OBL1-OS

1 The process abstraction

1.

When a program is started from disk:

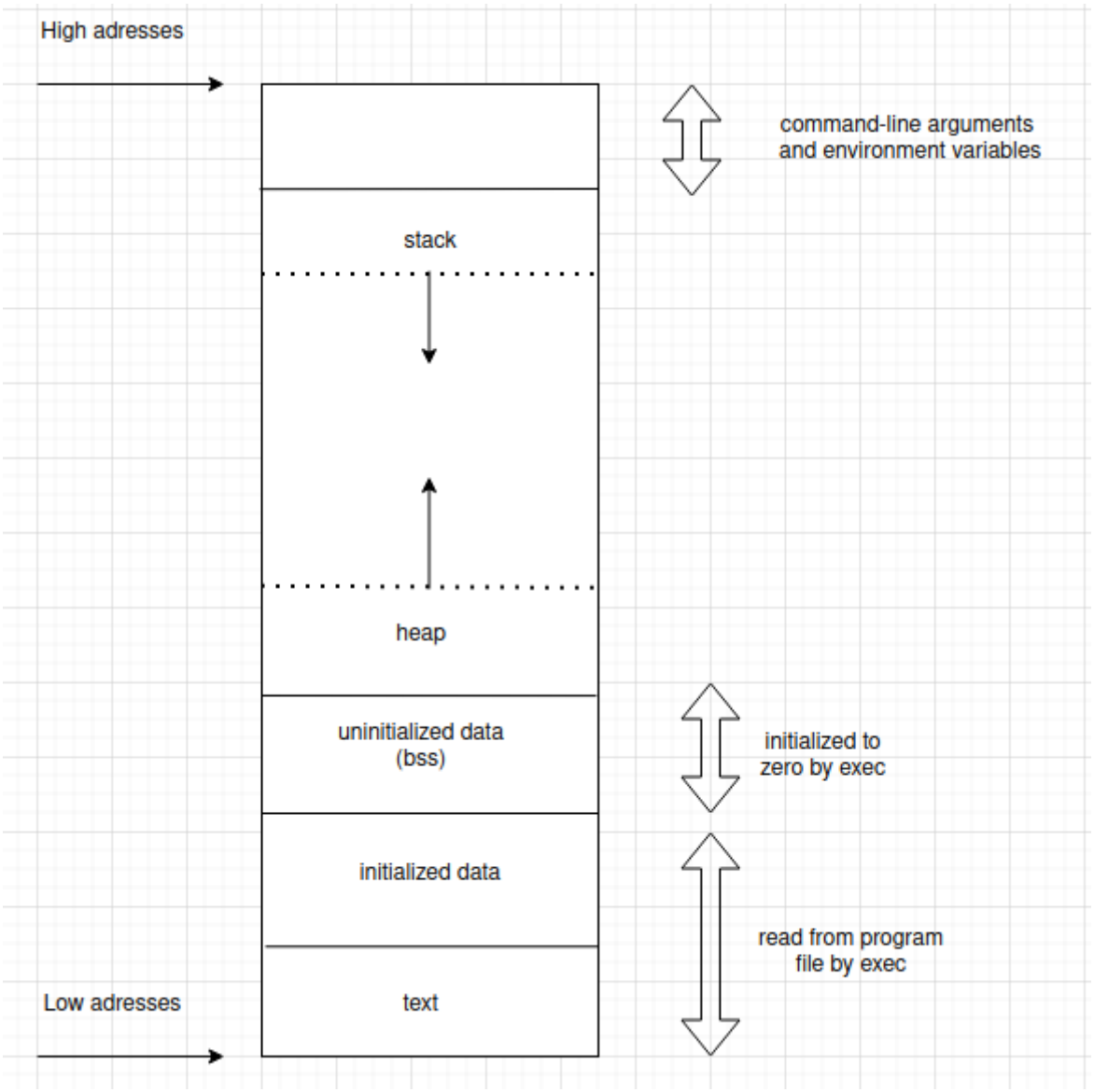
1. A process control block is allocated and initialized.
2. An address space is allocated in physical memory.
3. The program instructions are copied into memory.
4. The hardware context is set to the first instruction in the program segment.
5. A switch from kernel-mode to user-mode is necessary to start running the new process on the CPU.

2. (a) The field that stores the process ID is: pid_t pid.
(b) The field that points to the structure witch keeps track of accumulated virtual memory is: struct mm_struct *mm. The specific field within mm_struct that tracks accumulated virtual memory is called total_vm.

In top you can find the field PR that corresponds to prio in the task_struct. You can also find the field TIME+, that corresponds to the fields prev_cputime and cputime_expires in the task_struct.

2 Process memory and segments

1.



2. The text segment holds the executable program code, comprising a series of compiled instructions and an entry point, denoted as main(). The size of this segment is approximately the same as the binary executable file stored on the disk.
Both the data and uninitialized data segments store global and static variables. The data segment contains initialized variables, while the Bss segment houses uninitialized variables. Upon program initiation, these uninitialized variables in the Bss segment are set to zero.

The heap is a data structure that facilitates dynamic memory allocation for the process. To utilize this region, the program must explicitly invoke malloc() for memory allocation and free() to release the allocated memory. Both malloc() and free() serve as wrappers for the system calls brk() and sbrk() respectively.

The stack is another data structure, designed to manage function calls and provide storage for local variables within those calls. A function call leads to the expansion of the stack, and upon the function's conclusion, the stack contracts, reverting to its pre-call state.

Memory regions above the stack can store command line arguments and environment variables for the program. Additionally, a segment within the process space is often reserved for the kernel,

enabling quicker system calls. This segment isn't depicted in the aforementioned graphic and remains inaccessible to user-space processes.

The address 0x0 is earmarked for the null pointer. Programs use this to signify when a pointer doesn't reference a valid memory address.

3.

- Global variables are accessible throughout all .c files, and consequently, .o object files in a program during compilation and linking. If you declare two global variables with identical names in different .c files, the linker will raise an error.
- Global static variables are confined to a specific .c file, making them accessible to all functions within that file. This distinction means you can declare multiple global static variables with the same name across various .c files without any compilation issues. On the other hand, local static variables are employed within functions to maintain their value across consecutive function calls. Unlike local variables, which are stored on the stack and lose their value once the function concludes, local static variables are kept in the data segment, ensuring value retention between calls.
- Local variables are exclusive to the function they're declared in and are stored on the stack.

Var1 is stored in the data segment as it's a global variable. var2 is located on the stack since it's a local variable within the main() function. While var3 is a pointer that also sits on the stack due to its local scope in main(), the data it references is on the heap, as it's dynamically allocated using malloc().

3 Program code

1. $ size OBL1_OPG3

```
  text    data     bss     dec     hex filename
  1798     616       8    2422     976 OBL1_OPG3
```

2. $ objdump -f OBL1_OPG3

```
OBL1_OPG3:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000010a0
```

3. $ objdump -d OBL1_OPG3 | grep 10a0

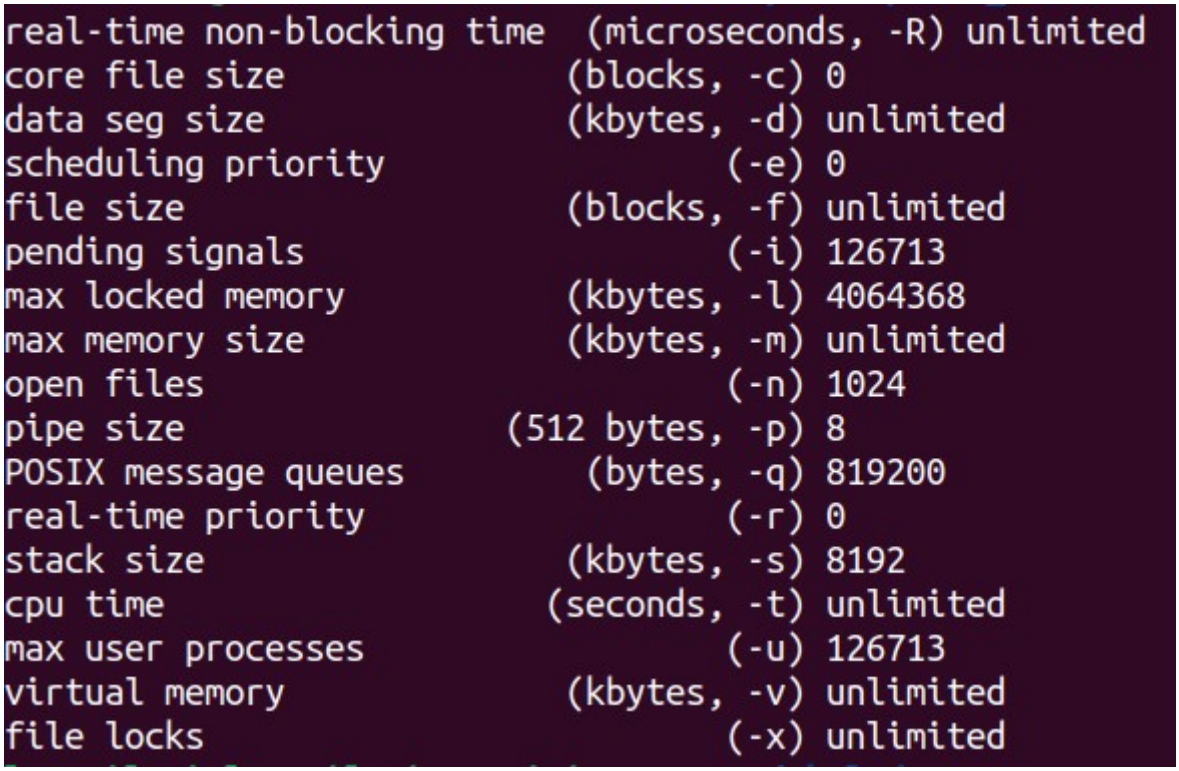
```
00000000000010a0 <_start>:
    10a0:       f3 0f 1e fa                 endbr64
```

_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. On many modern operating systems, address space layout randomization is enabled by default. This security measure shuffles the arrangement of a process's memory space, making it challenging for malicious code to exploit vulnerabilities in user applications. By randomizing the positions of the text segment, stack, heap, and shared libraries, it becomes less predictable for potential attackers.

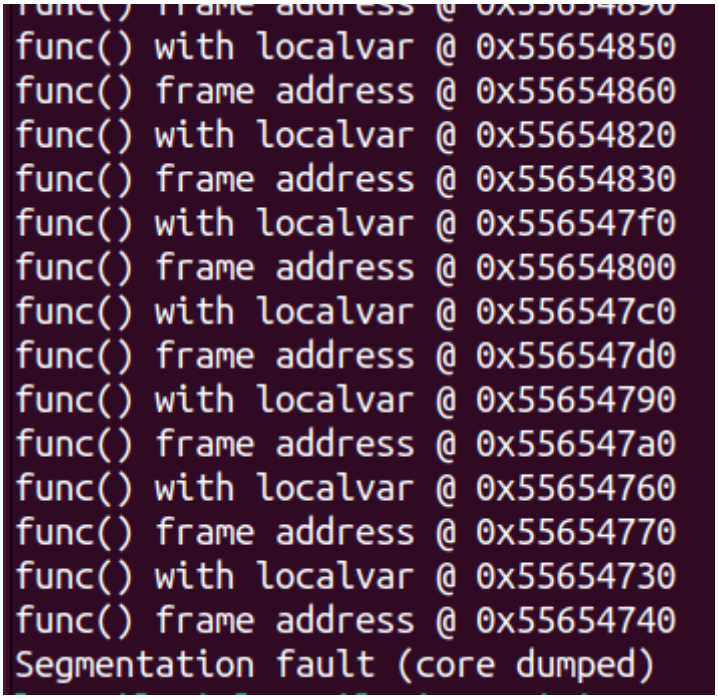
4 The stack

2. $ ulimit -a



The default stack size is 8192 KiB which is equivalent to 8 MiB

3. $ ./stackoverflow



There was a segmentation fault, because we ran out of stack space by calling func() recursively.

4. $ ./stackoverflow | grep func | wc -l

The number of times we find the text func using the grep command in the output before the program crashes is 349123. Since func is printed twice per call we divide this by two, to 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 function call.

5.
We can find out how much stack memory each recursive function call occupies by dividing our maximum default stack space 8192 KiB by the number of recursive calls (349123/2), giving us roughly 47 bytes.