**Project 4: Pager – A Virtual Memory Manager**

Atuhaire Ambala and Ricardo Escarcega

Prof. Isac Artzi

CST 315

June 9th 2024

**Project 4: Pager – A Virtual Memory Manager**

The Unix/Linux Command Line Interpreter represents a refined shell program designed to execute user commands easily in both interactive and batch modes. This shell accepts user input, executes a wide array of commands, and handles the concurrent execution of multiple commands separated by semicolons. This foundational project sets the stage for a full-functional, full-featured shell by focusing on the essential aspects of basic command execution and signal handling. Enhancements have been introduced to manage both physical and virtual memory effectively, with 1 GB of physical memory divided into fixed-size frames and 4 GB of virtual memory providing greater abstraction and flexibility. The project includes sophisticated memory allocation for processes, standardized page sizes, and meticulous page table management to track frame numbers, validity, and modification status. Frame table management and a first-fit allocation algorithm ensure efficient memory utilization. Page swapping is optimized using the Least Recently Used (LRU) algorithm and demand paging, loading pages only when accessed. Robust mechanisms handle page faults and segmentation faults, ensuring smooth execution and resource cleanup on process termination. These enhancements collectively contribute to a more robust and feature-rich command-line interpreter, capable of efficiently managing memory and resources, handling complex command execution, and providing a solid foundation for further development.

**Responsibilities**

Ricardo

* Documentation
* Implementation algorithm design
* Code review

Atu

* Coding
* Documentation review
* Implementation Research

**Methodology / Approach**

Enhancements have been introduced to manage both physical and virtual memory effectively. The methodology for these enhancements covers several key areas:

1. **Memory Allocation**: The shell will include a robust memory allocation system. This system ensures that memory is efficiently assigned to processes, dividing a set amount of physical memory into fixed-size frames. Each frame is managed to optimize utilization and performance.
2. **Virtual Addresses**: By leveraging virtual memory, the shell provides greater abstraction and flexibility. Virtual addresses allow the system to manage memory more dynamically and efficiently, enabling processes to use more memory than physically available by swapping data in and out of physical memory.
3. **Page Tables**: The project incorporates detailed page table management. Each process has its own page table that tracks the mapping of virtual addresses to physical frames. The page table entries include information on frame numbers, validity, and modification status. This structure ensures that the system can quickly resolve virtual addresses and maintain the integrity of data.
4. **Page Frames**: The physical memory is divided into fixed-size page frames, which serve as the smallest unit of memory allocation. The frame table keeps track of each frame's status, including whether it is free or allocated to a process. Efficient frame management is achieved using a first-fit allocation algorithm, ensuring that memory is utilized without fragmentation.

**Assumptions**

Several assumptions were made to streamline the design and enhance the functionality of the Unix/Linux Command Line Interpreter. First, we assumed a physical memory size of 1 GB, which is divided into fixed-size frames of 4 KB each. This assumption allows for efficient memory management and simplifies the implementation of the frame table, ensuring that memory allocation and deallocation are handled swiftly. Secondly, a virtual memory size of 4 GB was assumed, providing ample space for processes to utilize, and facilitating the abstraction of physical memory limitations. This larger virtual memory space supports the implementation of virtual addresses, enabling processes to access more memory than physically available, which is essential for effective demand paging and swapping. Additionally, it was assumed that each process requires a fixed memory size of 10 MB, which standardizes the calculation of page requirements and simplifies page table management. This assumption helps in planning and allocating the required pages, ensuring that each process receives adequate memory without excessive fragmentation. Finally, the assumption of a standard page size of 4 KB aligns with common practices in many operating systems, facilitating compatibility and efficient use of memory. These assumptions collectively support the algorithm's ability to manage memory efficiently, handle page faults and segmentation faults effectively, and maintain the overall functionality and robustness of the shell, ultimately leading to a more efficient and reliable command-line interpreter.

**Virtual Memory Manager Features**

The size of physical memory, defined as 1 GB, is divided into fixed-size frames. Each frame is a specific number of bytes, and we calculate the total number of frames available by dividing the total physical memory size by the frame size.

The virtual memory is larger, set to 4 GB, providing more flexibility and abstraction over the physical memory. Each process may require a different amount of memory; in our example, we assume each process needs 10 MB. This value is used to determine the number of pages required by the process.

The size of each virtual page is standardized to 4 KB, a common size in many systems. To calculate the number of pages a process needs, we use a formula that ensures all memory is covered by pages. For example, a process needing 10 MB will have its memory divided into the necessary number of 4 KB pages.

Page tables are used to manage these pages. Each page table contains entries that store information about each page, such as the frame number, whether the page is valid, and if it has been modified. These tables are initialized with the required number of entries.

For managing page frames, we use a structure that keeps track of each frame's status. This structure records whether a frame is free and, if it is allocated, which process and page it belongs to. The frame table is initialized by marking all frames as free. When a process needs a frame, the system searches for the first free frame and allocates it.

We use the First-Fit Allocation algorithm to find and allocate frames efficiently. This algorithm searches through the frame table to find the first available free frame and assigns it to the process.

When it comes to swapping pages, we employ the Least Recently Used (LRU) algorithm. This algorithm helps decide which page to replace by keeping track of the last time each frame was used. The system updates the usage time whenever a frame is accessed and identifies the least recently used frame for replacement when needed.

In handling page faults and segmentation faults, pages are loaded only when accessed, which causes a page fault. The system checks if the page has been modified and handles the fault by either allocating a new frame or swapping out an old page using the LRU algorithm. It loads the required page from the executable file and writes modified pages to swap space.

When a process terminates, all its resources, including dynamically allocated memory and open file descriptors, are cleaned up. The system deallocates frames, frees the page table, and ensures all resources held by the process are properly released, preventing any resource leaks.

**Paging Algorithm**

The paging algorithm in the Unix/Linux Command Line Interpreter is designed to manage memory efficiently and ensure seamless execution of user commands. The process begins with initializing the frame table and page tables for all processes. Each process's memory needs are calculated to determine the number of pages required, and frames are allocated using the first-fit allocation algorithm.

When a process accesses a page, the system checks if the page is already in memory. If the page is in memory, it is accessed, and its usage time is updated in the Least Recently Used (LRU) list. If the page is not in memory, a page fault occurs. The system then checks if there are free frames available. If there are, a frame is allocated to the page, and the page is loaded from the executable file. If no free frames are available, the system uses the LRU algorithm to find the least recently used frame for replacement.

If the frame to be replaced contains a modified page, the system writes the page to swap space before replacing it with the new page. The new page is then loaded from the executable file into the allocated frame. The page table and LRU list are updated to reflect the new mapping and usage time.

Throughout the process, the system ensures that each page's validity and modification status are accurately tracked in the page table. When a process terminates, the system deallocates all frames assigned to it, frees its page table entries, and releases any other resources held by the process, preventing resource leaks and maintaining system stability.

A screenshot of a computer

Description automatically generated

**Segmentation faults and page faults**

A segmentation fault occurs when a process tries to access a memory location that it's not allowed to access. This typically happens due to dereferencing a null or invalid pointer, accessing memory that hasn't been allocated, or writing to a read-only segment.

A diagram of a process

Description automatically generated

A page fault occurs when a process tries to access a page that is not currently in memory. This can happen if the page is on disk or hasn't been loaded yet.

A screenshot of a computer

Description automatically generated

Both segmentation faults and page faults are critical mechanisms for maintaining memory safety and efficiency in a system. Segmentation faults ensure that processes do not access memory outside their allocated segments, thereby protecting the system from crashes and security breaches. Page faults handle the dynamic loading of pages into memory, ensuring that processes can access large virtual address spaces efficiently, even when not all pages are in physical memory simultaneously. Through these mechanisms, the Unix/Linux Command Line Interpreter ensures robust memory management and efficient execution of user commands.

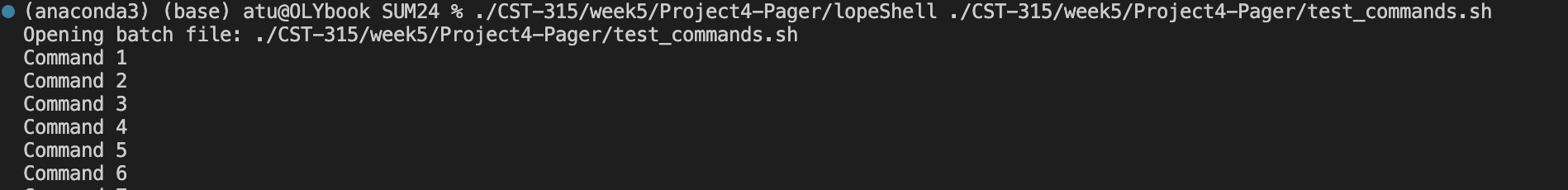
**Testing and Validation**

The testing process involved creating a variety of test cases to cover different aspects of the shell’s functionality, including command execution, memory management, page fault handling, and resource cleanup. Interactive mode was tested by manually entering commands and verifying the correct execution and output, while batch mode was tested by running predefined scripts containing multiple commands. Edge cases, such as accessing invalid memory addresses to trigger segmentation faults and running memory-intensive processes to induce page faults, were specifically tested to validate error handling and recovery mechanisms. Additionally, automated tests were designed to repeatedly execute common and complex scenarios to ensure consistent behavior. Validation also included checking the integrity of the memory management system by monitoring the allocation and deallocation of frames, ensuring no memory leaks or corruption. The correctness of the Least Recently Used (LRU) algorithm was verified by tracking page replacements and ensuring the least recently used pages were correctly identified and swapped. Through these rigorous testing and validation efforts, we ensured that the shell operates efficiently and reliably, providing a solid foundation for further enhancements and real-world usage. I had to ensure the paging mechanism only invoked when it was needed

A screen shot of a black and white screen

Description automatically generated

Testing batch with many commands using an external shell interpreter to run the entire batch script.



A screen shot of a computer

Description automatically generated

A black screen with white numbers

Description automatically generated

Page Fault handling testing

A computer screen shot of text

Description automatically generated

A screenshot of a computer code

Description automatically generated

Testing Resource Manager

A computer screen shot of a program code

Description automatically generated

A screenshot of a computer

Description automatically generated

Testing LRU algorithm

A computer screen shot of a program code

Description automatically generated

A screenshot of a computer

Description automatically generated

**Interesting Features**

1. Memory Allocation

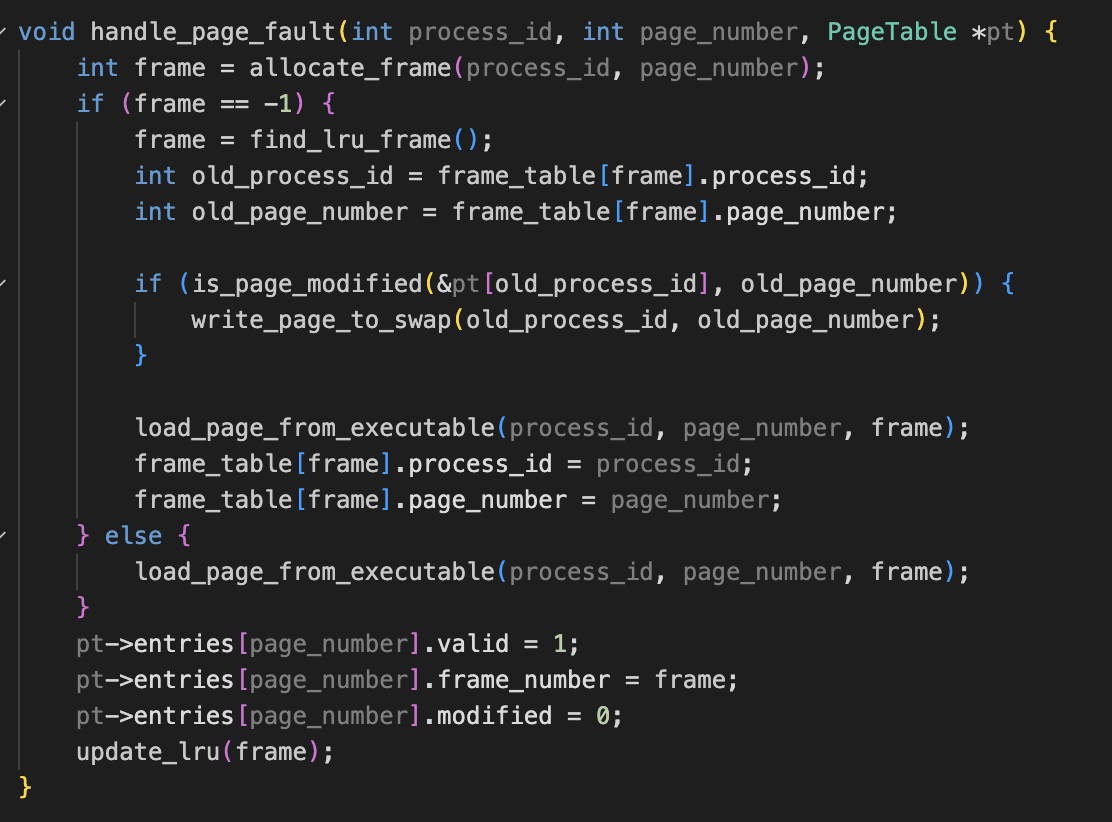
The shell implements a memory allocation system that divides physical memory into fixed-size frames. Each frame is managed to optimize utilization and performance. The allocate\_frame function uses a first-fit allocation algorithm to find and allocate the first available free frame to a process.

A computer screen shot of a code

Description automatically generated

2. Page Fault Handling

When a page fault occurs, the system checks if there are free frames available. If not, it uses the Least Recently Used (LRU) algorithm to find the least recently used frame for replacement. Modified pages are written to swap space before being replaced.



3. Least Recently Used (LRU) Algorithm

The LRU algorithm ensures efficient memory usage by replacing the least recently used page when a page fault occurs and no free frames are available. The update\_lru function updates the usage time of a frame each time it is accessed.

A computer screen with white text

Description automatically generated

4. Resource Management on Process Termination

To prevent resource leaks, the shell includes functions to clean up resources allocated to a process when it terminates. The cleanup\_process\_resources function frees dynamically allocated memory blocks and closes open file descriptors.

A screenshot of a computer program

Description automatically generated

5. Interactive and Batch Modes

The shell can operate in both interactive and batch modes. In interactive mode, it waits for user input and executes commands immediately. In batch mode, it reads commands from a file and executes them sequentially. This dual-mode capability enhances the shell's versatility.

A computer screen shot of text

Description automatically generated

6. Command History

The shell maintains a history of commands entered by the user, allowing the user to view past commands. This feature enhances user experience by providing a convenient way to recall and reuse previous commands.

A computer screen shot of text

Description automatically generated

**References**

Ammann, Paul, and Jeff Offutt. *Introduction to Software Testing*. 2nd ed., Cambridge University Press, 2016.

Baase, Sara, and Allen Van Gelder. Computer Algorithms: Introduction to Design and Analysis. 3rd ed., Addison-Wesley, 1999.

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel*. 3rd ed., O'Reilly Media, 2005.

Cormen, Thomas H., et al. Introduction to Algorithms. 3rd ed., MIT Press, 2009.

Love, Robert. *Linux Kernel Development*. 3rd ed., Addison-Wesley Professional, 2010.

Myers, Glenford J., Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd ed., Wiley, 2011.

Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. Operating System Concepts. 9th ed., Wiley, 2012.

Stallings, William. *Operating Systems: Internals and Design Principles*. 9th ed., Pearson, 2018.

Tanenbaum, Andrew S., and Herbert Bos. Modern Operating Systems. 4th ed., Pearson, 2014.