CSE 4600 Homework 6 Process Logical Address Space

Highlights of this homework assignment:

---

1. Preamble

Every Linux process has its own dedicated logical address space. The kernel maps these "virtual" addresses in the process address space to actual physical memory addresses as necessary. In other words, the data stored at address `0xDEADBEEF` in one process is not necessarily the same as the data stored at the same address in another process. When an attempt is made to access the data, the kernel translates the virtual address into an appropriate physical address in order to retrieve the data from memory... we are not really concerned with physical addresses. Rather, we are investigating exactly how the operating system allocates more virtual addresses to a process.

---

2. User Versus Kernel Space

A process may operate in user mode or kernel mode. The programs that we have been programming actually switch between these two modes. When a process changes mode, it is termed a **context switch**.

To summarize these two modes:

1. User Mode
    - application processes.
    - These are executed in an isolated environment so that multiple processes cannot interfere with each other's resources.
2. Kernel Mode
    - when the kernel acts on behalf of the process.
    - This would be when a system call is made, or when an exception or interrupt occurs.
    - Processes running in kernel mode are privileged and have access to key system data structures.

When a program is loaded as a process, it is allocated a section of virtual memory which is known as **user space**.

By contrast, there is another section of memory which is known as the **kernel space**. This is where the kernel executes and provides its services.

The remainder of these notes focus on user space.

---

3. Three Segments of User Space

The user context of a process is made up of the portions of the address space that are accessible to the process while it is running in user mode.

There are a few definitions that come in handy when we are talking about memory:

- global variable — one declared outside of any function
- local static variable — one declared inside a function with the *static* keyword. Such a variable keeps its value across function calls. For instance, if it has the value 57 after you return from the function call, when that function is called again, it will still be 57.
- local automatic variable — any variable declared inside a function without the *static* keyword
- heap—is dynamically allocated space.
    - In C, you use *malloc* or *calloc*
    - In C++, you use *new*

The portions of address space are: *text*, *data*, and *stack*. They are described in further detail below:

1. Text
    - sometimes called the instruction or code segment
    - contains executable program code and constant data
    - is read only
    - multiple processes can share this segment if a second copy of the program is to be executed concurrently

2. Data
    - contiguous (in a virtual sense) with the text segment
    - subdivided in to three sections:
        i. initialized data (static or global variables)
        ii. uninitialized data (static or global variables)
        iii. heap (dynamically allocated)
    - addresses increase as the heap grows

3. Stack
    - used for function call information including:
        - address to return to
        - the values or addresses of all parameters
        - local automatic variables
    - addresses decrease as the stack grows

Notice that the stack grows towards the uninitialized data and the heap grows towards the stack.

Some compilers and linkers call uninitialized data and heap **bss** (Block Started by Symbol—not bs) for historical reasons.

Memory references to Text, Data and Stack in a user space program are done with virtual addresses. The kernel translates these virtual addresses to physical memory addresses.

In working with these virtual addresses, you have access to three external variables:

- *etext*—first valid address above the text segment
- *edata*—first valid address above initialized data segment
- *end*—first valid address above the uninitialized data segment

The following program, in *Interprocess Communications in UNIX: The Nooks & Crannies*, provides an example of displaying these external variables. The program also displays the address of some key identifiers to verify that the identifiers (variables) are put into the correct segments.

```
// Example from Interprocess Communications in UNIX: The Nooks & Crannies.
// memoryexample.c

#include <stdio.h>
#include <stdlib.h>   //needed for exit()
```

```
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define SHOW_ADDRESS(ID, I) printf("The id %s \t is at:%8X\n", ID, &I)

extern int etext, edata, end;

char *cptr = "Hello World\n"; // static by placement
char buffer1[25];

int main(void)
{
    void showit(char *); // function prototype
    int i=0; //automatic variable, display segment adr


    printf("Adr etext: %8X\t Adr edata: %8X\t Adr end: %8X\n\n",
            &etext, &edata, &end);

    // display some addresses
    SHOW_ADDRESS("main", main);
    SHOW_ADDRESS("showit", showit);
    SHOW_ADDRESS("cptr", cptr);
    SHOW_ADDRESS("buffer1", buffer1);
    SHOW_ADDRESS("i", i);
    strcpy(buffer1, "A demonstration\n");   // library function
    write(1, buffer1, strlen(buffer1) + 1); // system call
    for (; i<1; ++i)
        showit(cptr); /* function call */

    return 0;
}


void showit(char *p)
{
    char *buffer2;
    SHOW_ADDRESS("buffer2", buffer2);
    buffer2 = (char *)malloc(strlen(p)+1);
    if (buffer2 != NULL)
    {
        strcpy(buffer2, p);    // copy the string
        printf("%s", buffer2); // display the string
        free(buffer2);         // release location
    }
    else
    {
        printf("Allocation error.\n");
        exit(1);
    }
}
```

A sample run on a Linux machine produced the following output:

```
Adr etext:   4013D5      Adr edata:   404068     Adr end:   4040A0

The id main      is at:  401196
The id showit    is at:  4012B3
The id cptr      is at:  404060
The id buffer1   is at:  404080
The id i         is at:1FF8134C
A demonstration
The id buffer2   is at:1FF81328
Hello World
```

4. Homework Exercise

The purpose of this assignment is to explore the memory associated with the user process: text, data, and stack.

1. Copy and paste the following source code:

```
// memory_segments.c
#include <stdio.h>
#include <stdlib.h>
```

```
int g0;       /* global variable, uninitialized */
int g1 = 14;  /* global variable, initialized */
int g2[1500]; /* global variable, uninitialized */
int g3 = 16;  /* global variable, initialized */
int g4;       /* global variable, uninitialized */

void proc1();
void proc2();

int main()
{
        extern int etext[], edata[], end[];

        int lc0;      /* local variable, stored on stack */
        int lc1 = 27; /* local variable, stored on stack; mix init and uninit */
        int lc2;      /* local variable, stored on stack */
        static int ls0; /* local static variable, uninitialized data */
        static int ls1 = 19;      /* local static variable, initialized data */
        int *pheap1;
        int *pheap2;

        pheap1 = new int[10];
        pheap2 = new int[100];

        printf("\n\n---------- main ------------------\n\n");
        printf("%14p (%15lu): Last address\n",
                0xfffffffffff, 999999999999999);

        printf("%14p (%15lu): Address etext\n",
                etext, etext);
        printf("%14p (%15lu): Address edata\n",
                edata, edata);
        printf("%14p (%15lu): Address end\n",
                end, end);

        printf("%14p (%15lu): Address of code for proc1\n",
                &proc1, &proc1);
        printf("%14p (%15lu): Address of code for proc2\n",
                &proc2, &proc2);

        printf("%14p (%15lu): Address of uninitialized global variable g0\n",
                &g0, &g0);
        printf("%14p (%15lu): Address of initialized   global variable g1\n",
                &g1, &g1);
        printf("%14p (%15lu): Address of uninitialized global array    g2\n",
                &g2[0], &g2[0]);
        printf("%14p (%15lu): Address of initialized   global variable g3\n",
                &g3, &g3);
        printf("%14p (%15lu): Address of uninitialized global variable g4\n",
                &g4, &g4);

        printf("%14p (%15lu): Address heap1 in heap space\n",
                pheap1, (unsigned long) pheap1);
        printf("%14p (%15lu): Address heap2 in heap space\n",
                pheap2, (unsigned long) pheap2);

        printf("%14p (%15lu): Address of local variable lc0\n",
                &lc0, &lc0);
        printf("%14p (%15lu): Address of local variable lc1\n",
                &lc1, &lc1);
        printf("%14p (%15lu): Address of local variable lc2\n",
                &lc2, &lc2);

        printf("%14p (%15lu): Address of local uninitialized static var ls0\n",
                &ls0, &ls0);
        printf("%14p (%15lu): Address of local initialized   static var ls1\n",
                &ls1, &ls1);

        proc1();
        proc2();

        return 0;
}

void proc1() {
        int lc3;
        int lc4 = 37;

        printf("\n\n----------- proc1 -----------------\n\n");
        printf("%14p (%15lu): Address of code for proc1\n",
                &proc1, &proc1);
```

```
        printf("%14p (%15lu): Address of global variable g0\n",
                &g0, &g0);
        printf("%14p (%15lu): Address of global variable g1\n",
                &g1, &g1);
        printf("%14p (%15lu): Address of global variable g2\n",
                &g2[0], &g2[0]);
        printf("%14p (%15lu): Address of global variable g3\n",
                &g3, &g3);
        printf("%14p (%15lu): Address of global variable g4\n",
                &g4, &g4);
        printf("%14p (%15lu): Address of local variable lc3\n",
                &lc3, &lc3);
        printf("%14p (%15lu): Address of local variable lc4\n",
                &lc4, &lc4);
}

void proc2() {
        int lc5;
        int lc6 = 51;
        static int ls2;
        static int ls3 = 47;

        printf("\n\n------------ proc2 -----------------\n\n");
        printf("%14p (%15lu): Address of code for proc2\n",
                &proc2, &proc2);
        printf("%14p (%15lu): Address of global variable g0\n",
                &g0, &g0);
        printf("%14p (%15lu): Address of global variable g1\n",
                &g1, &g1);
        printf("%14p (%15lu): Address of global variable g2\n",
                &g2[0], &g2[0]);
        printf("%14p (%15lu): Address of global variable g3\n",
                &g3, &g3);
        printf("%14p (%15lu): Address of global variable g4\n",
                &g4, &g4);
        printf("%14p (%15lu): Address of local variable lc5\n",
                &lc5, &lc5);
        printf("%14p (%15lu): Address of local variable lc6\n",
                &lc6, &lc6);
        printf("%14p (%15lu): Address of local uninitialized static var ls2\n",
                &ls2, &ls2);
        printf("%14p (%15lu): Address of local initialized   static var ls3\n",
                &ls3, &ls3);
}
```

2. Compile and run the program.

   Several addresses are output from the program including:

   - etext
   - edata
   - end
   - functions
   - global variables
   - static local variables
   - heap variables

   You may want to use a redirect (./memory_segments > output.txt) to save the output for later.

3. Your task is to fill in addresses of the variables, and functions, including:
   - instructions in machine code
   - global initialized variables
   - static initialized variables
   - global uninitialized variables
   - static uninitialized variables
   - dynamically allocated variables
   - local automatic variables from main
   - local automatic variables from each of proc1 and proc2

   Fill in the following template:

```
// Place the following picture as a comment at the beginning of
// your memory_segments.c file.

// Add the significant addresses to diagram to the right of the boxes.
// Add text, edata, end, and the names of the variables to right as well.


        +------------------+
        |  TEXT            |
        |                  |
        |  instructions in | x10b10 = proc1
        |  machine code    |        = proc2
        |                  |
        +------------------+        = etext
        |  DATA            |        = g1   global initialized
        |  - initialized   |        = g3
        |    variables     |        = ls1  static initialized
        |                  |        = ls3
        |                  |
        | ---------------  |        = edata
        |                  |        = g0   global uninitialized
        |                  |        = g2   big array
        |                  |        = g4
        |  - uninitialized |        = ls0  static uninitialized
        |    variables     |        = ls2
        |                  |
        |                  |
        | ---------------  |        = end
        |  - heap for      | x22670 pointed to by heap1
        |  dynamic         | x22680 pointed to by heap2
        |  allocation      |
        |                  |
        |                  |
        +------------------+
                 |
                 |
                 V

               ......

                 ^
                 |
                 |
        +------------------+
        |  STACK           |          = lc4, lc6   | Activation record for
        |  - function      |          = lc3, lc5   | proc1 or proc2
        |    activation    |
        |    records       |
        |  - local         |           = lc2       | Activation record for
        |    automatic     |           = lc1       | main
        |    variables     | xffbe2880 = lc0       |
        +------------------+
```

4. Before continuing, run the program again and confirm that the output is the same.

5. As variables are added to the stack, do the addresses get smaller or larger?

6. Do variables stored on the stack ever have the same address as other variables? Why or why not?

7. Where would you expect variables (or arguments) in recursive functions to be stored (stack, heap, or other data segment)? When you finish step 8 below, comment on whether your expectations were correct or not.

8. Test your expectation by adding a recursive factorial function. For instance, the factorial of 5 is represented by *5!* and is calculated as *5*4*3*2*1=120*.

   In the factorial function, you will print the address of the factorial function and the address of the argument passed to it.

   In main, you will prompt the user for what factorial they will want to calculate and send that input as an argument to the function. In main, you will also print the value returned from the factorial function.

   The idea behind the recursive factorial function is the following:

$$F_n = \begin{cases} 1 & \text{if } n = 0 \textbf{ (base case)} \\ n*F_{n-1} & \text{if } n >= 1 \textbf{ (recursive step)} \end{cases}$$

Deliverables:

Submit 2 files to Canvas, `memory_segments.c` and `hw6log.txt`

1. Answers to questions 5, 6, and 7 (as comments at the top of your code file `memory_segments.c`)
2. The memory diagram (as comments immediately below the answers in your code file `memory_segments.c`)
3. The new recursive factorial function is added in `memory_segments.c`
4. The script log file `hw6log.txt` contains two sample runs of the recursive factorial function showing:
    - *10!*
    - *6!*
