CS380L: Lab 3

Dinesh Reddy Chinta

Mar 2023

1 Report

1.1 Linux execve

Read the Linux implementation of execve and describe in pseudo-code the steps that occur in creating the memory image of a new process.

Listing 1: execve pseudocode

```
void allocate_memory() {
    // Step 1: Allocate a new process ID
    pid_t pid = allocate_pid();

    // Step 2: Allocate a new memory space for the process
    struct mm_struct *mm = allocate_mm_struct();

    // Step 3: Initialize the memory map for the new process
    initialize_memory_map(mm);

    // Step 4: Set the process's mm pointer to the new mm struct
    current->mm = mm;

    // Step 5: Set the process ID for the new process
    current->pid = pid;

    // Step 6: Set the parent process ID for the new process
    current->ppid = parent_process_id();
}

int execve(const char *filename, char *const argv[], char *const envp[]) {

    // Step 1: Load the executable file into memory
    load_executable(filename);

    // Step 2: Allocate memory for the new process
    allocate_memory();

    // Step 3: Copy the program's code and data into the new memory space
    copy_code_and_data();

    // Step 4: Set up the stack for the new process,
    // including passing the command line arguments and environment variables.
    setup_stack(argv, envp);

    // Step 5: Set up the program counter to point to the entry point of the program
    set_program_counter();

    // Step 6: Clean up any resources used by the previous process
    cleanup_resources();

    // Step 7: Start the new process and transfer control to it
    start_process();

    // Step 8: If execve returns, there was an error
    return -1;
}
```

The pseudo-code above outlines the steps that occur when a new process is created in Linux using execve().

The initialize_memory_map() function reads the elf file and creates the virtual memory layout of a process. The elf parser involves:

1. elf header parsing: Every elf file starts with this elf header that describes essential higher level information like number of program headers, program header entry size, section headers, Architecture etc.

2. program header parsing: Based on the program headers information from the elf header, it then reads program headers to determine the start and end of each segments – text, data, bss, brk and other.

3. Initialize memory map strucuture based on the above start and end information of each of the segments.

Setting up the Stack involves, allocating space for Environment variables, Argument Vectors, Auxiliary Vectors, and pointers to environment and argument vectors.

Overall, the steps shown in the pseudo-code ensure that the new process has its own memory space and is able to execute the code from the executable file with the appropriate arguments and environment variables.

1.2 All-at-once Loading

Describe what functionality of your loader your test programs exercise.

The Apager (All-at-once loader) exercies static loading of the text, data and bss sections. All our programs are statically linked and the elf file has the complete map of the three segments. The loader memory maps the entire sections while loading the program itself.

What happens if the program under test calls malloc?

The malloc system-call allocates space in the shared heap of the loader. While we expect the dynamic memory to be allocated in the heap of the loaded program i.e., after the bss section, the dynamic memory is allocated in the heap of the loader program. This is because there is only one kernel data structure being shared by both the loader and the loaded progroam. Since our loader is invoked by the execve system call, it initialized the brk of the process to that of the loader. Hence, even if the loaded program is requesting for dynamic memory, the OS searches the mm struct of the loader and allocates the memory from the heap of the loader.

1.3 Demand Loading

Demonstrate at least one program that runs faster for the all-at-once loader, and one that uses less memory for the demand paging loader.

For a large sequential memory access workload, that we described in Listing 3., the Apager performas better than the demand paging. Since the workload anyways access all the pages, the dynamic loader harms the performance by waiting to map the memory.

Listing 2: Sequential Access of a Large Array

```
unsigned long memsize =  1024 * 1024 * 1024;

void mem_access()
{
    int page_size = getpagesize();
    for(unsigned long i = 0; i < memsize;){
        bss_mem[i] = 'a';
        i += page_size;
    }
}

int main(int argc, char **argv) {
    mem_access();
    return 0;
}
```

Listing 3: Small Access of a Large Array

```
unsigned long memsize =  1024 * 1024 * 1024;

void mem_access()
{
    int page_size = getpagesize();
    for(unsigned long i = 0; i < memsize;){
        bss_mem[i % page_size] = 'a';
        i += page_size;
    }
}

int main(int argc, char **argv) {
    mem_access();
    return 0;
}
```

If the working set of the program is a small portion of the bss section then demand loader is better for virtual memory usage. This would largely reduce the virtumal memory foot print of the since we only demand load one page in a total for Dpager (Demand Pager). For this workload, the apager took slightly over 1GB of virtual memory.

1.4 Memory Access Errors

What does your demand pager do for a program that de-references NULL? What should it do?

My demand pager raises Segmentation Fault for Null de-references. When a Null dereferance happens, the kernel invokes the Segfault handler of the Dpager. If we don't model our Segfault Handlers properly then we might change the program behaviour and unintentionally silence the memory referance errors of the program under test. We do not want to change the behaviour of the program and silincing the errors is a faulty behaviour of a loader.

Describe what your pager does to preserve memory access errors.

To preserve the memory access erros, my pager searches the memory mapping for the process. When we parse the elf file of the program, we keep track of the start and end address of each of the text, data, and bss sections. If the faulting address is in none of the above sections then it is a memory access error of the program and we do not want to mmap the faulting page. We simply do not serve the segfault and exit by calling exit(EXIT_FAILURE). This will preserve the original sigv context and error information.

1.5 Hybrid Loader

Describe the heuristic used to map the additional page(s).

I used a simple policy of mapping the next contiguous pages to the faulting addres as our heuristic when predicting the next access in the Hybrid Pagers. While the poilcy is imperfect, we can find workloads for which the policy is advantageous.

Find a workload that is faster with your heuristic and explain why.

The Sequential Access workload that we described in the section 1.3 can be used to observe the impact of enabling a memory access predictors and proactively speculating the memory access behaviour. For the workload, a simple "next-page" predictor is sufficient to correctly speculate the memory mapping. We discuss the slight improvement in performance in the Results section.

1.6 Experiments

Describe the environments and workloads (test programs) for evaluating your pagers.

I run the experiments on a Ubuntu 20.04.5 LTS and compiled the code with a gcc 9.4.0.

As described in Section 1.3, the first workload is a Sequential access of 1GB of a global array (bss section). This is very simple workload that whose behaviours is simple and predictable. However, simply accessing every byte of the momory would involve lot of interferance from caches. Hence, to avoid the impact of caches we access only the first byte of every page in the array.

The second experiment is to access only one page of the the 1GB global array. This is to expose the efficacy of the dpager to dynamically allocate virtual memory. Even though the array size is large, if the workload only access a small portion of the array then there is not additionaly advantage of mapping the entire address space during the initialization itself as it is done by the Apager.

For both of the workloads, we used All-at-once loader (Apager), Dynamic loader (Dpager), Hybrid loader (Hpager), Hybrid loader with allocation of one predicted page (Hpager_p1), and Hybrid loader with allocation of two predicted pages.

Pager	Seq: Time μ (σ) s	Seq: Vm (estimated)	Small: Time μ (σ) s	Small: Vm (estimated)
Apager	**0.62** (0.097)	1GB	0.701 (0.0358)	**1GB**
Dpager	**3.01** (0.040)	1GB	0.006 (0.0009)	**1MB**
Hpager	**2.95** (0.156)	1GB	0.006 (0.001)	1MB
Hpager_p1	**2.79** (0.063)	1GB	0.005 (0.0003)	1MB
Hpager_p2	**2.64** (0.096)	1GB	0.006 (0.0008)	1MB

Table 1: Comparision of Execution times (mean and std) and Virtual memory footprints for Sequential workload (Seq) and Small memory footprint workload loaded with each of the Apager, Dpager, Hpager, Hpager with one page and two page speculative mapping.

Each of the experiments is performed 5 times to nullify statistical abressions.

1.7 Experiment Results

From the table, we can note the the mean time of Apager for Sequential access workload is around 20% of the dpager. This is attributed to the context switches associated with each page fault in case of the dpager. Each of these context switches do not occur in the Apager since all the memory is already mapped and only the OS page faults while allocating physical memory. This is inline with our expectations.

The runtimes of the Hpager is also comparable to that of the Dpager. We can all notice that it decreases as we oppertunistically preallocate the pages. In case of the baseline Hpager – where we don't speculatively map the pages and only map the faulting page – the runtime is still slightly less than Apager. This is expected because the slight reduction is from the initial mappings of the text and data segments.

While I expected a large decrease in runtimes for the prediction enabled Hybrid pagers (Hpager_p1, Hpager_p2), since the number of segfault handler calls now reduce to 50% and 25% respectively, for Hpager_p1 and Hpager_p2 respectively, the runtime has only slightly decreased. Further investigation is required to determine why the runtime improvement is small.

Since the workload access all the pages of the array, the VM usage is estimated to be 1GB with all the loaders.

For the second workload, where the application only access one page of the entire array, the runtimes of the demand loaders is less than that of the Apager. This is because, in case of Apager, the kernel has to search a larger portion of the mm kernel data structure while mapping 1GB for the array. In case of the Demand pagers, the kernel only searches for the accessed page.

The estimated virtual memory usage of the second workalod is much lesser for the demand pagers than the Apager. This is because we dynamically map only the firt page leaving out all the unaccessed pages unmapped.

1.8 Dynamic Linking: LD_DEBUG and LD_PRELOAD

Describe what these environment variables do.

The LD_DEBUG environment variable is used to enable debug output from the dynamic linker/loader, which is responsible for loading shared libraries and resolving symbols. By setting the LD_DEBUG environment variable to a non-empty value, we can enable debug output for a given program.

Set the LD_DEBUG environment variable to a non-empty value by running the command export LD_DEBUG=libs.

The LD_DEBUG environment variable will cause the dynamic linker/loader to output debug information to standard error (stderr).

Why might LD_PRELOAD be useful?

The LD_PRELOAD environment variable is used to specify a list of additional shared libraries to be loaded before the standard set of shared libraries.

This allows you to override functions in standard shared libraries with your own implementations.

LD_PRELOAD can be useful for debugging, testing, or profiling purposes, as it allows you to modify the behavior of functions in shared libraries without modifying the source code of the program. However, it can also be used maliciously to inject code into a program, so it should be used with caution.

1.9 "Hello World" Symbol Lookups

How many symbol lookups were made to run "hello world" compiled with default options?

There are 186 symbol looks ups while running the simple "hello world" program that we compiled with default options.

Listing 4: Symbol Lookups for the simple "hello world" program

```
$ LD_DEBUG=all ./test_bin/hello 2&>1 | tee log/symbol.log
...
2252907:     symbol=_dl_catch_error;  lookup in file=./test_bin/hello [0]
2252907:     symbol=_dl_catch_error;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
2252907:     binding file /lib64/ld-linux-x86-64.so.2 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `
     _dl_catch_error' [GLIBC_PRIVATE]
...
$ cat log/symbol.log | grep "symbol=" | wc -l
186
```

What are the trade-offs of relying on dynamic libraries?

The trade-offs of relying on dynamic libraries include:

1. Increased complexity: Dynamic linking adds an extra layer of complexity to the software development process, as developers need to ensure that all necessary shared libraries are available at runtime.
2. Performance overhead: Loading and linking shared libraries at runtime can cause a small amount of overhead, which may be noticeable in certain performance-critical applications.
3. Versioning issues: Dynamic linking can lead to versioning issues if multiple versions of a shared library are installed on a system. This can lead to conflicts and unexpected behavior.
4. Flexibility: On the other hand, dynamic linking provides greater flexibility and modularity in software development, as it allows libraries to be updated and replaced without requiring recompilation of the entire program. Additionally, dynamic linking can lead to smaller executable sizes, as shared libraries can be reused across multiple programs.

1.10 Malloc Hook Library

Describe dlsym, RTLD_NEXT and LD_PRELOAD.

dlsym() is a function in the C standard library that allows a program to obtain the address of a symbol in a shared object at runtime. This can be useful for programs that need to dynamically load and use shared libraries. The syntax of dlsym() is as follows:

Listing 5: dlsym prototype

```
void *dlsym(void *handle, const char *symbol);
```

1. The first argument, handle, is a handle to the shared object that contains the symbol.
2. The second argument, symbol, is a string containing the name of the symbol to look up.
3. The return value is a pointer to the symbol's address.

RTLD_NEXT is a flag that can be passed to dlsym() to look up a symbol in the next shared object in the search order, rather than the one specified by handle. This can be useful for intercepting function calls in a shared library, as it allows a program to replace a function in a shared library with its own implementation.

LD_PRELOAD environment variable allows a program to specify a list of shared libraries to be loaded before all other libraries. This can be useful for intercepting function calls in a program or for replacing functions in a shared library with a custom implementation. When a program is started with LD_PRELOAD set, the specified libraries are loaded into the program's address space before all other libraries, and their symbols are used in preference to the symbols in the other libraries.

Together, these features allow a program to intercept and modify the behavior of shared libraries at runtime, which can be useful for debugging, testing, or adding new functionality to a program without modifying its source code. However, care must be taken when using these features, as they can introduce subtle bugs or security vulnerabilities if not used correctly.

Describe how your shared library works at a high level.

This library defines a new malloc function that first loads the original malloc function using dlsym() and then calls it with the same arguments as the original malloc function. Before calling the original malloc function, the new malloc function prints the malloc arguments to stdout.

1.11 Time Spent

It took me over 80 hours to complete the lab.

1.12 Referances

Please find the code at my git repo git@github.com:chintadinesh/adv_os-lab3.git.