

Lecture 8.2

execve and Scheduling

how a file becomes a process, and who gets the CPU

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
./stack hello world (real)

argc = 3
argv[1] at 0x7ffe5cfa236e -> "hello"
auxv AT_PHDR 0x55f059196040
auxv AT_BASE 0x7efc626ee000
auxv AT_ENTRY 0x55f059197080
# all put there by the kernel
# before the first instruction
```

What you will learn today



Trace `execve` from the system call to `load_elf_binary`, and say what each step does.



Draw the first stack of a new program: `argc`, `argv`, `envp`, `auxv`.



Explain how `#!` scripts run, and the errors `ENOEXEC` and `EACCES`.



Predict CPU shares from nice values, and explain how `EEVDF` picks the next task.

Today's plan (60 min)

0-5

Recap of 8.1

5-22

`execve` inside the kernel

22-32

The first stack

32-38

Scripts and errors

38-55

Scheduling

55-60

Checkpoint

Lab 8.2: `stack.c` and the nice experiment.

Quick recap of Lecture 8.1

Q1

How many lines: three fork() calls in a loop, then one printf?

$8 = 2^3$. Every fork doubles the processes that go on through the loop (Homework 8.1, run: 8).

Q2

What does exec keep, and what does it replace?

Keeps: the PID, the parent, open files. Replaces: code, data, heap, stack. Today: how.

Q3

Parent and child print the same &x but different x. Why?

The address is virtual: each process has its own memory behind it (copy-on-write).

PART 1

execve inside the kernel

From one system call to a new program in memory: fs/exec.c and fs/binfmt_elf.c.

The path: from `execve()` to `load_elf_binary()`

- 1 `SYSCALL_DEFINE3(execve)` (`exec.c` 2129): user space enters here (Lecture 9: how).
- 2 `do_execveat_common` (1908): open the file, asking for **execute** permission (line 917); copy `argv` and `envp` in.
- 3 `bprm_execve` → `exec_binprm` (1853, 1808): at most 5 rounds of rewriting (`#!` in `#!`), then **ELOOP** (1822).
- 4 `search_binary_handler` (1761): read the first **256 bytes** of the file into `bprm->buf`.
- 5 Ask each registered format: "is this yours?" ELF, `#!` script, ...

```
fs/exec.c, Linux 6.8 (real lines)
1776 retry:
1778 list_for_each_entry(fmt, &formats, lh) {
    ...
1783     retval = fmt->load_binary(bprm);
    ...
1787     if (bprm->point_of_no_return ||
        retval != -ENOEXEC)
1789         return retval;
1791 }

binfmt_elf.c 103:
    .load_binary = load_elf_binary,
```

C link: a list of structs holding **function pointers**, called in turn, like the comparator you pass to `qsort`.

Book: Love, LKD Ch. 3; Bovet & Cesati Ch. 20 (program execution)

load_elf_binary(): four phases (binfmt_elf.c, real line numbers)

CHECK

843

magic 7f 45 4c 46?

846

type EXEC or DYN?

855

read program headers

868

PT_INTERP: need ld.so?

NO RETURN

996

begin_new_exec: drop old memory

1013

setup_arg_pages: stack

exec.c 1273

point_of_no_return = true

MAP

1027

loop over PT_LOAD

1132

elf_load: mmap + zero .bss

1189

e_entry + load_bias

1200

load ld.so if needed

STACK + RUN

1237

map the vDSO

1242

create_elf_tables: argc ... auxv

1264

heap start (brk)

1298

START_THREAD: rip, rsp

Point of no return: before begin_new_exec, any error goes back to the caller as -1 (the old program keeps running). After it, the old program is gone: if loading fails now, the kernel kills the process with SIGSEGV (bprm_execve, exec.c 1898-1899).

Book: Bovet & Cesati Ch. 20 (the exec functions)

Watch the kernel build cat's memory

execve: the kernel loads a program (Ubuntu 6.8 guest)

CHECKMAPSTACKRUN

the file /bin/busybox (cat)

LOAD 1 R ELF header, program headers
offset 0x000000, 0x0006c8 bytes

LOAD 2 R E code (.text)
offset 0x001000, 0x1a7865 bytes

LOAD 3 R read-only data
offset 0x1a9000, 0x0521e5 bytes

LOAD 4 RW data + .bss
offset 0x1fb820, 0x009568 bytes

entry 0x410870 · 10 program headers · no INTERP

the process's memory map (old program: sh)

sh's code, data, heap, stack ...

(all of it is about to go)

fs/binfmt_elf.c

Inside the QEMU guest the shell runs: `execve("/bin/cat", ["cat", "/proc/self/maps"], envp)`. `/bin/cat` is busybox, a static program. Watch the kernel build cat's memory.

Watch for

1. The 4 bytes **7f E L F** pick the loader.
2. **No PT_INTERP**: busybox is static.
3. Each PT_LOAD becomes **one line** of maps.
4. **.bss**: memory with no file behind it.
5. **rip = e_entry**, **rsp = argc**.

Click the video to play or pause. About 1.5 minutes, no sound.

Who runs first: your `_start`, or `ld.so`?

```
fs/binfmt_elf.c, Linux 6.8 (real lines)
1189 e_entry = elf_ex->e_entry + load_bias;
1199 if (interpreter) { /* PT_INTERP */
1200     elf_entry = load_elf_interp(...);
1209     interp_load_addr = elf_entry;
1210     elf_entry += interp_elf_ex->e_entry;
1224 } else {
1225     elf_entry = e_entry;
}
```

```
strace ./stack a (first calls after exec, real)
execve("./stack", [ "./stack", "a" ], ...) = 0
brk(NULL) = 0x5645ba851000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT
openat(..., "/etc/ld.so.cache", ...) = 3
```

	busybox (static, guest)	./stack (PIE, our machine)
PT_INTERP	none	/lib64/ld-linux-x86-64.so.2
AT_BASE	0x0	0x7efc626ee000 (ld.so)
first code	busybox's <code>_start</code>	ld.so's <code>_start</code>
load address	fixed 0x400000	random 0x55f059196000

No system call loads `ld.so`. The kernel maps it during `execve` (line 1200) and jumps to it. The calls above are `ld.so`'s own work: finding `libc` (Lecture 5.2). Then `ld.so` jumps to your program's `_start`, whose address it finds in `AT_ENTRY`.

PART 2

The first stack of a new program

What the kernel leaves for `main(argc, argv, envp)` before the first instruction.

The stack at the first instruction (create_elf_tables)



The kernel writes from the top down: strings first, argc last. The arrow shows the order.

```
gdb: starti ./tiny one two (real)
0x0000000000400078 in ?? ()
(gdb) info registers rip rsp
rip 0x400078
rsp 0x7fffffff8b10
(gdb) x/4gx $rsp
...8b10: 0x0000000000000003 0x00007fffffff9330
...8b20: 0x00007fffffff934a 0x00007fffffff934e
(gdb) x/s 0x7fffffff9330
"/home/claude/l8/demo/tiny"
```

Check the gaps: $0x934a - 0x9330 = 26$ = the path's 25 letters + $\backslash 0$ '. $0x934e - 0x934a = 4$ = "one" + $\backslash 0$ '. The pointers point at the strings above them.

Book: CS:APP §8.4.5 (Figure 8.22)

Reading it from C: argv, envp, then the auxiliary vector

lab/stack.c (core lines)

```
int main(int argc, char *argv[], char *envp[])
    int n = 0;
    while (envp[n]) n++;
    Elf64_auxv_t *a = (Elf64_auxv_t *) (envp + n + 1);
    for (; a->a_type != AT_NULL; a++) ...
```

./stack hello world (our machine, real)

```
envp: 203 strings, the first at 0x7ffe5cfa237a
auxv AT_PAGESZ          0x1000
auxv AT_PHDR (headers)  0x55f059196040
auxv AT_PHNUM (how many) 0xd
auxv AT_BASE (ld.so)     0x7efc626ee000
auxv AT_ENTRY (_start)   0x55f059197080
main is at 0x55f059197169
```

fs/binfmt_elf.c 241-261 (real, some lines)

```
241 NEW_AUX_ENT(AT_HWCAP, ELF_HWCAP);
242 NEW_AUX_ENT(AT_PAGESZ, ELF_EXEC_PAGESIZE);
244 NEW_AUX_ENT(AT_PHDR, phdr_addr);
247 NEW_AUX_ENT(AT_BASE, interp_load_addr);
251 NEW_AUX_ENT(AT_ENTRY, e_entry);
257 NEW_AUX_ENT(AT_RANDOM, ...u_rand_bytes);
261 NEW_AUX_ENT(AT_EXECFN, bprm->exec);
```

Who reads auxv?

ld.so: AT_PHDR finds your program's headers, AT_ENTRY says where to jump. libc: AT_RANDOM for the stack canary.

Try it

`LD_SHOW_AUXV=1 /bin/true` makes ld.so print the whole list (22 entries on our machine). `getauxval()` reads one entry.

PART 3

#! scripts, and when exec says no

The second binary format, and the errors you will meet in Lab 8.1's shell.

#!: the kernel rewrites argv and starts again

```
● ● ● hashbang (a two-line file, chmod +x) (real)

#!/bin/echo I was started with:
this line is never read
$ ./hashbang a b
I was started with: ./hashbang a b
$ strace -e trace=execve ./hashbang a b
execve("./hashbang", [".hashbang", "a", "b"], ...) = 0
```

```
● ● ● fs/binfmt_script.c, Linux 6.8 (real lines)

41 if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!'))
42     return -ENOEXEC;
77 i_name = next_non_spaceta(bprm->buf+2, i_end);
116 copy_string_kernel(i_arg, bprm); /* optional */
121 copy_string_kernel(i_name, bprm);
132 file = open_exec(i_name);
    /* back to search_binary_handler */
```

The new argv, built by the kernel:

/bin/echo	interpreter
I was started with:	its one argument
./hashbang	the script
a	
b	

One execve, two loads. strace shows only one call: the kernel itself opened /bin/echo and ran search_binary_handler again. That is how `#!/usr/bin/python3` works: Python gets the script's name as an argument. A `#!` may point to another script, at most 5 levels deep (ELOOP).

Book: Kerrisk, TLPI §27.3 (interpreter scripts)

When execve says no: three errors, and one rescue

```
● ● ● run.c calls execv() directly (real)
$ ./run ./nosuchfile
./nosuchfile: No such file or directory
$ ls -l noexec          # an ELF file, no x bit
-rw-r--r-- 1 root root 26936 ... noexec
$ ./run ./noexec
./noexec: Permission denied
$ ./run ./noshebang     # text, no #! line
./noshebang: Exec format error
```

```
● ● ● strace -f bash -c ./noshebang (real)
[pid 5248] execve("./noshebang", ...) = -1 ENOEXEC
I have no #! line
[pid 5248] +++ exited with 0 +++
```

errno	Meaning	Where it is decided
ENOENT	no such file	opening the file
EACCES	no execute permission	do_open_execat, MAY_EXEC (917)
ENOEXEC	no format said "mine"	search_binary_handler
ELOOP	#! nested > 5 levels	exec_binprm (1822)

The rescue is bash, not the kernel. When execve fails with ENOEXEC, bash (and dash) read the file as a shell script themselves, in the child. Our mysh just reports the error. Even **root** gets EACCES on a file with no x bit at all.

PART 4

Scheduling: who gets the CPU

Run queues, nice values, EEVDF, and context switches you can count.

More R tasks than CPUs: someone must choose

Each CPU has a **run queue** (struct rq) of tasks in state R. The scheduler, `schedule()`, picks the next one. It runs when:

- blocks** the task waits (read, sleep): **voluntary** switch
- tick** the timer says its turn is used up: **involuntary**
- wakes** a woken task deserves the CPU now: **preemption**

```
● ● ● chrt -m ; chrt -p $$ (real)
      SCHED_OTHER min/max priority      : 0/0
      SCHED_FIFO  min/max priority      : 1/99
      SCHED_RR    min/max priority      : 1/99
      SCHED_DEADLINE min/max priority : 0/0
      pid 5365's current scheduling policy: SCHED_OTHER
```

Class (checked in this order)	Policies	Used for
stop	(kernel only)	stopping a CPU, migration
deadline	SCHED_DEADLINE	tasks with a time budget
real-time	SCHED_FIFO, SCHED_RR (1-99)	audio, control loops
fair (EEVDF)	SCHED_OTHER, BATCH, IDLE	almost everything: nice -20 ... 19
idle	(kernel only)	the idle loop when nothing is R

Book: Love, LKD Ch. 4 (process scheduling)

nice → weight → share of the CPU

nice	-10	-5	0	1	3	5	10	19
weight	9548	3121	1024	820	526	335	110	15

from `sched_prio_to_weight[]`, kernel/sched/core.c 11497. Each step of nice $\approx \times 1.25$.

Two busy tasks on one CPU:
share of A = $w(A) / (w(A) + w(B))$

shares.c: nice 0 against ...		Predicted	Measured, our machine	Measured, QEMU guest (6.8)
nice 1	1024 / 1844	55.5 %	55.5 %	
nice 5	1024 / 1359	75.3 %	75.4 %	75.3 %
nice 10	1024 / 1134	90.3 %	90.3 %	90.3 %
nice 19	1024 / 1039	98.6 %	98.5 %	

A ratio, not a priority. Raising nice by 1 always costs about 10 % of the CPU, whatever the starting point. A nice 19 task still runs: it just gets 1.5 %.

C link: the kernel has no floating point here: shares come from integer weights and a fixed-point multiply.

Book: Love, LKD Ch. 4 (nice values, the weight table)

EEVDF: Earliest Eligible Virtual Deadline First

```
kernel/sched/fair.c, Linux 6.8 (real lines)

/* every tick: charge the running task */
1168 curr->vruntime += calc_delta_fair(delta_exec, curr);
/* calc_delta_fair = delta × 1024 / weight (296) */
988 se->slice = sysctl_sched_base_slice; /* 0.75 ms */
993 se->deadline = se->vruntime +
      calc_delta_fair(se->slice, se);
/* 863: pick 1) eligible (owed CPU time)
   2) earliest virtual deadline */
```

vruntime: CPU time, scaled: a nice 5 task (weight 335) is charged $1024/335 \approx 3\times$ faster.

eligible: $\text{vruntime} \leq \text{weighted average}$: it has received no more than its share.

deadline: $\text{vruntime} + \text{one slice}$, scaled the same way. Earliest deadline wins.

turn	A: v / deadline	B: v / deadline	eligible	runs
1	0.00 / 0.75	0.00 / 2.29	A B	A
2	0.75 / 1.50	0.00 / 2.29	B	B
3	0.75 / 1.50	2.29 / 4.59	A	A
4	1.50 / 2.25	2.29 / 4.59	A	A
5	2.25 / 3.00	2.29 / 4.59	A	A
6	3.00 / 3.75	2.29 / 4.59	B	B

A = nice 0, B = nice 5, one CPU (a model of these rules, `eevdf_sim.py`, in ms of virtual time). Pattern: **A B A A A B A A A B ...**: A runs 3 turns for each of B's, $\approx 1024 : 335$. Over 4000 turns the model gives 75.3 % : 24.6 %, as measured.

Book: LKD Ch. 4 describes CFS; EEVDF replaced it in Linux 6.6

Context switches you can count (ctxsw.c)

/proc/self/status	sleeper: 100 × usleep(10 ms)	hog, our machine (HZ = 250)	hog, QEMU guest (HZ = 1000)
voluntary_ctxt_switches	100	0	0
nonvoluntary_ctxt_switches	0	276	901

Sleeper: 100 naps = 100 voluntary switches: it gave the CPU away each time it called usleep.

Hog: Never gives up the CPU; it is taken away. Two hogs share CPU 0 for 2 s: ≈ 1 s each.

Why 276 and 901: Preemption is checked at the timer tick: 4 ms here, 1 ms in Ubuntu's kernel. $1 \text{ s} / 276 \approx 3.6 \text{ ms}$, $1 \text{ s} / 901 \approx 1.1 \text{ ms}$ per turn.

C link: a context switch is like `setjmp/longjmp` between two programs: save one set of registers and stack pointer, load another. The kernel does it in assembly (`switch_to`).

```
kernel/sched/core.c, Linux 6.8 (real lines)
5656 void scheduler_tick(void) /* every tick */
    → update_curr → resched_curr
6607 static void __schedule(...)
6691     next = pick_next_task(rq, prev, &rf);
6727     rq = context_switch(rq, prev, next, &rf);
5344 context_switch(): switch mm, then
    registers and kernel stack
```

Book: Love, LKD Ch. 4 (preemption and context switching)

LAB 8.2

stack.c and the nice experiment

Read the auxiliary vector yourself; then predict and measure CPU shares.

Lab 8.2 tasks, step by step

- 1 **TODO 1** in `stack_starter.c`: count the `envp` strings (a NULL-ended array, like `argv`)
- 2 **TODO 2**: point an `Elf64_auxv_t` * just past `envp`'s NULL; print until `AT_NULL`
- 3 Compare with `LD_SHOW_AUXV=1 ./stack`; build with `-static` and compare `AT_BASE`
- 4 Predict the shares for nice 1, 5, 10, 19 from the weight table, then run `./shares N 5`
- 5 Run `./ctxsw`; find `CONFIG_HZ` of your kernel and explain the hog's number

Report: a table predicted vs measured for task 4, and one sentence per difference larger than 1 %.

● ● ● expected output (our machine, real)

```
$ ./stack hello world
argc = 3
argv[0] at 0x7ffe5cfa2366 -> "./stack"
argv[1] at 0x7ffe5cfa236e -> "hello"
argv[2] at 0x7ffe5cfa2374 -> "world"
envp: 203 strings, ...
auxv AT_PAGESZ      0x1000
auxv AT_PHDR ...    0x55f059196040
...
$ ./shares 5 5
nice 0 : 2.99 s CPU (75.4%)
nice 5 : 0.98 s CPU (24.6%)
$ ./ctxsw
sleeper voluntary_ctxt_switches: 100
hog      nonvoluntary_ctxt_switches: 276
```

Checkpoint: test yourself

Q1

A file has no x bit. Which errno does execve give, and when?

→ EACCES, when opening the file, before reading a byte

Q2

After begin_new_exec, loading fails. What happens?

→ no return to the old program: the process is killed (SIGSEGV)

Q3

Where does auxv start on the first stack?

→ right after the NULL that ends envp

Q4

Who first runs in a dynamic program after execve?

→ ld.so (its address is AT_BASE); then it jumps to AT_ENTRY

Q5

nice 0 and nice 10 share one CPU. Share of nice 0?

→ $1024 / (1024 + 110) = 90.3 \%$

Key ideas from today



execve: open with MAY_EXEC, read 256 bytes, ask each binary format; -ENOEXEC means "not mine".



load_elf_binary: check, point of no return, mmap each PT_LOAD, build the stack, jump to the entry.



First stack: argc, argv, NULL, envp, NULL, auxv, then the strings. ld.so reads auxv.



#!: the kernel builds argv = interpreter, argument, script, args, and loads again.



nice sets a weight; $\text{CPU share} = \text{weight} / \text{total weight}$. Measured within 0.1 %.



EEVDF: run the eligible task with the earliest virtual deadline; the tick enforces it.

Homework 8.2 (due before Lecture 9)

1. Two #!

script1 is one line `#!/script2`; script2 is one line `#!/bin/echo hi`. Both are executable. Predict the output of `./script1 x`, then run it.

2. Static

Build `stack.c` with `-static`. Which `auxv` entries change compared with the normal build, and why? Run it twice: which addresses change?

3. Predict

`nice 0` against `nice 3` on one CPU: predict the shares from the weight table, then measure with `./shares 3 5`.

4. Three

Change `shares.c` to start three busy tasks: `nice 0`, `nice 0`, `nice 10`. Predict and measure each share.

5. Your HZ

Find `CONFIG_HZ` of your lab PC's kernel. Predict the hog's `nonvoluntary_ctxt_switches` in `ctxsw`, then measure.

Next lecture and reading

NEXT: LECTURE 9

System calls

How a program crosses into the kernel: the syscall instruction, the system call table, registers for arguments, errno, and the vDSO you saw in `/proc/self/maps` today.

Main text

R. Love, **Linux Kernel Development**, 3rd ed., Ch. 4 (today; it describes CFS, the scheduler before EEVDF) and Ch. 5 (system calls) for Lecture 9.

C side

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §7.9 (loading), §8.4.5 (the stack of a new program, Figure 8.22).

Source

Linux 6.8: `fs/exec.c`, `fs/binfmt_elf.c`, `fs/binfmt_script.c`, `kernel/sched/core.c`, `kernel/sched/fair.c` (Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

Lab files

`lab8_files.zip`: `stack_starter.c`, `shares.c`, `ctxsw.c` and the demo programs.