OBL1-OS

August 23, 2023

This is a mandatory assignment. Use resources from the course to answer the following questions. **Take care to follow the numbering structure of the assignment in your submission**. Some questions may require a little bit of web searching. Some questions require you to have access to a Linux machine. Working in groups is **permitted**, but submissions to Blackboard must be **individual**. **Submissions in PDF are preferred.**

1 The process abstraction

1. Briefly describe what happens when a process is started from a program on disk. A mode switch from kernel- to user-mode must happen. Explain why this is necessary.

   Solution:

   (a) A process control block (PCB) is allocated and initialised

   (b) An address space is allocated in physical memory

   (c) The program instructions are copied into memory

   (d) The hardware context (program counter) is set to the first instruction in the program (text) segment

   (e) A switch from kernel- to user-mode is necessary to start running the new process on the CPU

2. Download the latest Linux kernel source code from `https://kernel.org` and unpack it. Use a web search engine to help identify the file in the source tree that contains the process descriptor structure (hint: its name is `task_struct`). List the field name from this structure that:

   Solution: The file is `include/linux/shed.h`

   (a) Stores the process ID

   Solution: `pid`

   (b) Keeps track of accumulated virtual memory

Solution: `acct_vm_mem1`

Use the Linux command-line tool `top` to explore other fields relating to running processes. Can you match them to field names in the process descriptor `task_struct`? Name two such fields (besides those listed above).

Solution: Many examples exist. Here are two:

- `top: PR, task_struct: prio`
- `top: TIME+, task_struct: prev_cputime, cputime_expires`

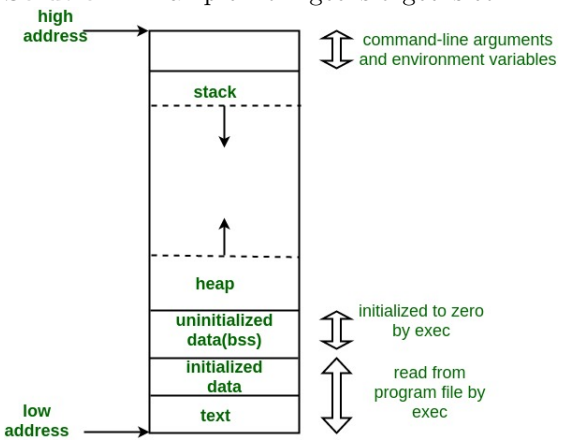
2 Process memory and segments

The memory region allocated to a process contains the following segments.

- Text segment
- Data segment
- Stack
- Heap

1. Sketch the organisation of a process' address space. Start with high addresses at the top, and the lowest address (0x0) at the bottom.

Solution: Example from geeksforgeeks.com:



See also `http://alumni.cs.ucr.edu/~saha/stuff/memaddr.html`

2. Briefly describe the purpose of each segment. Why is address 0x0 unavailable to the process?

Solution: The `text` or *program* segment contains the executable program code. This consists of a listing of compiled instructions with an entry point (`main()`, or more correctly `_start()`). The size of the text segment is roughly equal to the size of the binary executable file that is stored on the disk (roughly, because this executable also contains format headers and other information which is removed when the program is loaded into memory).

The *data* and *bss* segments contain global and static variables. Variables in the data segment are initialised (e.g., `int j=0;`. Bss contains unitialised variables which are set to zero when the progarm starts.

The *heap* is a data structure used by the process for dynamic memory allocation. To use this memory region the program must explicitly call `malloc()` to allocate memory and `free()` to release it when it is done using it. `malloc()` and `free()` are simply wrapper functions to the system calls `brk()` and `sbrk()`.

The *stack* is a data structure that allows function calls as well as space to store local variables used by a function call. Each function call causes the stack to grow, and when the function returns the stack shrinks so that it returns to the state it was before the function call.

Memory in the region above the stack can be used for storing command line arguments for the program and environment variables. It is also common for a segment in the process space to be reserved for the kernel, which allows faster system calls. This is not illustrated in the graphic above, and this memory is inaccessible to the process in user space.

The address 0x0 is reserved for the *null pointer*, which can be used by programs to indicate when a pointer does not point to a valid memory address.

3. What are the differences between a *global*, *static*, and *local* variable?

Solution:

- Global variables are visible across all .c files (and by extension .o object files) in a program being compiled and linked. This means if you declare two global variables with the same name and in different .c files, the linker will complain because it will not know which variable should be used. They are stored in the data segment.
- Global static variables are visible to all functions within a specific .c file, so unlike above, you can declare multiple global static variables with the same name in multiple .c files and the program will compile. Local static variables are used in functions when you want to remember the value of a variable between successive function calls. Unlike local variables (below) which are stored on the stack (and thus are forgotten when the function returns), local static variables are stored in the data segment, which allows them to retain their value between function calls.
- A local variable is only visible to the function in which it is declared, and is stored on the stack.

Given the following code snippet, show which segment each of the variables (var1, var2, var3) belong to.

```
#include <stdio.h>
#include <stdlib.h>

int var1 = 0;
void main()
```

```
{
    int var2 = 1;
    int *var3 = (int *)malloc(sizeof(int)); // Note, since we are using malloc(), var3 will be a
                                            // pointer into the heap!
                                            // So the question is, where is the pointer stored?
    *var3 = 2;
    printf("Address: %x; Value: %d\n", &var1, var1);
    printf("Address: %x; Value: %d\n", &var2, var2);
    printf("Address: %x; Address: %x; Value: %d\n", &var3, var3, *var3);
}
```

Solution: var1 resides in the data segment because it is a global variable. var2 resides on the stack because it is a local variable to function `main()`. var3 is a pointer which resides on the stack because it is local to `main()`, however, the data it points to resides on the heap because we are calling `malloc()` to dynamnically allocate some memory.

3 Program code

1. Compile the example given above using `gcc mem.c -o mem`. Determine the sizes of the **text, data, and bss** segments using the command-line tool `size`.

   Solution: Note: The following assumes Intel x64 architecture. Other architectures may differ in output.

   ```
   $ size mem
      text           data            bss            dec            hex        filename
      1838            616              8           2462            99e        mem
   ```

2. Find the start address of the program using `objdump -f mem`.

   Solution: Note: The following assumes Intel x64 architecture. Other architectures will differ in assembly output.

   ```
   $ objdump -f mem

   mem:     file format elf64-x86-64
   architecture: i386:x86-64, flags 0x00000150:
   HAS_SYMS, DYNAMIC, D_PAGED
   start address 0x00000000000005f0
   ```

   The start address is 0x5f0 (hexadecimal).

3. Disassemble the compiled program using `objdump -d mem`. Capture the output and find the name of the function at the start address. Do a web search to find out what this function does, and why it is useful.

Solution: We can pipe the output of `objdump` to `grep`, which searches for a string (here, the start address) in the output:

```
$ objdump -d mem | grep 5f0
00000000000005f0 <_start>:
 5f0:        31 ed                           xor    %ebp,%ebp
```

`_start()` is a wrapper function for `main()` which is defined by the C runtime environment. It does some simple setup, calls `main()`, and then invokes the `exit()` system call when `main()` returns.

4. Run the program several times (hint: running a program from the current directory is done using the syntax `./mem`). The addresses change between consecutive runs. Why?

Solution: By default on many recent operating system kernels (Linux, BSD, macOS, Windows), address space layout randomisation (ASLR) is turned on. This is a security feature which makes it difficult for malicious code to exploit vulnerabilities in user applications by randomising the layout of a process' memory space such that it is harder to predict where the text segment, stack, heap, and shared libraries are located. For more information see `https://en.wikipedia.org/wiki/ASLR`.

4 The stack

Consider the following C program:

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

void func()
{
    char b = 'b';
    /*long localvar = 2;
    printf("func() with localvar @ 0x%08x\n", &localvar);
    printf("func() frame address @ 0x%08x\n", __builtin_frame_address(0));
    localvar++;*/
    b = 'a';
    func();
}

int main()
{
    printf("main() frame address @ 0x%08x\n", __builtin_frame_address(0));
    func();
    exit(0);
}
```

1. Compile the example given above using `gcc stackoverflow.c -o stackoverflow`.
2. Determine the default size of the stack for your Linux system. Hint: use the `ulimit` command (a web search or running the command `ulimit --help` will help find the appropriate command-line flags).

Solution:

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 61598
max locked memory       (kbytes, -l) 16384
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 61598
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

The default stack size (per process) is 8 MiB (8192 KiB, or 8 x 1024 x 1024 bytes).

3. Run the program. Describe your observations and find the cause of the error.

Solution: Note: The following assumes Intel x64 architecture. Other architectures will differ in output.

```
$ ./stackoverflow
main() frame address @ 0x13fb9c30
func() with localvar @ 0x13fb9c14
func() frame address @ 0x13fb9c20
...
func() frame address @ 0xf5458760
func() with localvar @ 0xf5458734
func() frame address @ 0xf5458740
Segmentation fault (core dumped)
```

There was a segmentation fault, due to the fact that we ran out of stack space by recursively calling `func()`.

4. Run the program and pipe the output to `grep` and `wc -l`:

   `./stackoverflow | grep func | wc -l`

   What does this number tell you about the stack? How does this relate to the default stack size you found using the `ulimit` command?

Solution:

```
$ ./stackoverflow | grep func | wc -l
523324
```

The number of times we find the text `func` (using the `grep` command) in the output before the program crashes is 523324 (the exact result will depend on your system). If we divide this by two (func is printed twice per call) we get the number of times the function was called recursively. Since we know that every function call requires allocating some space on the stack for local variables, CPU registers, and a return address, we can calculate the number of bytes used by each funcion call in the next question.

5. How much stack memory (in bytes) does each recursive function call occupy?

Solution: For this we can subtract the memory frame address (or the address of `localvar`) from one recursive call to the next (since the stack growns downwards in memory). In the listing above, we subtract the second-last frame address 0xf5458760 from the last frame address 0xf5458740, which yields 0x20, or 32 in decimal. Thus, each recursive call to `func()` occupies 32 bytes on the stack. We can check this another way: we can divide our maximum default stack space 8192 KiB by the number of recursive calls (523324/2), giving us roughly 32 (the remainder can be explained by the stack space used by C runtimes and `main()`).

We can calculate this without running the program. Each stack location on a 64 bit system is 8 bytes. We see from the source code that func() has one local variable. Each local variable occupies 16 bytes on the stack, regardless of the size of the variable (this is called *stack alignment*). For each function call, the base pointer (RBP) is pushed onto the stack, occupying 8 bytes. Finally, the return address (used to point to where we continue execution when the function returns) occupies 8 bytes. Thus the total stack memory used for each function call is $16 + 8 + 8 = 32$ bytes.