CS380L: Advanced Operating Systems Lab #3

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Environment

Unless otherwise noted, we use a Linux server for all the experiments. The server has 4 Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz processors and 16GB of memory, and runs Ubuntu 16.04.2 LTS (kernel version 4.11.0).

2 ELF

Executable and Linkable Format (ELF) is a file format for executable, object code, shared libraries, and core dump on UNIX-like systems [1]. ELF format is shown in Figure 1, which is taken from [2]. An ELF binary starts with a fixed-length *ELF header*, which holds a "road map" describing the file's organization (e.g., instruction set architecture, endianness, system ABI, program's entry point). A *program header table*, if present, tells the system how to create a process image (i.e., execute a program). Each entry of program header table corresponds to a segment in the virtual address space. Part of the segment contents can be loaded from the executable file, which is also defined in the program header table. A *section header table* contains information describing the file's sections (e.g., `.text`, `.data`, `.rodata`, `.bss`). Some of them will be loaded into virtual address space as part of segments indicated by program header table. Each entry gives information such as section name, the section size, etc. Program header table is required for to-be-run file and optional for relocatable file; section header table is required for file used during linking and optional for others.

There are some special sections we care about in ELF. `.text` section contains the program's executable instructions; `.rodata` represents read-only data, such as ASCII string constants produced by the C compiler; `.data` holds the program's initialized data, such as global variables declared with initializers like `int x = 5;`; `.bss` section holds uninitialized data (e.g., uninitialized global variables such as `int x;`) that contribute to the program?s memory image. `.bss` immediately follows `.data` in memory. C requires that "uninitialized" global variables start with a value of zero. Thus, the system initializes the data with zeros when the program begins to run. The section occupies no file space (linker records just the address and size of the `.bss` section). The loader or the program itself must arrange to zero the `.bss` section.

[1] 30 hours spent on this lab.

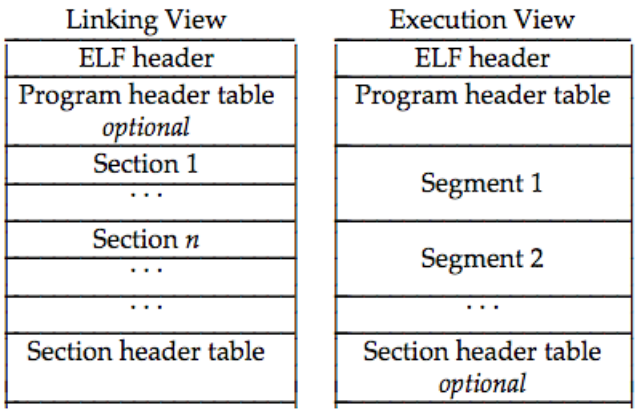

Figure 1: ELF format. There are two views of ELF binary: linking view and execution view. Linking view is used during the dynamic linking, whereas execution view is needed when load the program into memory for the program to run.

To gather better sense of ELF, we write a simple "Hello World" program (`test_helloworld.c`) shown below to study its ELF content.

```
#include <stdio.h>

int main() {
  printf("Hello world.\n");
  return 0;
}
```

We compile the program via `gcc -static -O0 -g -std=c11 -o test_helloworld test_helloworld.c -lpthread` and then inspect its ELF content via `readelf -l test_helloworld`. The printout of the `readelf` is shown in Figure 2. We see there are six segments with five kinds of `p_type`: `PT_LOAD`, `PT_NOTE`, `PT_TLS`, `PT_GNU_STACK`, and `PT_GNU_RELRO`. `PT_LOAD` indicates loadable segments; `PT_NOTE` specifies the location and size of auxiliary information; `PT_TLS` specifies the thread-local storage template [3]. `PT_GNU_STACK` and `PT_GNU_RELRO` are linux-specific `p_type` values [4]: `PT_GNU_STACK` indicates whether stack is executable and `PT_GNU_RELRO` specifies the location and size of a segment which may be made read-only after relocation shave been processed. As we will see in the following section, linux `execve` implementation mainly concerns about program header table entry with type `PT_LOAD`.

`readelf -l` also gives section to segment mapping. As we can see, multiple sections can be contained within the same segment. For example, the first segment contains sections like `.init` (contains the process initialization code, which is executed before calling the main program entry point), `.text`, `.rodata` and the second segment contains sections like `.data`, `.bss`. Note the second

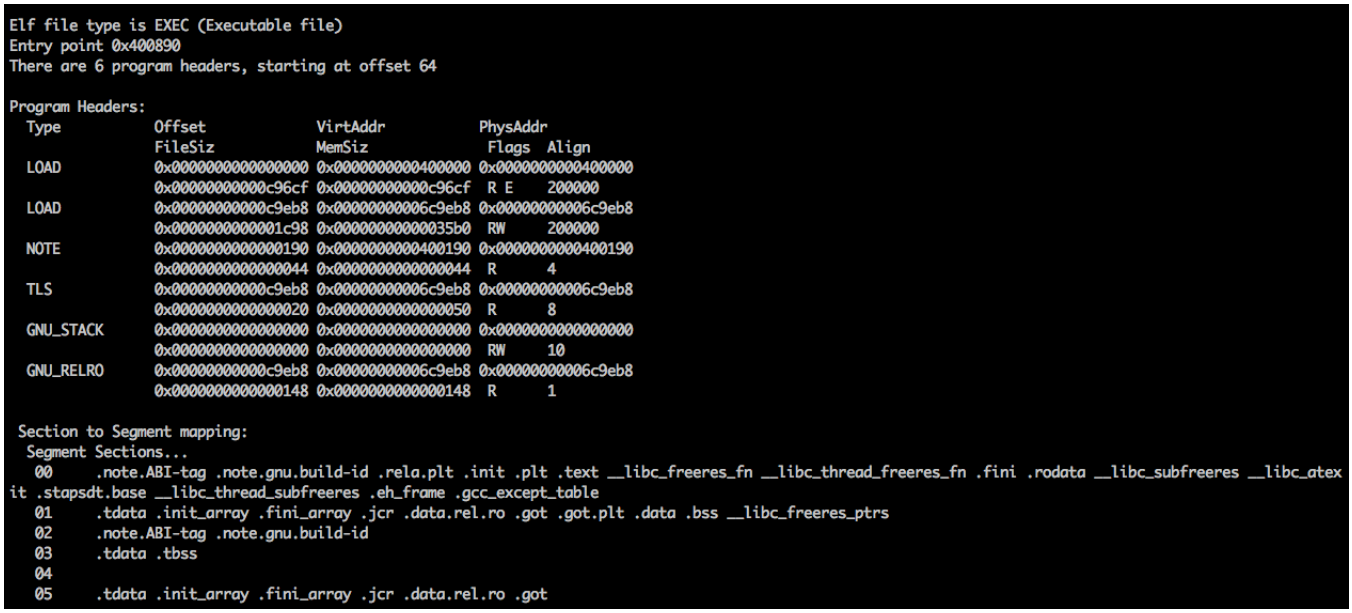

Figure 2: `readelf -l` printout of the `test_helloworld` executable.

segment also contains `.init_array` and `.fini_array`, which are pointers to functions that will be executed when program starts and ends respectively [5]. In addition, we see the memory layout starts from address `0x400000`, which is configured by linker. We need to change this address in some way for our loader program.

3 `execve` Implementation

To write our own loader program, we first need to understand how `execve` system call is implemented in Linux kernel [2]. Essentially, `execve` performs the heavylifting work of loading a program into memory and start the execution of program as a process in Linux. The main logic of `execve` is implemented in `do_execveat_common` in `fs/exec.c`. Inside `do_execveat_common`, we see the critical function call stack looks like below:

```
1 do_execveat_common
2 |- exec_binprm
3    |- search_binary_handler
4       |- load_binary
```

For binary with ELF format, `load_elf_binary` is registered with `load_binary` in `fs/binfmt_elf.c`. Thus, we locate the core function `load_elf_binary`, which is responsible to load binary with ELF format in the Linux kernel. Further study of the function reveals that the following critical steps are performed in order to load the program into memory and start execution:

[2] we study the source code of Linux 4.8.12

- Read the ELF header and perform some simple consistency checks. For example, the function checks whether the binary file is executable file (`ET_EXEC`) or shared object file (`ET_DYN`).
- Read the program header table by calling `load_elf_phdrs`.
- Walk through the entries of the program header table and perform specific actions based on program header table entry type (e.g., `p_type`). Note that `e_phnum` holds the number of entries in the program header table. Each entry of the table corresponds to a memory segment in the binary file. Kernel is only interested in three types of program header entries during the walk through of the program header table:
  - `PT_INTERP` entry, which identifies the run-time linker needed to assemble the complete program [6].
  - `PT_GNU_STACK` entry, which determines whether the program's stack should be executable [7].
  - `PT_LOPROC ... PT_HIPROC`, which are values reserved for processor-specific semantics [6].
- Once all the program header table entries have been processed, the function performs some checks based on `p_type` value it obtained from the previous step (e.g., check for interpreter, check for specific processor architecture).
- The function now is ready to set up the new program. The very first step is to call `flush_old_exec` , which clears up state in the kernel that refers to the previous program. Some minor setups (e.g., set up program's personality [8]) are performed immediately after.
- `setup_new_exec` is called to set up the kernel's internal state for the new program and new credentials for this executable is installed via calling `install_exec_creds`.
- `setup_arg_pages` is invoked to set up kernel's memory tracking structures (e.g., stack `vm_area_struct`). This step is part of the goal to set up virtual memory for the new program.
- The function now traverse the program header table again and look for entries with type `PT_LOAD`, which indicates loadable segments (i.e., areas of the new program's running memory). The entries contain code and data sections that come from the executable file and the size of a `BSS` section. For each `PT_LOAD` entry, the function maps it into the process' address space via `elf_map` call and sets up the new program's memory layout accordingly.
- `set_brk` is invoked to set up zero-filled pages that correspond to the program's BSS [9] segment.
- `create_elf_tables` is called to set up the rest of the new program's stack. Basically, the function puts `argc`, `argv`, `envp`, and auxiliary vectors. LWN article [7] provides an illustration

of the content of the stack set up by the kernel. In addition, Figure 3.11 of System V ABI for `x86_64` manual [10] provides what initial process stack looks like.

- `start_thread` is invoked to start the execution of the new program.

4 User-space Loader

The main goal of this lab is to construct user-space loaders. The main functionality of a loader is to copy the code and data in given executable file into memory and then run the program by jumping to its first instruction (*entry point*). To organize the layout of our process memory (i.e., coexistence of memory segments for both loader program and loaded program), we check `/proc/self/maps` of the previous "hello world" program and the output is shown below:

```
00400000-004ca000 r-xp 00000000 fd:01 12551950 /<-snip->/test_helloworld
006c9000-006cc000 rw-p 000c9000 fd:01 12551950 /<-snip->/test_helloworld
006cc000-006ce000 rw-p 00000000 00:00 0 /<-snip->/test_helloworld
01138000-0115b000 rw-p 00000000 00:00 0 [heap]
7ffdaab86000-7ffdaaba7000 rw-p 00000000 00:00 0 [stack]
7ffdaabd6000-7ffdaabd8000 r--p 00000000 00:00 0 [vvar]
7ffdaabd8000-7ffdaabda000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

There are four memory segments we need to pay attention to: the first two memory segments contain the code and data loaded from executable (e.g., `.data`, `.text`); the third segment corresponds to `.bss` section; heap follows immediately after and stack. Note, the starting address of the first memory segment is `0x400000` and stack resides in very high portion of the address space. In addition, `[vvar]`, `[vdso]`, `[vsyscall]` are not our main concern here as they are dealing with speedup of some syscall. Those observations allude our plan to organize both loader and loaded programs in a single address space as shown in Figure 3. From the figure, one can find that we choose `0x7f400000` as the starting address of the loader program, which has an offset of `0x7f000000` from `0x400000`, which amounts to roughly 2GB for loaded program's segment. This space is necessary as we do not support relocation (i.e., the addresses in the program headers of the loaded program are treated as fixed addresses) and we do not want two program's memory segments overlap.

To such memory organization, we need to modify the starting address of the first memory segment of our loader program, we use a linker script to tell `ld` the intended memory layout. We obtain the default linker script used by `gcc` via `ld --verbose`. We copy the default linker script to some file (e.g., `linker.lds`) and modify the following line:

```
PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x400000)); . =
    SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADERS;
```

by replacing `0x400000` with the desired starting address value of our loader program (e.g., `0x7f400000` in our case).

Besides process memory organization, we also need to figure out how to build loaded program's stack. Based on System V ABI for x86-64 [10], our stack has the layout shown in Figure 4, which is exactly same as Figure 3.11 of the manual.

Now we have all the pieces we need to build a user-space loader and the workflow looks like below:

- Read and parse the ELF executable of the loaded program to extract necessary file header

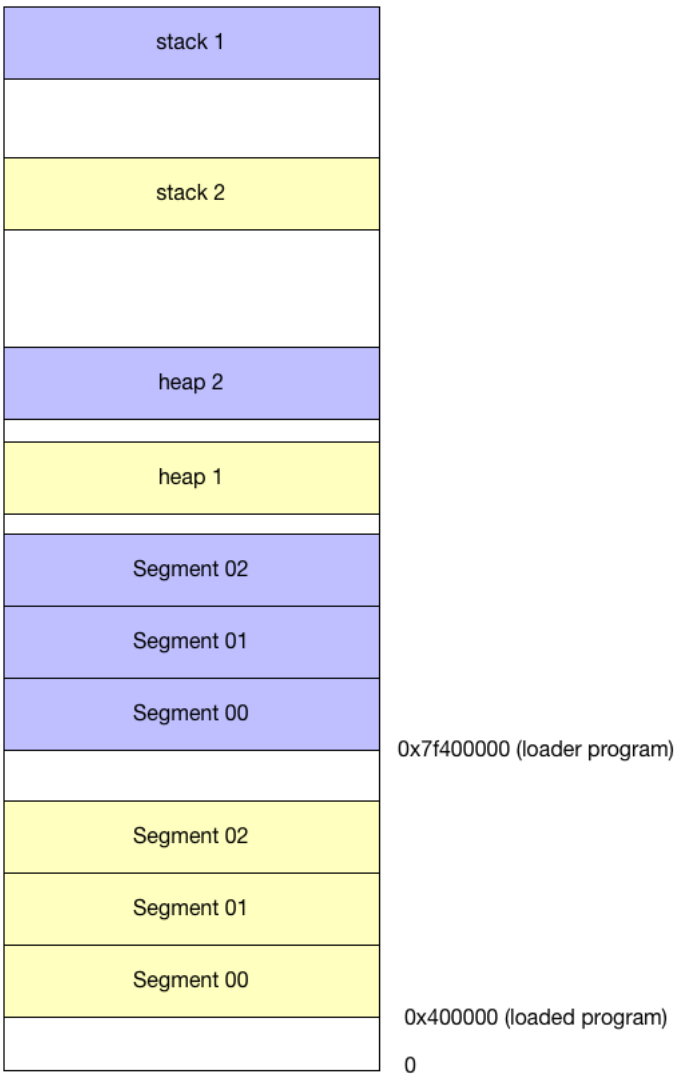

Figure 3: Virtual address space organization of loader and loaded program

Purpose	Start Address	Length
Unspecified	High Addresses	
Information block, including argument strings, environment strings, auxiliary information ...		varies
Unspecified		
Null auxiliary vector entry		1 eightbyte
Auxiliary vector entries ...		2 eightbytes each
0		eightbyte
Environment pointers ...		1 eightbyte each
0	`8+8*argc+%rsp`	eightbyte
Argument pointers	`8+%rsp`	argc eightbytes
Argument count	`%rsp`	eightbyte
Undefined	Low Addresses	

Figure 4: Initial stack structure

table and program header table information

- Load the memory segments of the loaded program into the loader's virtual address space (setup by the kernel's loader). There are three approaches: all-at-once paging, demand paging, and hybrid paging. We will discuss them in details in the following sections. For safety, we need to ensure that memory segments (except stack) of the loaded program do not go beyond `0x7f400000`.

- Build stack for the loaded program. Per the previous discussion, initial stack contains `argc`, `argv`, `envp`, and auxiliary vector entries. We use `mmap` to allocate space for the loaded program's stack (kernel helps to decide which to put stack in the loader's address space). Auxiliary vectors are the same ones provided by kernel to the loader. The only place we need to change is to manually setup the auxiliary vector of program headers, which should point to the loaded program's program header entries.

- Setup registers and jump to the entry point of the loaded program. We need to setup `%rsp` register to point to the top of stack. In addition, we have to set `%rdx` register to 0 as `%rdx` is a function pointer that application should register with `atexit` [10].

To carry out the last step, we use the following assembly embedded in C:

```
1 void __attribute__((noinline)) go(void *entry, void *rsp) {
2   __asm__("movq $0, %rdx;"
3           "movq %rsi, %rsp;"
4           "jmp *%rdi;");
5 }
```

Note that according to System V x86 ABI [10], function arguments of the function is passed via registers `%rdi` and `%rsi` respectively.

4.1 All-at-once loading

In this setting, all the loaded program's memory segments are loaded into loader's address space before the execution of the loaded program. In our implementation, memory segments are read and placed in the memory region of address space via `mmap`. The detailed description of the test programs are left to 5 section. We find that when `malloc` invoked by the test program, `brk` syscall is invoked and there is no issue with test program's behavior. This is due to the fact that we setup heap segment for loaded program as well in the loader address space.

4.2 Demand loading

In this setting, we only map a single page of the executable under test. When segmentation fault happens, loader will first try to load the segment indicated by the faulting address. If no such segment is found, that means that a true segmentation fault happens. The loader will call `exit` to stop the execution. we use this method to preserve memory access errors. For example, for the given "memory access errors" program in the lab description, both demand loader and the kernel loader raise the segmentation fault. If the segment indicated by the faulting address is found, loader will set up the page containing the faulting address. Please note that in the loader, only one page will set up, not the entire segment.

Implementation-wise, we leverage `sigaction` to set up the signal handler for `SGSEGV` signal. We use a new stack for the signal handler (created using `sigaltstack`). The main reason for using another stack is that if we use the original stack, the signal handler will not be able to execute when the `SIGSEGV` signal is caused by stack overflow.

4.3 Hybrid loading

When comes to hybrid loading, the loader will map all the segments except `.bss` section before the execution of the loaded program. `.bss` section will be loaded on demand in the same way as demand loading. In the implementation, for each segment, we only set up the part with contents from the loaded program's executable.

Compared with demanding loading in the previous section, demand loading in this case will load more than one page when segmentation fault is triggered: one page that contains faulting address and the other one or two pages based on some heuristics, which is shown in 1. The idea is based on a simple observation: memory access shows strong locality. When p is loaded, we want to look at neighbor pages: p_{prev} and p_{next}. We try to first load p_{next} first indicated by d (initialized to 1). If

the load fails (due to page is already loaded), we switch to opposite direction by setting $d = -1$ and retry. One can see this heuristics aligns with array access very well: when iterate an array in the forwarding direction (starting from element with index 0), we can load p_{next} so that there won't be a segmentation fault for the next element access. Similar situation applies to reverse array access. This heuristics generalize to arbitrary number of pages preloading (e.g., one additional page or two additional pages).

Algorithm 1 Predication heuristics to pick pages to load

```
1: function PICK(addr, num)
2:     p be the page containing the faulting address
3:     p_prev be the page before p
4:     p_next be the page after p
5:     d = 1
6:     n = 1
7:     Load page p that contains addr
8:     while n < num do
9:         if d == 1 then
10:            status = Load p_next
11:            if status != 0 then
12:                d = −1
13:            else
14:                n++
15:            end if
16:        else if d == −1 then
17:            status = Load p_prev
18:            if status != 0 then
19:                d = 1
20:            else
21:                n++
22:            end if
23:        end if
24:    end while
25:    return 0
26: end function
```

The key implementation relies on whether we can decide a page is already loaded. We use a naive approach by maintaining an array of boolean with each entry corresponding to a page in the address space. In specific, since the address of loaded program's memory segment cannot go beyond `0x7f400000` (e.g., the starting address of the loader), the array has size 2^{19}.

5 Evaluation

We design four programs to benchmark loaders with different paging strategy: traversing an array both forward (e.g., `seq`) and backward (e.g., `revseq`); traversing a matrix in row order (e.g., `matrix`) and in column order (e.g., `revmatrix`). Figure 5 illustrates different loader's performance on all these four test programs. Table 1 lists all the user time, system time along with memory usage of each

	matrix			revmatrix			seq			revseq		
	user	sys	mem	user	sys	mem	user	sys	mem	user	sys	mem
apager	**0.926**	**0.024**	392656	**0.927**	**0.027**	392660	**1.761**	**0.073**	1050628	**1.591**	**0.079**	1050628
dpager	1.596	0.278	**392296**	1.572	0.312	**392364**	1.864	0.578	**1050180**	1.704	0.586	**1050180**
hpager (pf0)	1.593	0.268	394184	1.568	0.299	394184	1.829	0.623	1052336	1.683	0.615	1052160
hpager (pf1)	1.558	0.218	394184	1.578	0.234	394184	1.812	0.442	1052336	1.649	0.47	1052160
hpager (pf2)	1.545	0.22	394184	1.588	0.199	394188	1.834	0.372	1052336	1.66	0.401	1052160

Table 1: User time, system time, and memory usage of all loaders on all four benchmark programs. Memory is measured in terms of maximum resident set size used (e.g., `ru_maxrss`) via `getrusage`.

loader under all benchmark programs.

In both `matrix` benchmark and `revmatrix` benchmark, user time of all-at-once loading is significantly better than any other loader with different page loading strategy. Demand loading and hybrid loading have similar user time. However, prefetching pages in hybrid loading still performs better than demand loading, even the difference is relative small. This observation meets the expectation as the memory is accessed in either row or column order, and the inner loop of memory access resembles arithmetic sequence access. In terms of memory usage, demand loading performs better than both all-at-once loading and hybrid loading.

When accessing memory in an array in both forward direction and reverse direction (e.g., `seq` and `revseq`), user time of all loaders perform relatively the same but all-at-once loading performs slightly better. In terms of system time, all-at-once loading performs significantly better than the rest. This is mainly due to that many `mmap` calls are invoked with small region length in demand loading. We find that demand loading and hybrid loading without any prefetching have relatively same performance. This is because segments except `.bss` section are usually small, which can be filled into serveral pages. Demand loading of these pages does not lead to visible overhead. In addition, prefetching more pages likely have better performance, which meets our heuristic expectation of sequential memory access. In terms of memory usage, demand loading uses less memory than all-at-once loading, again.

Based on these benchmarks, we conclude that all-at-once loading performs better than demand loading and hybrid loading in terms of runtime. However, demand loading has advantage in saving memory usage. Prefetching with more pages can have better performance in terms of runtime.

References

[1] "Executable and Linkable Format." `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`.

[2] "Eexcutable and Linkable Format (ELF)." `http://www.skyfree.org/linux/references/ELF_Format.pdf`.

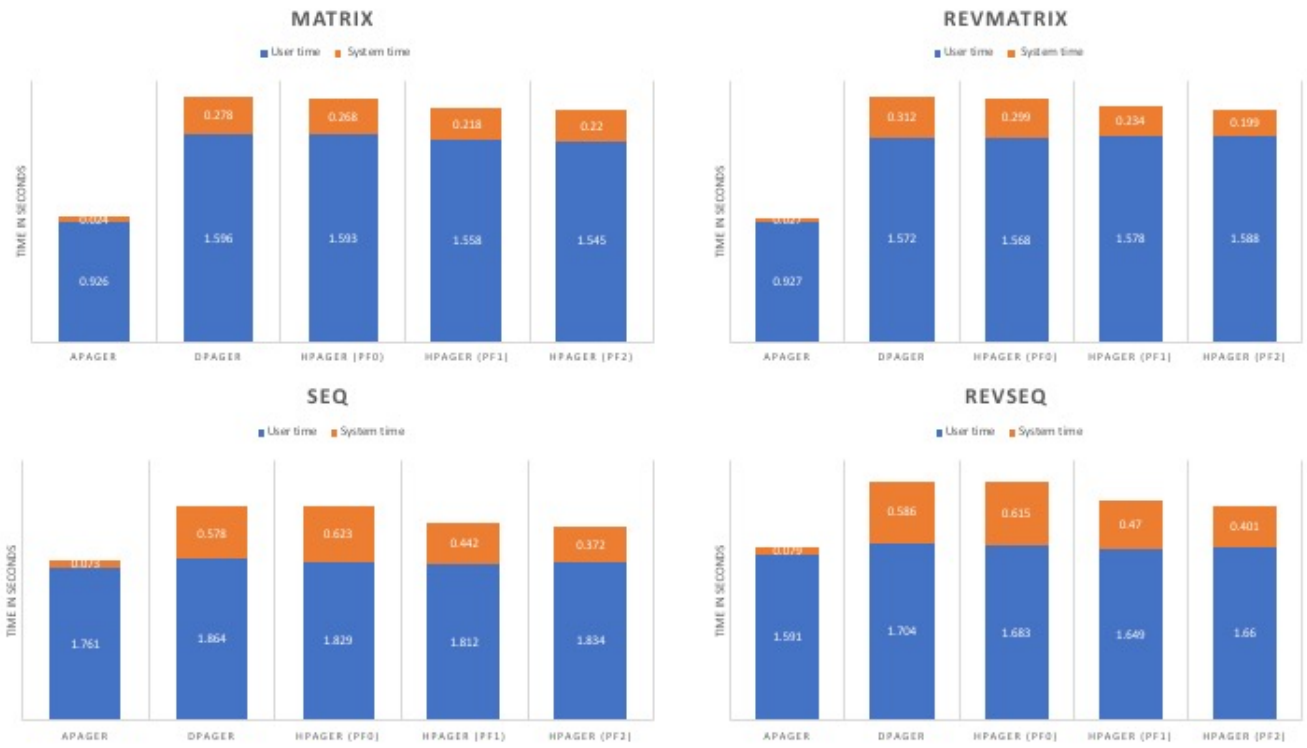

Figure 5: User time and system time for loaders on four benchmark programs. `apager`, `dpager`, and `hpager` denote all-at-once loading, demand loading, and hybrid loading, respectively. For hybrid loading, `pf0`, `pf1`, and `pf2` represent the number of additional page prefetching, which are 0, 1, 2 respectively. The results are averaged over 10 trails.

[3] "Program Header." http://www.sco.com/developers/gabi/latest/ch5.pheader.html.

[4] "Linux Standard Base Core Specification 3.1." http://refspecs.linuxbase.org/LSB_3.1.1/LSB-Core-generic/LSB-Core-generic.html#PROGHEADER, 2018.

[5] "Acronyms relevant to Executable and Linkable Format (ELF)." https://www.cs.stevens.edu/~jschauma/631/elf.html.

[6] "Linker and libraries guide." https://docs.oracle.com/cd/E19957-01/806-0641/6j9vuqujs/index.html#chapter6-71736.

[7] "How programs get run: Elf binaries." https://lwn.net/Articles/631631/.

[8] "personality(2) - linux man page." http://man7.org/linux/man-pages/man2/personality.2.html.

[9] ".bss." https://en.wikipedia.org/wiki/.bss.

[10] "System v application binary interface." https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.