Lab 3: Program loading and memory mapping

Ankit Goyal
ankit@cs.utexas.edu
CS380L

October 27, 2014

1 Setup

1.1 Hardware

Host Processor: 64 bit 4 core Intel(R) Xeon(R) CPU E3-1270 V2 @ 3.50GHz
Host Memory: 16GB
HyperThreading: Yes
Logical CPUs after Hyperthreading: 8
CPU frequency scaling: Disabled in BIOS (turned off Intel SpeedStep and C-states)

1.2 Software

Host Operating System: Ubuntu with 3.13.0-34-generic 64 bit kernel.

2 Creating the memory image of a new process

`sys_execve` is responsible for setting up the environment for running the program. Below are the steps taken by `sys_execve` which calls `do_execve_common`:

1. Check that `NPROC` limit is not exceeded (i.e., total number of process), if it is then exit. (L: 1443)
2. Allocate memory for data structure in kernel. (L: 1458)
3. Open the `exec` file using `do_open_exec` (L: 1469)
4. Now the kernel data structures are initialized and `exec_binprm` is called.
5. `exec_binprm` calls `search_binary_handler` which finds the binary format handler, in our case elf. So it finds `load_elf_binary`. (fs/binfmt_elf.c L:84 & 571)
    - `load_elf_binary` does consistency checks by making sure that it's an ELF format file by comparing the main number and ELF in `e_ident` field in header.
    - `load_elf_binary` reads the header information and looks for `PT_INTERP` segment to see if an interpreter was specified. This segment is only present for dynamically linked programs and not for statically linked.
    - Now all the loadable segments are `mmapped` into memory, by reading the ELF Program headers. `bss` segment is also mapped.
    - `create_elf_tables` creates a stack at a random offset and sets the auxiliary vectors, arguments and environments according to the standard.
    - Finally the control is transferred to `e_entry` point using `start_thread` method.(fs/binfmt_elf.c L:990)

3 Implemented Loaders

3.1 All at Once Loader

In this all the pages were mapped while loading the test program including `bss`. **On calling malloc**, memory is allocated by kernel and not the loader program. The process executes successfully.

3.2 On-demand Loader

In this loader, at startup only the first page is mapped. A `segfault handler` is registered and when the program tries to access an unmapped page, it segfaults and the `segfault handler` maps the corresponding address from file. Performance is discussed later

3.3 Hybrid Loader

In this loader, at startup all text and initialized data is mapped. The `bss` memory is mapped on demand. **If a program tries to access an invalid memory address**, the `segmentation fault` is propagated and is thrown as a user would expect. Hybrid loader also uses the following prediction algorithm, where on every successful prediction a counter is increased and it allows us to map pages in advance.

```
int correctPredictions = 0;

void pagePrediction(unsigned long currentAddress){
        if (last_prediction_was_correct)
                correctPredictions++;
        else
                correctPredictions--;

        if (correctPredictions == 1){
                map next 1 page
        } else if (correctPredictions > 1) {
                map next 2 pages
        }
        predict the next page to be 1 page after the last mapped page
}
```

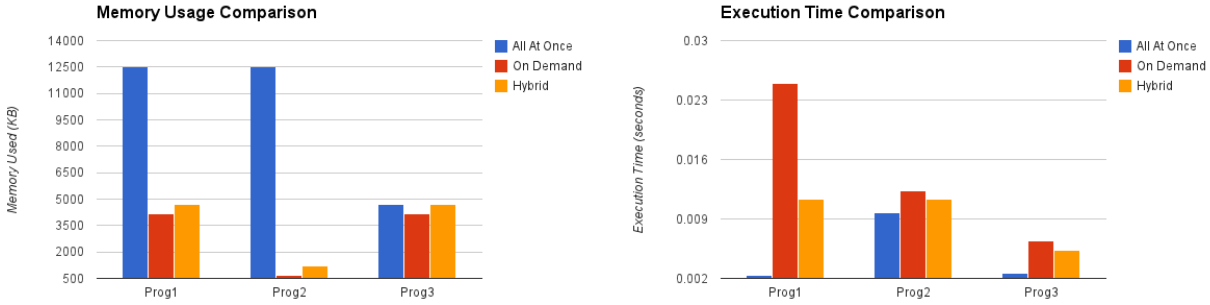
Codeblock 1: Prediction Algorithm

4 Test Programs and Results

Following section first describes the program and then the results of running each loader on that program. All the results in this section refer to Fig. 1

4.1 `prog1`

`prog1` contains 3 uninitialized global variables (large bss segment). It accesses only one of the arrays. This shows the effect of loading all the pages in memory rather than getting them on demand or through some heuristic. This program makes printf call to make sure that it runs correctly.



(a) (b)

Figure 1: Memory Usage and Execution Time comparison for three loaders

1. **All at once** uses a large amount of memory when compared to other two since it maps everything on startup. The execution time of all at once is less because the memory is already mapped and the program doesn't segfault as in other two cases.
2. **On Demand** uses the least amount of memory since the memory is mapped only when needed. Since the program seagulls more often and it has an overhead associated with it. It can be seen that the execution time of On-demand is greater than the other two due to this overhead.
3. **Hybrid** maps only the text and data segment while loading. Since the array is accessed sequentially it is able to predict quite well and is using almost the same amount of memory as used by demand paging. Due to the decrease in overhead due to segfaults, it's execution time is better than on-demand loader.

```
int a[1000000];
int b[1000000];
int c[1000000];

int main() {
        int i;
        for (i=0; i<1000000; i++) {
                a[i] = 12;
        }
        printf("hello!\n");
 }
```

Codeblock 2: prog1

4.2 prog2

prog2 contains 3 uninitialized global variables (large bss segment). It accesses only one of the arrays but in a random order and accesses only a part of it. This shows the variation in performance of hybrid pages.

1. **All at once**: Again all-at-once uses a large amount of memory when compared to other two since it maps everything on startup. The execution time of all at once is less because the memory is already mapped and the program doesn't segfault as in other two cases.

2. **On Demand** uses the least amount of memory since the memory is mapped only when needed. Randomness doesn't affect on-demand since it only maps pages when they are needed.

3. **Hybrid** maps only the text and data segment while loading. Since the array is accessed randomly it not is able to predict as effectively as the `prog1` given the nature of the algorithm.

```
int a[1000000];
int b[1000000];
int c[1000000];

int main() {
        int xRan;
        srand(time(0));
        int i;
        for (i=0; i<1000000/2; i++) {
                xRan = rand()%100000;
                a[xRan] = 12;
        }
 }
```

Codeblock 3: prog2

4.3 prog3

`prog3` contains a simple array that loops through it. The memory usage of each method should be equivalent in this case.

1. **All at once** Since almost all the `bss` memory is accessed, at the end all three loaders map all the pages. All-at-once uses the same amount of memory and it's execution is again better due to the same reasons.

2. **On Demand** On-demand has a high overhead and since we are mapping the whole memory it has the worse execution time.

3. **Hybrid** has a higher overhead than all at once but less than on-demand. This is reflected in the execution time and again it uses the same amount of memory.

```
static int a[1000000];
int main() {
        for (i=0; i<1000000; i++) {
                        a[i] = 12;
          }
}
```

Codeblock 4: prog3

4.4 prog4

`prog4` is the code given on course page. This code throws a segmentation fault when run. All three loaders throw a `segmentation fault`

When demand or hybrid pager gets `segfault`, they check if the given faulting address belongs to a address in the loaded file. If not, they `raise` the signal `SIGSEGV` again. Since it's a double fault, the program aborts throwing a segmentation fault to the user.

```
int
main() {
   int *zero = NULL;
   return *zero;
}
```

Codeblock 5: prog4

Time Spent on the lab ≈ 30 hours

5 References

1. http://eli.thegreenplace.net/2012/08/13/how-statically-linked-programs-run-on-linux/
2. http://www.skyfree.org/linux/references/ELF_Format.pdf
3. http://linux.die.net/man/5/elf
4. http://articles.manugarg.com/aboutelfauxiliaryvectors.html
5. http://pubs.opengroup.org/onlinepubs/009695399/functions/sigaction.html
6. http://stackoverflow.com/questions/8116648/why-is-the-elf-entry-point-0x8048000-not-changeable
7. http://lxr.free-electrons.com/source/fs/exec.c#L1425

6 Actual Numbers in the Graph

6.1 Execution Time

	Prog1	Prog2	Prog3
All At Once	0.002344	0.009721	0.002592
On Demand	0.024904	0.012285	0.006405
Hybrid	0.011381	0.011351	0.005276

6.2 Memory Usage

	Prog1	Prog2	Prog3
All At Once	12506	12509	4693
On Demand	4176	652	4172
Hybrid	4704	1192	4696