 # Introduction

The smart_loader loads and executes ELF binary files, handling segmentation faults, memory management, and internal fragmentation calculation.

 # Code Overview

1. Global Variables

- `ehdr`: A pointer to an `Elf32_Ehdr` structure representing the ELF header.
- `phdr`: A pointer to an `Elf32_Phdr` structure representing the program header.
- `fd`: A file descriptor for the ELF file being loaded.
- `n_segmentation_faults`: A counter for tracking segmentation faults.
- `n_page_allocations`: A counter for tracking page allocations.
- `total_internal_fragmentation`: A variable for tracking the total internal fragmentation.
- `new_start`: A function pointer for the entry point of the ELF binary.
- `termination_flag`: A flag to indicate when the process has finished executing.
- `head`: A pointer to the head of a linked list used for cleanup.

2. Data Structures

- `struct node`: A structure representing a node in a linked list used for cleanup. It contains information about memory allocations and ELF program headers.

3. Functions

a. `void loader_cleanup()`

- Purpose: This function cleans up allocated resources when the program exits. It deallocates memory and closes the ELF file.
- Implementation: It iterates through the linked list of allocated memory and program headers, deallocates the memory using `munmap`, and frees the associated data structures.

b. `void load_and_run_elf(char **exe)`

- Purpose: This function loads and runs the ELF binary specified in the `exe` parameter. It also handles segmentation faults and calculates internal fragmentation.
- Implementation: It opens the ELF file, reads the ELF header, initializes the entry point, and

calls the `new_start` function. It uses `handleSegmentationFault` to handle segmentation faults.

c. `void handleSegmentationFault(void *segmentation_address)`

- Purpose: This function is responsible for handling segmentation faults. It checks which segment caused the fault and allocates memory for it.
- Implementation: It iterates through the program headers, checks if the faulting address falls within a segment, allocates memory, and updates internal fragmentation calculations.

d. `static void my_handler(int signum, siginfo_t *info, void *context)`

- Purpose: This is a signal handler that handles segmentation faults (SIGSEGV). It increments the segmentation fault counter and calls `handleSegmentationFault` to handle the fault.

e. `int main(int argc, char **argv)`

- Purpose: The main function of the program. It sets up the signal handler, checks for correct command-line arguments, loads and runs the ELF binary, and performs cleanup when the program finishes executing.

Implementation:

- - Our code is an implementation of an ELF (Executable and Linkable Format)simpleSmartloader and runner.
- - It opens the specified ELF file, reads the ELF header, and initializes the entry point.
- - A signal handler is set up to handle segmentation faults (SIGSEGV).
- - Upon a segmentation fault, the `handleSegmentationFault` function is called.
- - This function iterates through the program headers, checks if the faulting address falls within a segment, and allocates memory page by page for the segment.
- - A linked list of nodes is used to track allocated memory, and a new node is created if the segment is encountered for the first time.
- - Allocated memory is used to load the segment from the ELF file.
- - The total number of page faults is counted.
- - The `loader_cleanup` function is responsible for deallocating the allocated memory and

closing the ELF file.

● - The program's execution results, such as the number of page faults and the size of allocated memory, are printed.

● - The main function handles command-line arguments, sets up the signal handler, loads and runs the ELF binary, and performs cleanup upon program termination.

● - The code loads ELF binaries page by page to ensure that virtual memory for intra-segment space remains contiguous, and multiple page faults are handled as segments are loaded progressively.

Error handling:

● - Error Handling: Error handling is implemented throughout the code using assertions and error messages.

● - Resource Cleanup: A linked list is used to keep track of allocated memory, allowing for resource cleanup upon program termination.

● - Signal Handling: The code uses a signal handler to handle segmentation faults (SIGSEGV) gracefully.

 # Future Improvements

- Improved Error Handling: The code could benefit from more comprehensive error handling, such as handling errors when allocating memory and reading from files.
- Documentation: In addition to comments, comprehensive documentation could be added to explain the purpose of functions and data structures.
- Portability: The code is currently designed for 32-bit ELF files. Extending it to handle 64-bit ELF files could be a future improvement.
- Security: Additional security measures could be implemented, such as validating input files and avoiding potential vulnerabilities.