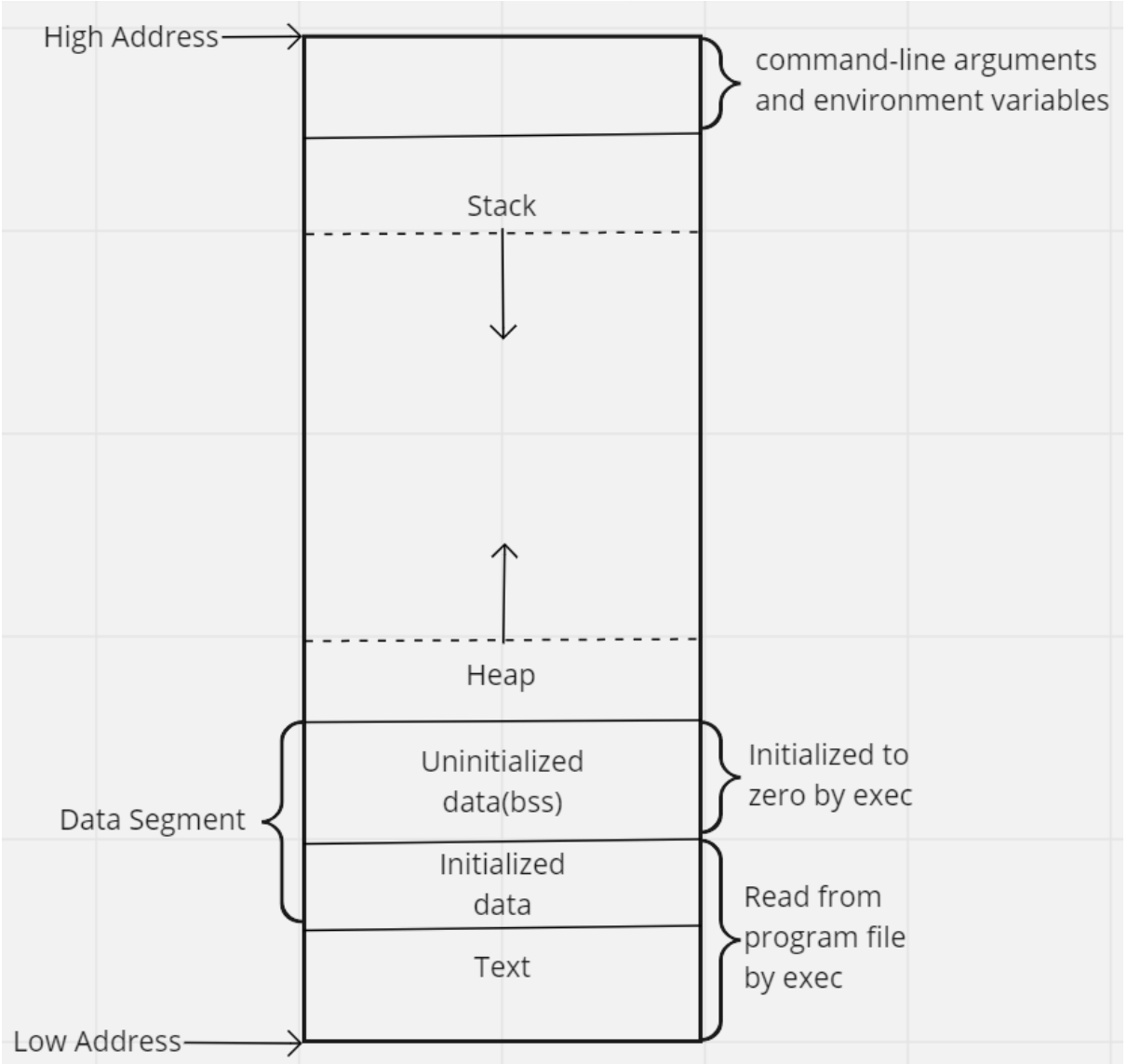
OBL1-OS

1 The process abstraction

1. When a process is started from a program on a disk, an instance of the program is copied into a location in memory as instructions and data. The operating system sets aside different locations for the stack, heap, text and data. To start the program the operating system can instruct the CPU to execute the first instruction in the processes allocated memory.
   After this the CPU has to be set to operating in user mode. The mode is signified as a 0 flag in a status/EFLAG register inside the CPU.
   When running a process from a user provided program, user mode is used to protect the operating system from malicious code or bugs. Without this protection, bugs could cause damage to the operating system or other programs. In user mode each instruction is checked, and only allowed if it falls under the privileges provided by the kernel. If the program needs to execute instructions outside of its privilege, e.g. writing to memory outside its range, a switch to kernel mode is required.

2. In linux-5.19.5/include/linux/sched.h contains the definition for the task_struct. Task_struct is the universal structure that processes and threads are implemented through. The process ID is stored in the pid_t field, which stands for thread group ID.
   ```
   pid_t                   pid;
   ```
   The accumulated virtual memory is stored in the acct_vm_mem1 field.
   ```
   u64                     acct_vm_mem1;
   ```
   The priority is stored in the prio, static_prio, normal_prio and rt_priority fields.
   ```
   int                     prio;
   int                     static_prio;
   int                     normal_prio;
   unsigned int            rt_priority;
   ```
   The TIME+ is calculated from values stored in the utime and stime fields.
   ```
   u64                     utime;
   u64                     stime;
   ```

2 Process memory and segments

1. 

2. **Text segment**: Contains executable instructions. It may be placed below the heap or stack in order to avoid heaps and stack overflows from overwriting it. The text segment is usually sharable so that only a single copy needs to be in memory for frequently executed programs. and it's also often read-only, to prevent accidental modification of its instructions by a program.
   Initialized Data segment: Usually called "The data segment", is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

   Uninitialized Data segment: Often called the "bss" (block started by symbol) segment. Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing. Uninitialized data starts at the end of the data segment and contains all global and static variables that are initialized to 0 or do not have explicit initialization.

Stack: The stack is a space in memory set aside by the operating system when the program is run. The stack is often called the execution stack and contains the program's procedures and their local variables. The stack is navigated by the operating system using a stack pointer.

Heap: The heap is the part of a program's memory used for dynamic memory allocation. It starts at the top of the bss segment and grows towards the stack. Once the stack and the heap meet, there is no more free memory. The heap can be utilized by programs with malloc, realloc and free.

The address 0x0 is in C known as the null pointer. It cannot be dereferenced or used because it is needed for important operations. These can include initializing pointers, representing unknown conditions and indicating errors when returning an unusable pointer from a function(if memory is empty, or out of range etc.).

3. A local variable is a variable that is initialized inside a function that can only be used in that same function. A global variable is initialized outside functions and can be used in any function. A static variable is initialized with the static type and is only initialized once. If a static variable is initialized inside a function, it will only be initialized the first time, and not change value back to its initial value when the function runs multiple times.
   Local variables are stored in the stack. Global and static variables are stored in data. They are stored in the initialized data if they have been assigned a value (int i = 10;) or in the uninitialized data segment if they have only been declared (static int j;).

   var1 is a global variable stored in initialized data.
   var 2 is a local variable stored stack.
   var3 is a pointer stored in the stack, pointing to the heap.
   We observe that var3 and var2 are located towards the top of the stack, while var 1 and the content of var3 are stored towards the bottom of the heap

3 Program code

1. Sizes using size CLI:
   text: 2082 bytes
   data: 624 bytes
   bss: 8 bytes
2. Start address using objdump: 0x00000000000010a0:
3. The function at the start address is **_start**. _start is the entry point of the program, and it's the address jumped to on program start. When _start has fulfilled its function such as loading command line arguments into argv, main() is called.
4. Each time a new process with the program starts the operating system assigns a place in memory using Address Space Layout Randomization (ASLR). The addresses in the heap retain three nibbles between runs, the addresses in the stack retain one nibble. This is because the address remains the same in relation to the rest of the program. Without ASLR the address for the program would be the same each run, resulting in the addresses for the variables being the same each run.

4 The stack

1. Done
2. Stack size (found with ulimit -s and ulimit -a): 8192 kB
3. The error message at the end of the program is “segmentation fault”. This means that the program is trying to write to a part of memory that it is not allowed to. This is because the function func() calls itself, creating an infinite loop and causing the program to run out of space, eventually causing the address for the next function to be outside the bounds for the program.
4. This number is the lines printed by the program. Each time func() is run, 2 lines are printed meaning that the number 523444 means that func() has been run 261722 times. This is the amount of times the function can run before the stack size of 8192 kB is depleted.
5. The function uses approximately 32 bytes each attempt, because the stack size is 8192000 bytes and 8192000 / 261800 = 31.3. The main() function also uses some stack space but becomes negligible compared to the 261800 times func() is run.