Part A:

What is the vDSO? What does it stand for? What is its goal.

A:

vDSO - virtual dynamic shared object (virtual dynamically linked shared objects)

Shared library that the kernel maps into the address space of all user-space applications

System calls that are used frequently, like gettimeofday, made available to process memory space since it's not a harmful function. This gets rid of the extra overhead

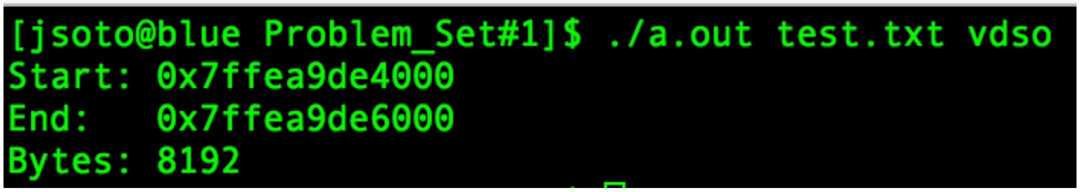
Part B:

Read this article, written for kernel developers:

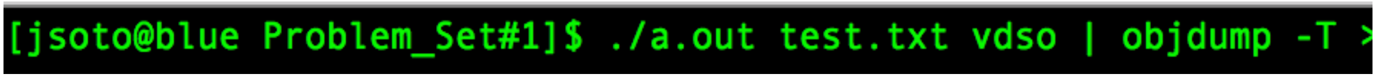
> Matt Davis, "Creating a vDSO: the Colonel's Other Chicken," *Linux Journal*, 2012, https://www.linuxjournal.com/content/creating-vdso-colonels-other-chicken.

Use Listing 1 from the article to build a program that dumps the vdso segment from memory on blue.cs.sonoma.edu. Then, use `objdump` to output the dynamic symbol table from this segment of memory. Put each these in your write-up: the command you used to run the program, its output from stdout, the command you used run objdump, and its output from stdout.

A:



```
[jsoto@blue Problem_Set#1]$ ./a.out test.txt vdso
Start: 0x7ffea9de4000
End:   0x7ffea9de6000
Bytes: 8192
```



```
[jsoto@blue Problem_Set#1]$ ./a.out test.txt vdso | objdump -T >
```

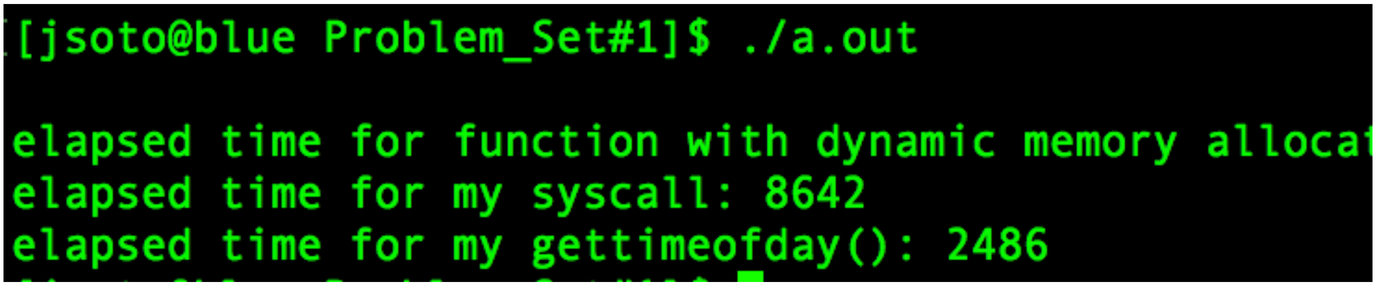
Objdump output:

```
a.out:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:

0000000000000000      DF *UND*       0000000000000000  GLIBC_2.2.5 free

0000000000000000      DF *UND*       0000000000000000  GLIBC_2.2.5 abort
```

```
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 fread
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 fclose
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 strchr
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 printf
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 __libc_start_main
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 fgets
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 strtoll
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 fprintf
0000000000000000  w D *UND*           0000000000000000              __gmon_start__
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 malloc
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 fseek
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 fopen
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 perror
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 fwrite
0000000000000000      DF *UND*        0000000000000000  GLIBC_2.2.5 strstr
00000000006020a0 g    DO .bss 0000000000000008  GLIBC_2.2.5 stderr
```

Part C:

Write a program, p4.cpp, based on p3.cpp but adding a call to `gettimeofday()` whose time cost measured like the other calls. The gettimeofday() call is implemented using the vDSO. Put the program output in your report and write a sentence that qualitatively compares the cost of gettimeofday() to the other calls your program measures.



```
[jsoto@blue Problem_Set#1]$ ./a.out

elapsed time for function with dynamic memory allocat
elapsed time for my syscall: 8642
elapsed time for my gettimeofday(): 2486
```

A:

The output from the p4.cpp indicates that the gettimeofday() function approximately 800 ms longer than a dynamic memory allocation for an integer. However it is fast than a syscall, getpid(), by 6200 ms. The dynamic memory allocation, is allocating space in the heap of size integer, while the syscall is going to kernel for further instructions, however the gettimeofday speed still lies between the two function calls. This is most likely due to the gettimeofday function being implemented using vDSO, thus being slower than the integer dynamic memory allocation, but faster than the syscall getpid().