all this getting loaded to?

- Each **Linux process** has its own **virtual memory space**. It contains:
  - The **binary**
  - The **libraries**
  - The "**heap**" (for dynamically allocated memory)
  - The "**stack**" (for function local variables, return addresses, control data)
  - Any memory specifically mapped by the program.
  - **kernel code** in the "**upper half**" of memory (above **0x8000000000000000** on **64-bit architectures**), **inaccessible** to the process.
- All of these virtual memory resides in physical memory. But it is not mapped according to its virtual addresses!
- **Virtual** memory is **dedicated** to your process.
- **Physical** memory is **shared** among the **whole system**.

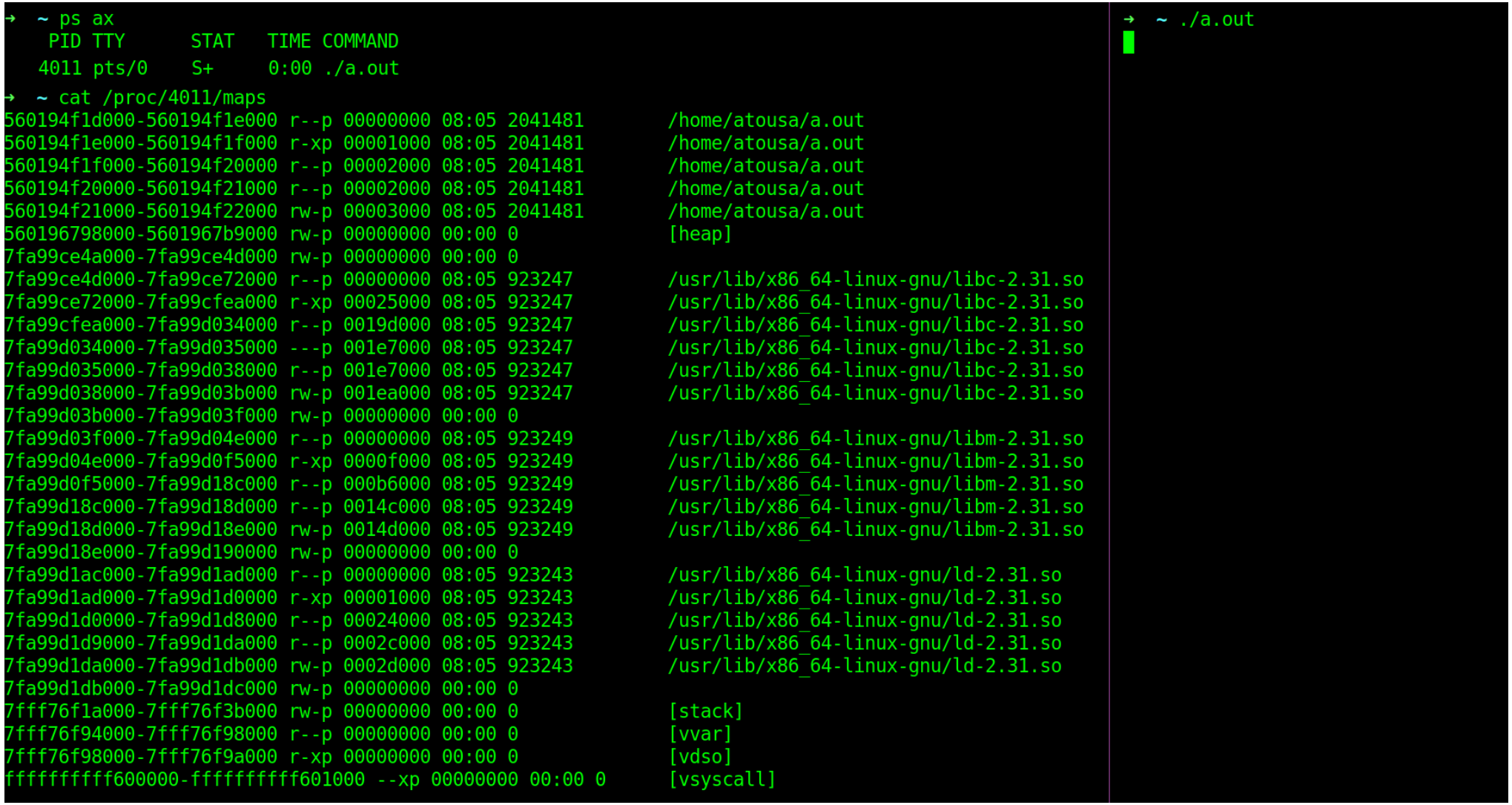
Where is all this getting loaded to?

- You can see this whole space by looking at */proc/self/maps*.

```
→  ~ ./cat /proc/self/maps
Hello
55a2a95be000-55a2a95bf000 r--p 00000000 08:05 1966263                    /home/atousa/cat
55a2a95bf000-55a2a95c0000 r-xp 00001000 08:05 1966263                    /home/atousa/cat
55a2a95c0000-55a2a95c1000 r--p 00002000 08:05 1966263                    /home/atousa/cat
55a2a95c1000-55a2a95c2000 r--p 00002000 08:05 1966263                    /home/atousa/cat
55a2a95c2000-55a2a95c3000 rw-p 00003000 08:05 1966263                    /home/atousa/cat
55a2a9cf0000-55a2a9d11000 rw-p 00000000 00:00 0                          [heap]
7f34cffbe000-7f34cffe3000 r--p 00000000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f34cffe3000-7f34d015b000 r-xp 00025000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f34d015b000-7f34d01a5000 r--p 0019d000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f34d01a5000-7f34d01a6000 ---p 001e7000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f34d01a6000-7f34d01a9000 r--p 001e7000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f34d01a9000-7f34d01ac000 rw-p 001ea000 08:05 923247                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f34d01ac000-7f34d01b2000 rw-p 00000000 00:00 0
7f34d01ce000-7f34d01cf000 r--p 00000000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f34d01cf000-7f34d01f2000 r-xp 00001000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f34d01f2000-7f34d01fa000 r--p 00024000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f34d01fb000-7f34d01fc000 r--p 0002c000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f34d01fc000-7f34d01fd000 rw-p 0002d000 08:05 923243                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f34d01fd000-7f34d01fe000 rw-p 00000000 00:00 0
7ffe76d34000-7ffe76d55000 rw-p 00000000 00:00 0                          [stack]
7ffe76dd5000-7ffe76dd9000 r--p 00000000 00:00 0                          [vvar]
7ffe76dd9000-7ffe76ddb000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Where is all this getting loaded to?

- */proc/self/maps* Fields:
  - ***address***
    - This is the **starting** and **ending** address of the region in the process's address space.
  - ***permissions***
    - r/w/x/p
  - ***offset***
    - If the region was mapped from a **file** (using ***mmap***), this is the **offset** in the **file** where the mapping **begins**. If the memory was **not mapped** from a file, it's just **0**.
  - ***device***
    - If the region was mapped from a **file**, this is the major and minor **device number** (in hex) where the file lives.
  - ***inode***
    - If the region was mapped from a file, this is the file number.
  - ***pathname***
    - If the region was mapped from a **file**, this is the **name** of the file.

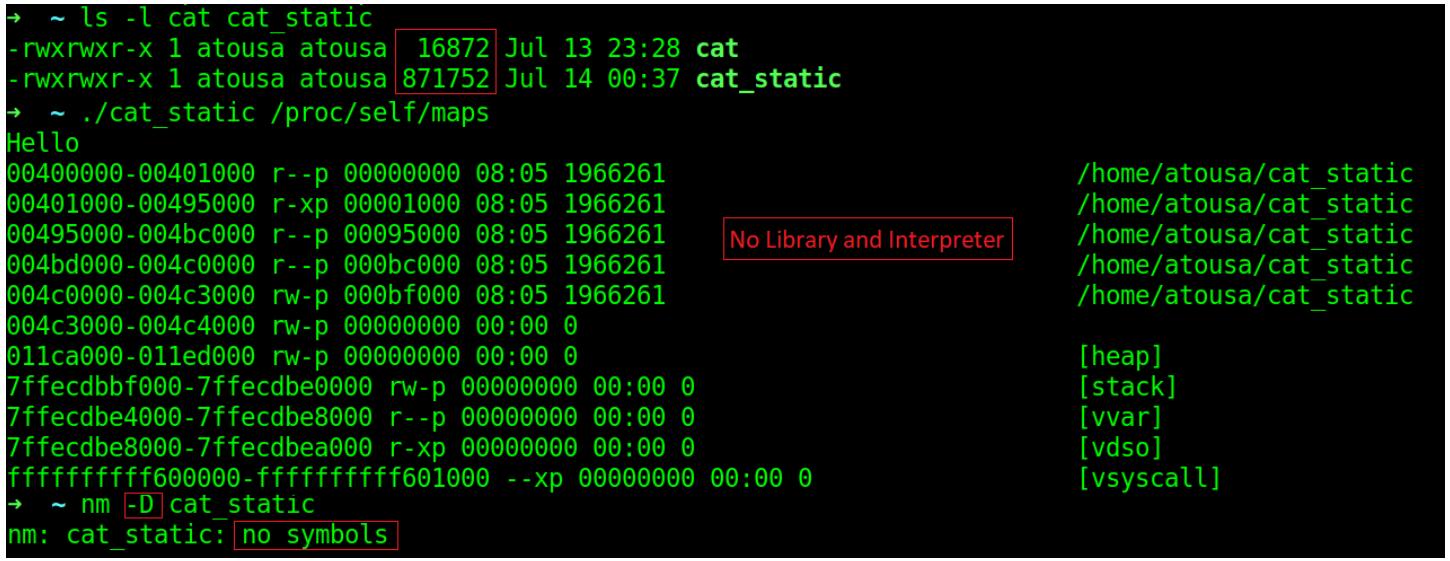

→ ~ ps ax
PID TTY STAT TIME COMMAND
4011 pts/0 S+ 0:00 ./a.out
→ ~ cat /proc/4011/maps
560194f1d000-560194f1e000 r--p 00000000 08:05 2041481 /home/atousa/a.out
560194f1e000-560194f1f000 r-xp 00001000 08:05 2041481 /home/atousa/a.out
560194f1f000-560194f20000 r--p 00002000 08:05 2041481 /home/atousa/a.out
560194f20000-560194f21000 r--p 00002000 08:05 2041481 /home/atousa/a.out
560194f21000-560194f22000 rw-p 00003000 08:05 2041481 /home/atousa/a.out
560196798000-5601967b9000 rw-p 00000000 00:00 0 [heap]
7fa99ce4a000-7fa99ce4d000 rw-p 00000000 00:00 0
7fa99ce4d000-7fa99ce72000 r--p 00000000 08:05 923247 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa99ce72000-7fa99cfea000 r-xp 00025000 08:05 923247 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa99cfea000-7fa99d034000 r--p 0019d000 08:05 923247 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa99d034000-7fa99d035000 ---p 001e7000 08:05 923247 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa99d035000-7fa99d038000 r--p 001e7000 08:05 923247 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa99d038000-7fa99d03b000 rw-p 001ea000 08:05 923247 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa99d03b000-7fa99d03f000 rw-p 00000000 00:00 0
7fa99d03f000-7fa99d04e000 r--p 00000000 08:05 923249 /usr/lib/x86_64-linux-gnu/libm-2.31.so
7fa99d04e000-7fa99d0f5000 r-xp 0000f000 08:05 923249 /usr/lib/x86_64-linux-gnu/libm-2.31.so
7fa99d0f5000-7fa99d18c000 r--p 000b6000 08:05 923249 /usr/lib/x86_64-linux-gnu/libm-2.31.so
7fa99d18c000-7fa99d18d000 r--p 0014c000 08:05 923249 /usr/lib/x86_64-linux-gnu/libm-2.31.so
7fa99d18d000-7fa99d18e000 rw-p 0014d000 08:05 923249 /usr/lib/x86_64-linux-gnu/libm-2.31.so
7fa99d18e000-7fa99d190000 rw-p 00000000 00:00 0
7fa99d1ac000-7fa99d1ad000 r--p 00000000 08:05 923243 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa99d1ad000-7fa99d1d0000 r-xp 00001000 08:05 923243 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa99d1d0000-7fa99d1d8000 r--p 00024000 08:05 923243 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa99d1d9000-7fa99d1da000 r--p 0002c000 08:05 923243 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa99d1da000-7fa99d1db000 rw-p 0002d000 08:05 923243 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa99d1db000-7fa99d1dc000 rw-p 00000000 00:00 0
7fff76f1a000-7fff76f3b000 rw-p 00000000 00:00 0 [stack]
7fff76f94000-7fff76f98000 r--p 00000000 00:00 0 [vvar]
7fff76f98000-7fff76f9a000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
→ ~ ./a.out

The Standard C Library

- **libc.so** is linked by almost every process.
- Provides **functionality** you take for granted:
  - printf()
  - scanf()
  - socket()
  - atoi()
  - malloc()
  - free()
- … and a lot of other stuff!

Statically linked ELFs: the loading process

- The binary is loaded.



- Statically linked ELFs are less secure than dynamically linked ELFs.

Cat Lifecycle

1. A process is created.
2. Cat is loaded.
3. **Cat is initialized.**
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

Cat is initialized.

- Every ELF binary can specify **constructors**, which are **functions** that run **before** the program is actually launched.
- You can specify your own.

```c
__attribute__((constructor)) void myconstructor(){
    puts("Hello from myconstructor!!!");
}

int main(int argc, char **argv)
{
    char buf[1024];
    int n;
    int fd = argc == 1 ? 0 : open(argv[1], 0);
    while((n = read(fd, buf, 1024)) > 0 && write(1, buf, n) > 0);
}
```

```
→  3-ProcessInitialization_cat subl cat.c
→  3-ProcessInitialization_cat gcc cat.c -o cat
→  3-ProcessInitialization_cat ./cat
Hello from myconstructor!!!
^C
```

Fundamentals

Computer Architecture

Assembly Code

Introduction to Binary Files

Linux Process Loading

Linux Process Execution

Cat Lifecycle

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. **Cat is launched.**
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. Cat terminates.

Cat is launched.

- A normal **ELF** automatically calls **__libc_start_main**() in **libc**, which in turn calls the **program's main**() function.

Cat Lifecycle

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. **Cat reads its arguments and environment.**
6. Cat does its thing.
7. Cat terminates.

Cat reads its arguments and environment.

- Your process's entire **input** from the outside world, at launch, comprises of:
  - the **loaded objects** (binaries and libraries)
  - **command-line arguments** in ***argv***
  - "**environment**" in ***envp***

Cat reads its arguments and environment.

```
int main(int argc, void **argv, void **envp);
```

- ***argc***
  - Argument count
  - Length of the argument vector
- ***argv*** is a **tokenized representation** of the **command line** that invoked your program.
  - Argument vector
  - Array of character pointers
- ***envp*** is an array of pointers to **environment variables**.

Cat reads its arguments and environment.

- **Environment Variables:**
  - ***env*** terminal command
  - We can print them in ***main*** function.

```
→  4-LinuxProcessExecution_envp cat env.c
int main(int argc, char **argv, char **envp)
{
        for (int i = 0; envp[i] != 0; ++i)
                puts(envp[i]);
}
→  4-LinuxProcessExecution_envp gcc env.c -o envp
→  4-LinuxProcessExecution_envp ./envp
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/ubuntu:@/tmp/.ICE-unix/1806,unix/ubuntu:/tmp/.ICE-unix/1806
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/2cb56dae_57e3_41dd_968a_62228f83752e
SSH_AGENT_PID=1707
→  4-LinuxProcessExecution_envp AAAA=hello ./envp
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
AAAA=hello
```

Cat reads its arguments and environment.

- ls

```
➜  test env | grep LANG
LANG=en_US.UTF-8
➜  test ls -l
total 0
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 1
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 a
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 A
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 b
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 B
➜  test LANG=C ls -l
total 0
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 1
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 A
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 B
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 a
-rw-rw-r-- 1 atousa atousa 0 Jul 14 03:57 b
```

Cat Lifecycle

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. **Cat does its thing.**
7. Cat terminates.

Using library functions

- The binary's **import symbols** have to be resolved using the **libraries' export symbols**.
- Import symbols in cat:

```
→  2-LinuxProcessLoading_cat nm -D cat
                 w __cxa_finalize
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main
                 U open
                 U puts
                 U read
                 U __stack_chk_fail
                 U write
```

- Export symbols in cat:

```
→  2-LinuxProcessLoading_cat nm -a cat
00000000000011c9 T main
000000000000037c r .note.ABI-tag
0000000000000358 r .note.gnu.build-id
0000000000000338 r .note.gnu.property
                 U open@@GLIBC_2.2.5
0000000000001020 t .plt
0000000000001080 t .plt.got
0000000000001090 t .plt.sec
                 U puts@@GLIBC_2.2.5
                 U read@@GLIBC_2.2.5
```

Interacting with the environment

- Almost **all programs** have to interact with the outside world.
- This is primarily done via **system calls** (***man syscalls***).
- Each system call is well-documented in **section 2** of the man pages (i.e., ***man 2 open***).
- We can trace process system calls using ***strace***.

Signals

- System calls are a way for a **process** to call into the **OS**. But how can the **OS** talk to a **process**?
- **Signals**
  - Signals are **software interrupts** sent to a program to indicate that an important **event** has occurred.
  - Signals **pause** process execution and invoke the **handler**.
  - A **signal handler** is special function that gets executed when a **particular signal** arrives. They take **one argument**: the **signal number**.

Signals

- Full list in **section 7** of man (***man 7 signal***) and ***kill -l***.
- Default Actions:

```
Term   Default action is to terminate the process.
Ign    Default action is to ignore the signal.
Core   Default action is to terminate the process and dump core (see core(5)).
Stop   Default action is to stop the process.
Cont   Default action is to continue the process if it is currently stopped.
```

- Some common signals:

Signal	Standard	Action	Comment	Signal Number
SIGALRM	P1990	Term	Timer signal from **alarm**(2) (used for timers)	14
SIGINT	P1990	Term	Interrupt from keyboard (Ctrl + C)	2
SIGKILL	P1990	Term	Kill signal	9
SIGQUIT	P1990	Core	Quit from keyboard (Ctrl + D)	3
SIGTERM	P1990	Term	Termination signal	15

Signals

- `sighandler_t signal(int signum, sighandler_t handler);`
- Example

```c
int handler(int signal){
    printf("Ding!!!");
    exit(1);
}
int main(){
    alarm(3);
    signal(14, handler);
    while(1);
}
```

- This code prints “Ding!!!” after 3 seconds.

Signals

- Example

```
1  int handler(int signal){
2      printf("Got signal number %d!\n", signal);
3  }
4  int main(){
5      for(int i = 1; i <= 64; i++)
6          signal(i, handler);
7      while(1);
8  }
```

Shared memory

- Another way of interacting with the outside world is by **sharing memory** with **other processes**.

Cat Lifecycle

1. A process is created.
2. Cat is loaded.
3. Cat is initialized.
4. Cat is launched.
5. Cat reads its arguments and environment.
6. Cat does its thing.
7. **Cat terminates.**

Process termination

- Processes **terminate** by one of **two** ways:
  1. Receiving an **unhandled signal**.
  2. Calling the **exit**() system call: `int exit(int status);`
- After termination, all processes must be "**reaped**"
  - After termination, they will remain in a zombie state until they are **wait()ed** on by their parent.
  - When this happens, their **exit code** will be returned to the **parent**, and the process will be **freed**.
  - If their parent dies **without** wait()ing on them, they are **re-parented** to ***PID 1*** and will stay there until they're cleaned up.