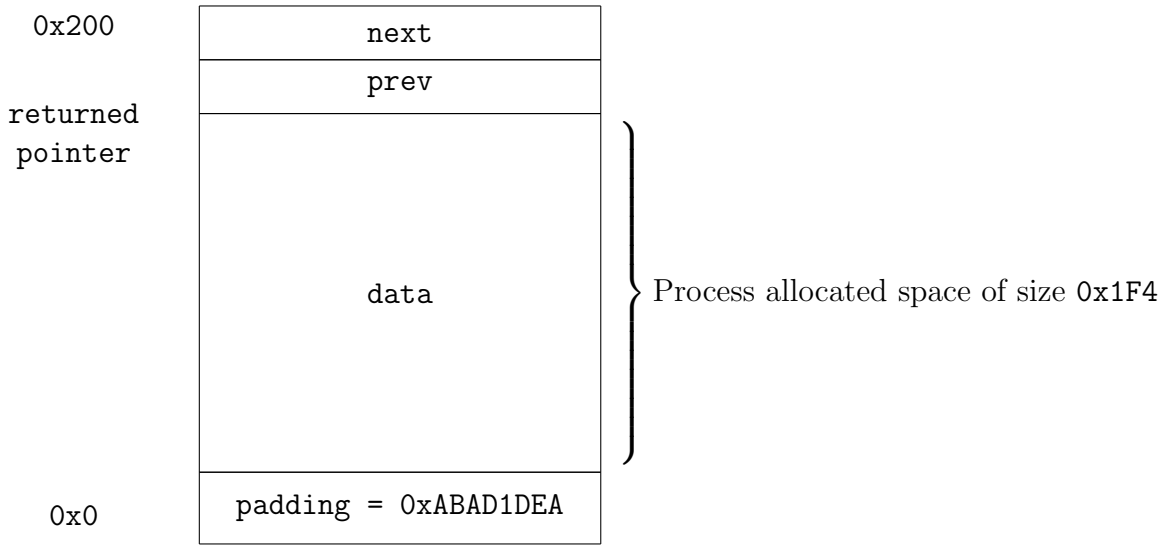
1 Memory

1.1 `void* request_memory_block()`

This method takes zero arguments and returns an address to a piece of memory that the caller can use. The returned address points to a continuous block of memory of size `0x1F4` bytes that the current process can read and write to. Currently, the memory protection unit is disabled so if the process writes outside of the data region indicated in Figure 1.1, then undefined behaviour will occur.

Figure 1: Memory Block



If the system is out of memory, the calling process will be pre-empted and be placed on its priority's memory blocked queue until more memory becomes available (i.e. when another process releases their memory). It is recommended that once the user is finished with their allocated block that they release it otherwise the system will have memory leaks and run out of memory very quickly. See Section 1.2 for more information.

1.2 `int32_t release_memory_block(void*)`

This method takes in a pointer returned by `request_memory_block` and deallocate it. If the pointer is invalid such as NULL or out of the range of the RAM, this method returns -1. This method also checks if there are processes blocked on memory with a priority greater than or equal to the calling process. If so, then the current process will get pre-empted and the scheduler will get called instead of continuing with the current process.

1.3 Example Usage of Requesting and Releasing Memory

```
void example_process(void) {
    int* array;

    while (1) {
        array = (int*) request_memory_block();
        array[0] = 0xDEADC0DE;
        release_memory_block(array);
    }
}
```