I. Introduction: The Minishell

The Minishell project is a pivotal experience within the 42 Amman curriculum, meticulously designed to cultivate a profound understanding of operating system interactions from a programmer's vantage point. Its objective extends beyond the mere replication of shell functionalities; it is an academic endeavor into dissecting and implementing the core mechanisms that enable user commands to interface with the kernel's capabilities.[1] This project offers a unique opportunity to, as some have described, "travel through time and come back to problems people faced when Windows didn't exist," thereby fostering a deeper appreciation for the elegance and power of the command-line interface. It is widely regarded as one of the most complex and educationally rewarding projects at 42, with a strong emphasis on process management, file descriptors, and adherence to POSIX standards.[2]
This report serves as a comprehensive guide, navigating through the essential Unix principles that form the bedrock of the Minishell. These include the intricacies of the process model—encompassing creation, execution, and termination—the nuances of file system interactions mediated by file descriptors, the fundamentals of inter-process communication, particularly through pipes, the robust handling of signals for asynchronous event management, and the meticulous management of the shell's environment. Each of these foundational principles will be intrinsically linked to its practical C language implementation within the context of the Minishell project.
Successfully navigating the Minishell project at 42 Amman necessitates strict adherence to a set of well-defined constraints and expectations. A specific list of authorized external C functions forms the developer's toolkit, and adherence to this list is mandatory; these functions include, but are not limited to, readline, malloc, free, write, access, open, read, close, fork, wait, waitpid, signal, sigaction, kill, exit, getcwd, chdir, stat, lstat, fstat, unlink, execve, dup, dup2, pipe, opendir, readdir, closedir, strerror, perror, isatty, ttyname, ttyslot, ioctl, getenv, tcsetattr, tcgetattr, tgetent, tgetflag, tgetnum, tgetstr, tgoto, and tputs. All submitted code, encompassing any bonus features, must rigorously comply with the 42 Norm, which governs aspects such as variable naming, function length (typically constrained to 25 lines), file organization, and overall coding style.
A critical evaluation criterion is the complete absence of memory leaks; all dynamically allocated memory, primarily through malloc, must be scrupulously deallocated using free.[1] The project subject unequivocally states, "No leaks will be tolerated." Furthermore, the project imposes a strict limitation of "no more than one global variable," compelling developers to adopt sophisticated state management strategies, often relying on structures passed between functions, and to be prepared to justify the use of any such global variable.[1] The shell must exhibit robustness, with functions refraining from unexpected termination due to errors like segmentation faults, bus errors, or double frees, except in cases of undefined behaviors explicitly permitted by POSIX standards. A Makefile is a mandatory submission component, utilizing cc as the compsiler with the flags -Wall, -Wextra, and -Werror. This Makefile must not relink and is required to include at least the rules $(NAME), all, clean, fclean, and re. Should bonus features be implemented, a bonus rule is necessary, compiling distinct

_bonus.{c/h} files. If the 42 libft library is employed, its source code and associated Makefile must be incorporated within a libft directory, and the main project's Makefile must ensure the library is compiled first. The scope of the project distinguishes between mandatory core functionalities and optional bonus features, such as logical operators (&&, ||) and wildcard expansion (*) [1]; the latter should only be pursued after the mandatory components are perfected. Finally, the shell is explicitly instructed *not* to interpret unclosed quotes or unspecified special characters like backslashes (\) or semicolons (;), which simplifies initial parsing requirements.[1]

The Minishell project can be viewed as a microcosm of operating system design, compelling an understanding of the shell's role as an intermediary that translates user directives into kernel actions via system calls.[6] The imposed constraints, such as limited global variables and rigorous memory management, mirror the resource-conscious discipline essential in kernel development. Managing processes, I/O, and signals involves orchestrating a multitude of system calls, thereby offering a practical insight into how OS services are exposed and utilized. This endeavor provides a foundational understanding not merely of *how* to use system calls, but *why* they are structured as they are, and how they contribute to the overall coherence and elegance of the Unix operating model.

The pedagogical value of 42 Amman's constraints—the Norm, the curated list of allowed functions, the zero-tolerance policy for memory leaks, and the restriction on global variables—lies in their function as carefully designed educational instruments. The "no leaks" rule is of paramount importance in systems programming, where applications like shells are typically long-running.[1] The Norm, with its stringent function length limits, naturally guides students towards a modular and more maintainable design.[5] The restricted set of allowed functions compels a deeper engagement with the implementation of core logic, rather than a reliance on high-level abstractions. These constraints collectively cultivate disciplined, robust, and efficient C programming habits, which are indispensable for effective low-level systems work. This rigorous methodology, complemented by a peer-review process [1], prepares students for the exacting demands of professional software development, where code quality, resource management, and adherence to specifications are critical.

II. Foundations: Unix Operating System Principles for Your Minishell

A. The Unix Architecture: Kernel, Shell, and System Calls

The Unix operating system architecture is typically conceptualized in distinct layers, providing a structured approach to managing system resources and user interactions. At the foundational layer is the **Hardware**, encompassing all physical components of the computer system. Directly interfacing with the hardware is the **Kernel**, which serves as the core of the operating system.[6] The Kernel is responsible for the direct management of hardware resources such as the CPU, memory, and peripheral devices. It orchestrates process scheduling, controls access to the system, and provides a suite of fundamental services

essential for the system's operation. Above the Kernel lies the **System Call Interface**, the layer where the Minishell predominantly operates.[6] System calls are well-defined, programmatic entry points into the Kernel, enabling user-level processes, such as the shell, to request and utilize Kernel-managed services, like creating a new process or opening a file. The **Shell** itself is a specialized application that leverages these system calls to interpret user commands and mediate interactions with the Kernel. Finally, at the uppermost layer are **Application Libraries and Tools**, which consist of higher-level utilities and libraries constructed upon the system call interface; for instance, the standard C library function printf ultimately relies on the write system call for its output operations.
Your Minishell will function as a command interpreter within this architecture, translating human-readable commands into a sequence of system calls to achieve the desired functionality. In doing so, it effectively bridges the communication gap between the user and the Kernel, acting as the user's primary agent in the system.[6]
The system call interface can be understood as the Kernel's Application Programming Interface (API). It represents a contract: user programs formulate requests in a specific format, and the Kernel, contingent upon appropriate permissions and valid parameters, executes the requested action. This is a crucial point, as [6] emphasizes that system calls form the interface between the kernel and other libraries. This abstraction is fundamental to maintaining the security and stability of the operating system. User programs are precluded from directly accessing kernel data structures or hardware; all such interactions must be channeled through the controlled gateway of system calls. The Minishell will be a significant consumer of this API.
The power of the shell is not derived from executing machine code directly for external programs, but rather from its sophisticated ability to parse user requests and then employ system calls—such as fork, execve, pipe, dup2, and open—to instruct the Kernel in setting up and managing the execution environment for other programs.[6] Consequently, the characteristic features of a shell, including pipelines, I/O redirection, and background process execution, are direct manifestations of its capacity to combine and control these primitive Kernel services.

B. Process Lifecycle: Creation (fork()), Execution (exec() family), and Synchronization (wait(), waitpid())

A **process** is an instance of a program in execution.[9] It encompasses the program's machine code, its data (including global variables and the heap), its runtime stack (for local variables and function calls), and a collection of resources managed by the operating system. These resources include open file descriptors, a unique Process ID (PID), environment variables, and the current working directory.
Process Creation with fork()
The primary mechanism for creating a new process in Unix-like systems is the fork() system call.9
Its C prototype is:
pid_t fork(void);

When fork() is invoked by a process (the parent), the operating system creates a new process (the child) that is a near-exact duplicate of the parent. This duplication includes the parent's memory space (though modern systems often employ a copy-on-write optimization to defer actual memory copying until one of the processes attempts to modify it), the set of open file descriptors (which means the child initially shares access to the same open files and their current offsets), environment variables, and the current working directory.
The fork() call returns different values to the parent and child processes, allowing them to differentiate their subsequent execution paths:

- In the parent process, fork() returns the PID of the newly created child process.
- In the child process, fork() returns 0.
- If an error occurs during process creation (e.g., the system has reached its process limit), fork() returns -1 in the parent process, and no child process is created.

Following a successful fork() call, both the parent and child processes resume execution concurrently from the instruction immediately following the fork().
Program Execution with exec() family
While fork() creates a new process, the exec family of system calls is used to replace the current process's image with a new program.9 The most fundamental of these is execve():
int execve(const char *pathname, char *const argv, char *const envp);
Other functions in the exec family, such as execl, execv, execlp, and execvp, are library functions that provide different ways to specify arguments and often use execve internally.
When execve is called:

- The operating system loads the program specified by pathname into the calling process's memory space, overwriting its existing code, data, and stack segments.
- If execve is successful, it *does not return* to the caller; the new program begins execution from its entry point (e.g., its main function). The PID of the process remains unchanged.
- If execve fails (e.g., pathname does not exist, the file is not an executable, or the process lacks execute permissions), it returns -1, and errno is set to indicate the error.
- argv: This is a NULL-terminated array of character pointers representing the argument strings to be passed to the new program. By convention, argv is the name of the program being executed.
- envp: This is a NULL-terminated array of character pointers, where each string is of the form "VAR=value", representing the environment variables for the new program.

Process Synchronization with wait() and waitpid()
Parent processes often need to wait for their child processes to complete or change state. The wait() and waitpid() system calls facilitate this synchronization.9
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
These functions allow a parent process to suspend its execution until one of its children changes state (typically terminates, but can also include stopping or continuing).

- wait(): Suspends the calling parent process until *any* of its child processes terminates. If a child has already terminated before the call to wait(), wait() returns immediately. It returns the PID of the terminated child or -1 if an error occurs (e.g., no children to wait for).

- waitpid(): Offers more granular control:
  - The pid argument specifies which child process(es) to wait for:
    - pid > 0: Wait for the child whose PID is equal to pid.
    - pid == -1: Wait for any child process (similar to wait()).
    - pid == 0: Wait for any child process whose process group ID is equal to that of the calling process.
    - pid < -1: Wait for any child process whose process group ID is equal to the absolute value of pid.
  - The options argument is a bitmask that modifies waitpid's behavior. Common options include:
    - WNOHANG: Return immediately (non-blocking) if no specified child has changed state. waitpid returns 0 in this case.
    - WUNTRACED: Also return if a child has stopped (e.g., due to a signal like SIGTSTP) but not yet terminated.
    - WCONTINUED: Also return if a stopped child has been resumed by SIGCONT.
- stat_loc: If this argument is not NULL, wait() and waitpid() store status information about the child process in the integer pointed to by stat_loc. This status can be interpreted using various macros defined in <sys/wait.h> [10]:
  - WIFEXITED(status): True if the child terminated normally (via exit() or returning from main).
  - WEXITSTATUS(status): If WIFEXITED is true, returns the child's exit status (the lower 8 bits of the argument to exit() or the return value from main).
  - WIFSIGNALED(status): True if the child was terminated by a signal.
  - WTERMSIG(status): If WIFSIGNALED is true, returns the number of the signal that terminated the child.
  - WIFSTOPPED(status): True if the child is currently stopped (only if WUNTRACED was used).
  - WSTOPSIG(status): If WIFSTOPPED is true, returns the number of the signal that stopped the child.

A crucial role of wait() and waitpid() is to "reap" **zombie processes**.[9] A child process that terminates but whose parent has not yet called wait() or waitpid() on it becomes a zombie. Its process table entry is kept by the OS solely to hold its exit status for the parent. Reaping the zombie by calling wait() or waitpid() allows the OS to release this process table entry and other associated resources. Failure to reap zombies can lead to resource exhaustion if many child processes are created.

The Unix paradigm of separating process creation (fork()) from program loading (exec()) is a cornerstone of its flexibility. This two-step **fork-exec pattern** allows the child process, which initially is a copy of the parent (the shell), to perform setup operations *before* it transforms into the new program. For instance, the child can manipulate its file descriptors for I/O redirection using dup2(). Once these redirections are in place, the child calls execve(). The new program loaded by execve() inherits these modified file descriptors and uses them

transparently, often unaware that its standard input or output has been redirected from/to a file or a pipe. This separation is fundamental to the elegant implementation of shell features like I/O redirection and pipelines.
While wait() provides a simple mechanism for a parent to await child termination, waitpid() offers the sophisticated control necessary for a shell. A shell often needs to manage multiple child processes, distinguishing between foreground jobs (for which it waits specifically) and background jobs (which it might check non-blockingly using WNOHANG). The ability to wait for a specific PID or any child in a process group, and to be notified of stopped children (via WUNTRACED), is essential for implementing features like job control [12] and correctly managing complex pipelines. The 42 Minishell project permits the use of wait3 and wait4 , which can provide even more detailed resource usage information about terminated children, though waitpid is typically the workhorse for the core functionalities.

C. File System Interaction: File Descriptors (0, 1, 2), Standard I/O Streams, and Basic File Operations (open(), close())

In Unix-like operating systems, **file descriptors** (FDs) are small, non-negative integers that the kernel uses to uniquely identify an open file or other I/O resource (such as a pipe or socket) within a particular process.[13] Each process maintains its own table of file descriptors. By convention, three file descriptors are pre-opened for every new process:

- **FD 0:** Standard Input (STDIN_FILENO). This is the default source from which a process reads input.
- **FD 1:** Standard Output (STDOUT_FILENO). This is the default destination to which a process writes its normal output.
- **FD 2:** Standard Error (STDERR_FILENO). This is the default destination for error messages and diagnostic output. Typically, when a shell starts interactively, these three file descriptors are connected to the user's terminal.[13]

Several system calls are fundamental for basic file operations:

- int open(const char *pathname, int flags,... /* mode_t mode */); 13
  This call is used to open an existing file or create and open a new file specified by pathname.
  - flags: A bitmask of options that control the opening mode. Common flags include:
    - O_RDONLY: Open for reading only.
    - O_WRONLY: Open for writing only.
    - O_RDWR: Open for reading and writing.
    - O_CREAT: If the file does not exist, create it. Requires the mode argument.
    - O_TRUNC: If the file exists and is opened for writing, truncate its length to zero.
    - O_APPEND: If the file is opened for writing, append data to the end of the file.
  - mode: (Used only when O_CREAT is specified) An octal value defining the permissions for the newly created file (e.g., 0644 for read/write by owner, read by group and others). The open() call returns a new file descriptor, which is

guaranteed to be the lowest-numbered unused descriptor in the process's FD table.[14] On error, it returns -1 and sets errno.

- int close(int fd); 13
  This call closes the file descriptor fd. This makes fd available for reuse by subsequent open(), pipe(), or socket() calls. If this fd was the last reference to a particular open file description (an internal kernel structure), the resources associated with that open file description are released.
- ssize_t read(int fd, void *buf, size_t count); 14
  This call attempts to read up to count bytes from the file or device associated with fd into the buffer pointed to by buf. It returns the number of bytes actually read (which may be less than count if end-of-file is reached or fewer bytes are available), 0 if end-of-file is reached, or -1 on error.
- ssize_t write(int fd, const void *buf, size_t count); 14
  This call attempts to write up to count bytes from the buffer pointed to by buf to the file or device associated with fd. It returns the number of bytes actually written (which may be less than count if, for example, the disk is full) or -1 on error.

File descriptors provide a crucial abstraction for I/O operations in Unix. They offer a uniform interface regardless of whether the underlying I/O object is a regular file on disk, a pipe connecting two processes, a network socket, or a terminal device. Programs can use read() and write() on file descriptors 0 and 1 without needing to know if these are connected to a terminal, a file (due to redirection), or another process via a pipe. This abstraction is a cornerstone of the Unix "everything is a file" philosophy and is what enables the powerful and flexible I/O redirection and pipelining capabilities that are central to the utility of a shell.

The rule that open() returns the "lowest numbered unused descriptor" [14] is a subtle but significant aspect of file descriptor management. For example, if a process explicitly closes file descriptor 0 (standard input) using close(0), and then immediately calls open("input.txt", O_RDONLY), the file "input.txt" will almost certainly be assigned file descriptor 0. This behavior, while sometimes used for redirection, is less explicit than using dup2(). The dup2() system call is generally the preferred and more robust method for redirecting standard I/O streams because it allows direct specification of the target file descriptor number. Understanding this open() behavior, however, provides deeper insight into the system's file descriptor allocation strategy.

D. Inter-Process Communication: The Power of Pipes (pipe())

Inter-process communication (IPC) is essential for allowing different processes to collaborate and exchange data. In Unix-like systems, one of the simplest and most common forms of IPC is the pipe, created using the pipe() system call.16

The C prototype for pipe() is:

int pipe(int pipefd);

When pipe() is called successfully:

- It creates a unidirectional data channel, an anonymous pipe, which exists within the kernel.

- The pipefd argument, an array of two integers, is populated with two new file descriptors:
    - pipefd is the file descriptor for the **read end** of the pipe. Data can be read from this end.
    - pipefd is the file descriptor for the **write end** of the pipe. Data can be written to this end.
- Data written to pipefd is buffered by the kernel and becomes available for reading from pipefd. Pipes operate in a First-In, First-Out (FIFO) manner.[17]
- The pipe() call returns 0 on success and -1 if an error occurs (e.g., the process's file descriptor table is full).

Pipes are fundamental to implementing shell pipelines, such as command1 | command2, where the standard output of command1 is fed as standard input to command2. The typical sequence of operations for a shell to set up such a pipeline is as follows:

1. **Pipe Creation:** The parent shell process first calls pipe(pipefd) to create the pipe.
2. **Fork for command1:** The parent shell then fork()s to create a child process that will execute command1.
    - **Child 1 (command1) Configuration:**
        - This child process closes pipefd because it will only write to the pipe, not read from it.
        - It then uses dup2(pipefd, STDOUT_FILENO) to duplicate the write end of the pipe (pipefd) onto its standard output file descriptor (1). Now, any data command1 writes to its standard output will go into the pipe.
        - After the dup2() call, the original pipefd in this child is closed, as standard output now serves the same purpose.
        - Finally, Child 1 calls execve() to load and run command1.
3. **Fork for command2:** The parent shell fork()s again to create a second child process that will execute command2.
    - **Child 2 (command2) Configuration:**
        - This child process closes pipefd because it will only read from the pipe, not write to it.
        - It uses dup2(pipefd, STDIN_FILENO) to duplicate the read end of the pipe (pipefd) onto its standard input file descriptor (0). Now, when command2 reads from its standard input, it will receive data from the pipe.
        - After dup2(), the original pipefd in this child is closed.
        - Child 2 then calls execve() to load and run command2.
4. **Parent Shell Actions:**
    - The parent shell process must close *both* of its copies of the pipe file descriptors: close(pipefd) and close(pipefd). This is a critical step. If the parent (or any other process that isn't writing) keeps pipefd (the write end) open, Child 2 (reading from pipefd) might never receive an end-of-file (EOF) indication and could hang indefinitely. Similarly, closing pipefd ensures the parent doesn't inadvertently interfere with the pipe.
    - The parent shell then typically calls waitpid() for both child processes to await

their completion and collect their exit statuses.

Pipes can be conceptualized as in-memory "virtual files".[17] Processes interact with them using the standard file I/O system calls read() and write(), just as they would with regular files. This consistency is a direct result of the file descriptor abstraction, reinforcing the "everything is a file" philosophy of Unix and simplifying IPC design for programmers.

The critical importance of meticulously closing unused pipe file descriptors cannot be overstated. When a pipe is created and a process forks, both the parent and the child inherit file descriptors for both ends of the pipe. Each process must close the end(s) of the pipe it does not intend to use. For example, for a reading process to detect EOF on a pipe, all file descriptors corresponding to the write end of that pipe, across all processes that might have it open, must be closed. If any process (including the parent or another sibling that isn't supposed to write) holds the write end open, the reading process will block, waiting for data that may never arrive.[17] Failure to correctly manage these closures is a very common source of bugs, particularly hanging processes, in shell implementations. This underscores the necessity for careful resource management in concurrent programming, even when using relatively simple IPC mechanisms like anonymous pipes.

E. Asynchronous Events: Understanding and Handling Unix Signals

A **signal** in Unix is a software-generated interrupt, a notification delivered to a process by the operating system or another process to indicate that a specific event has occurred.[18] Signals are identified by unique integer numbers, often represented by symbolic names like $SIGINT (which typically has the value 2). They are a fundamental mechanism for handling asynchronous events.

For the Minishell project, several signals are of particular importance [4]:

- $SIGINT: The interrupt signal, usually generated when the user presses Ctrl-C at the terminal. The default action for a process receiving $SIGINT is to terminate.
- $SIGQUIT: The quit signal, typically generated by Ctrl-\. The default action is to terminate the process and, on many systems, produce a core dump file for debugging.
- $SIGTSTP: The terminal stop signal, commonly generated by Ctrl-Z. The default action is to suspend (stop) the process's execution. This signal is primarily relevant for implementing job control, which is often a bonus feature in Minishell.
- $SIGCHLD: This signal is sent to a parent process whenever one of its child processes terminates, stops, or is continued after being stopped. It is crucial for managing child processes, especially for reaping zombies.

When a process receives a signal, it can take one of three actions [18]:

1. **Default Action:** Perform the system-defined default action for that signal (e.g., termination for $SIGINT).
2. **Ignore:** Explicitly ignore the signal. This is often done by setting the signal's handler to $SIG_IGN.
3. **Catch:** Provide a custom function, known as a **signal handler**, that will be executed when the signal is delivered.

It's important to note that two signals, $SIGKILL and $SIGSTOP, cannot be caught, ignored, or

blocked by a process. They provide a guaranteed way for the system or privileged users to terminate or stop a process.
Unix provides two primary mechanisms for a process to specify how it wants to handle signals:

- signal(int signum, void (*handler)(int)) 18:
  This is the older, simpler interface for setting a signal's disposition.
  - signum: The signal number to handle.
  - handler: Can be $SIG_DFL (for default action), $SIG_IGN (to ignore), or a pointer to a custom signal handler function that takes the signal number as its integer argument. The reliability and specific behavior of signal() (e.g., whether the handler is automatically reset to $SIG_DFL after one invocation) can vary across different Unix implementations, making it less suitable for portable and robust applications.[21]
- int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact) 18:
  This is the POSIX standard interface and is generally preferred for its robustness, portability, and richer feature set. It is an allowed function for the 42 Minishell project.
  - signum: The signal number.
  - act: A pointer to a struct sigaction that specifies the new action to be taken for signum. If act is NULL, sigaction() can be used to query the current action without changing it.
  - oldact: If not NULL, sigaction() stores the previous action associated with signum in the structure pointed to by oldact.
    The struct sigaction is central to this call and has several important members [22]:
  - sa_handler: A pointer to a "simple" signal handler function of type void handler(int signum). This is used if the $SA_SIGINFO flag is *not* set in sa_flags.
  - sa_sigaction: A pointer to an "extended" signal handler function of type void handler(int signum, siginfo_t *info, void *context). This handler is used if the $SA_SIGINFO flag *is* set. The siginfo_t structure provides more detailed information about why the signal was generated (e.g., sending process ID, fault address for memory errors). The void *context argument (usually cast to ucontext_t *) provides machine context information.
  - sa_mask: A sigset_t (signal set) that specifies a set of signals to be blocked (added to the process's signal mask) during the execution of the signal handler. Additionally, the signal that triggered the handler is automatically blocked by default while the handler is running, unless the $SA_NODEFER flag is used.
  - sa_flags: A set of flags that modify the behavior of the signal handling process. Key flags include:
    - $SA_RESTART: If a system call is interrupted by a signal whose handler was installed with this flag, the system call is automatically restarted after the signal handler returns. This is crucial for preventing functions like read() from returning prematurely with EINTR.[22]
    - $SA_SIGINFO: Instructs the system to use the sa_sigaction field for the handler and provide extended signal information.

        - $SA_NOCLDSTOP: If signum is $SIGCHLD, the parent process will not receive this signal when its children stop (e.g., due to $SIGTSTP) or continue (due to $SIGCONT). It will still receive $SIGCHLD when children terminate.[22]
        - $SA_RESETHAND (or the older $SA_ONESHOT): Restores the signal action to $SIG_DFL before calling the signal handler. This mimics the behavior of some older signal() implementations.
        - $SA_NODEFER: Do not add the triggering signal to the process's signal mask while its handler is executing.

The sigaction() system call is the modern standard for reliable signal handling due to several advantages over signal(). As noted in [21], signal() might not be able to correctly re-establish handlers that were originally set using sigaction(). sigaction() provides explicit control over signal masking during handler execution via sa_mask and offers more predictable behavior regarding the restarting of interrupted system calls through the $SA_RESTART flag.[23] These features are essential for writing robust, signal-aware programs. The inclusion of sigaction() in the list of allowed functions for the 42 Minishell project underscores its importance.
The introduction of signals brings asynchronicity into program execution, which can significantly increase complexity. Signal handlers can interrupt the main program flow at virtually any point between machine instructions. This necessitates that signal handlers be written with extreme care, adhering to the concept of "async-signal-safety." An async-signal-safe function is one that can be safely called from within a signal handler without risking deadlocks or data corruption. Many standard library functions (especially those involving dynamic memory allocation like malloc/free or complex I/O like printf) are *not* async-signal-safe. Consequently, signal handlers should typically perform minimal work, such as setting a global volatile sig_atomic_t flag that the main program loop can check, or calling only other async-signal-safe functions. Modifying global data structures shared with the main program from within a handler is particularly perilous and can lead to race conditions unless proper synchronization mechanisms (like blocking signals during critical sections in the main code) are employed. While the signal handling in Minishell might appear straightforward (e.g., re-displaying the prompt on $SIGINT), ensuring these handlers do not corrupt the shell's internal state or interact negatively with complex library functions like readline requires careful design and implementation.

F. The Shell Environment: Variables, environ, and Context

The **environment** of a process in Unix-like systems is a collection of name=value string pairs, known as **environment variables**.[25] These variables serve to pass configuration information, user preferences (like LANG or EDITOR), system paths (PATH), and other operational parameters to programs. When a new process is created via fork(), it inherits a *copy* of its parent's environment.[26]
The C programming language provides several ways to interact with the process environment:

- extern char **environ; 26:
  This is a global variable, typically declared as extern char **environ;. It is a NULL-terminated array of pointers, where each pointer in the array points to a character

string representing an environment variable in the format "NAME=value". While not always declared in a standard header, unistd.h may declare it if the $_GNU_SOURCE feature test macro is defined.26 This environ variable provides direct access to the environment list.

- **Accessing Environment Variables with getenv():**
  - char *getenv(const char *name); [26] This function searches the current environment list for a variable named name.
  - If found, it returns a pointer to the value part of the "NAME=value" string.
  - If not found, it returns NULL.
  - Crucially, the string returned by getenv() should *not* be modified by the application.[29] It might point to static data that could be overwritten by subsequent calls to getenv(), setenv(), or unsetenv(), or it might point directly into the strings managed by environ.
- **Modifying Environment Variables (primarily for implementing shell built-ins like export and unset):**
  - int setenv(const char *name, const char *value, int overwrite); 30
    This function adds or modifies an environment variable.
    - It attempts to add the variable name with the specified value to the environment.
    - If a variable name already exists:
      - If overwrite is non-zero, the existing variable's value is updated to the new value.
      - If overwrite is zero, the existing variable's value is *not* changed. Unlike the older putenv() function, setenv() typically makes copies of the name and value strings it is given, allocating new memory for them within the environment.[32] This makes it safer as the caller does not need to worry about the persistence of the passed strings.
    - It returns 0 on success and -1 on error (e.g., EINVAL if name is invalid, such as containing an '=' character, or ENOMEM if memory allocation fails).
  - int unsetenv(const char *name); 30
    This function removes the environment variable specified by name from the current environment.
    - It returns 0 on success and -1 on error (e.g., EINVAL if name is invalid). If the variable does not exist, it is not an error, and the function still succeeds.
  - int putenv(char *string); 27
    This function takes a string argument of the form "NAME=value". Unlike setenv(), putenv() places the pointer to string itself into the environment. This means the string passed to putenv() must be persistent (e.g., allocated on the heap or static) and not on the stack, as it becomes part of the environment. Modifying the string after the call will modify the environment. Due to these complexities and potential for errors, setenv() is generally preferred over putenv().30

The environ variable represents the ground truth of the process's environment list. While

direct manipulation of environ (e.g., reallocating the array of pointers, managing the strings) is possible, it is highly complex and error-prone. Standard library functions like getenv(), setenv(), and unsetenv() provide a safer, more robust, and portable abstraction layer for these operations.[26] setenv(), for instance, handles the necessary memory allocation and copying when adding or modifying variables, insulating the programmer from these low-level details. This is a common and beneficial pattern in C library design: providing well-defined functions to manage complex or sensitive data structures rather than exposing raw structures for direct manipulation, thereby enhancing portability and reducing the likelihood of bugs.
The overwrite flag in setenv() [30] is more than a minor feature; it reflects the need for intentional state management. It allows a program—and by extension, a shell's built-in commands like export—to define a clear policy for variable modification. Should a new value always supersede an old one, or should an initial setting be preserved if already present? This is crucial for how shells and initialization scripts manage their environments. For example, a shell's export VAR=value command would typically use setenv with overwrite set to 1. However, a user script might wish to set a default value for a variable only if that variable is not already defined in the environment, a scenario where calling setenv with overwrite set to 0 would be appropriate.

III. Building Your Minishell: A Comprehensive Implementation Guide

A. Setting the Stage: Project Structure and Makefile Essentials

A well-organized project structure is fundamental to managing the complexity of the Minishell project. A logical separation of concerns into different files and directories will greatly enhance maintainability, readability, and collaboration. Consider the following organizational scheme:

- **src/ Directory:** This directory will house all your C source files (.c). To further improve organization, especially as the project grows, subdirectories within src/ can be created for distinct modules of functionality.[5] For example:
  - src/parsing/: For files related to command line parsing, tokenization, and syntax analysis.
  - src/execution/: For files handling the execution of commands, including fork, execve, and waitpid logic.
  - src/builtins/: For the implementation of each built-in command (echo, cd, etc.).
  - src/signals/: For signal handling routines.
  - src/utils/: For any utility functions shared across different modules.
- **include/ Directory:** This directory should contain all custom header files (.h). These headers will declare function prototypes that need to be accessible across different source files, define any shared data structures (like those used to represent parsed commands), and include necessary system headers. It is imperative to use **include guards** (e.g., #ifndef MY_HEADER_H #define MY_HEADER_H... #endif) in every header

file to prevent issues arising from multiple inclusions of the same header.
- **libft/ Directory:** If you are utilizing your 42 libft library, its source files and its own Makefile must be placed in a dedicated libft directory at the root of your project.[1]
- **Bonus Files:** As per the 42 project subject, if you implement bonus features, the corresponding source and header files should be named _bonus.c and _bonus.h respectively (or follow a similar pattern if multiple bonus files are needed). These files are also subject to Norm compliance checks.

Your **Makefile** is a critical component, automating the compilation process and ensuring adherence to project requirements :

- **Compiler and Flags:** The Makefile must use cc as the compiler. All compilations must include the flags -Wall -Wextra -Werror. These flags enable a wide range of compiler warnings (-Wall, -Wextra) and treat all warnings as errors (-Werror), enforcing a high standard of code quality and helping to catch potential bugs early.
- **Required Targets:**
  - $(NAME): This rule should compile all source files and link them to create the final executable (e.g., minishell). The variable NAME should be defined in your Makefile to hold the desired executable name.
  - all: This rule should be an alias for the $(NAME) rule.
  - clean: This rule must remove all object files (.o) generated during compilation.
  - fclean: This rule must remove all object files *and* the final executable ($(NAME)).
  - re: This rule must perform an fclean followed by an all, effectively recompiling the entire project from scratch.
- **No Relinking:** The Makefile must be structured to avoid unnecessary relinking. Object files should only be recompiled if their corresponding source files (or any included headers) have changed.
- **Libft Compilation:** If libft is used, your main Makefile must first call the Makefile within the libft/ directory to compile the library, and then link your Minishell object files against the generated libft.a.
- **Bonus Rule:** If bonus features are implemented, a bonus target is required. This rule should compile the main project along with the bonus files (e.g., _bonus.c). It may involve different compilation flags or linking different libraries if the bonus part uses functions not allowed in the mandatory part (though for Minishell, the allowed functions list is usually the same for both).

A well-structured Makefile not only simplifies the build process but also serves as documentation for how the project is compiled.

B. The Core: The Read-Eval-Print Loop (REPL)

The heart of any interactive shell is its Read-Eval-Print Loop (REPL). This loop continuously performs three fundamental actions:

1. **Read:** Reads a line of input from the user.
2. **Eval:** Parses and executes the command(s) entered by the user.
3. **Print:** (Implicitly) Displays the output of the command and, upon completion, presents a new prompt for the next command.

1. Reading User Input with GNU Readline:
For the Minishell project, the use of the GNU Readline library is permitted and highly recommended for handling user input.1 Readline provides sophisticated line-editing capabilities, command history, and completion features, significantly enhancing the user experience beyond what simple read() calls could offer.34

- **char *readline(const char *prompt);** [34]
    - This is the primary function for getting a line of input.
    - It displays the prompt string to the user.
    - It allows the user to type and edit the command line using familiar Emacs-like (default) or vi-like keybindings.
    - Upon pressing Enter, readline returns a pointer to a dynamically allocated string (malloc'd) containing the entered line (excluding the newline character).
    - If EOF is encountered (e.g., user types Ctrl-D on an empty line) or an error occurs, readline returns NULL.
    - **Important:** The string returned by readline must be freed by your program when it's no longer needed to prevent memory leaks.[36] While the 42 subject mentions that leaks from readline itself don't need to be managed [4], any memory you allocate based on its return value, or the returned string itself, is your responsibility.

C
```
// Simplified REPL structure
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h> // For add_history

int main(void) {
  char *line;
  // Initialize shell state, environment, signal handlers etc.

  while (1) {
    line = readline("minishell> "); // Display prompt and read input
    if (line == NULL) {
      // EOF (Ctrl-D) detected
      printf("exit\n"); // Mimic bash behavior
      break; // Exit the loop, and then the shell
    }

    if (line!= '\0') { // If the line is not empty
      add_history(line); // Add non-empty line to history

      // TODO: Parse and execute the command in 'line'
      // eval_and_execute(line, &shell_state);
    }
```

```
        free(line); // Free the memory allocated by readline
        line = NULL;
    }
    // Cleanup shell state, free environment etc.
    // rl_clear_history(); // Optional: clear history on exit
    return (0); // Or last command's exit status
}
```

2. Managing Command History:
Readline seamlessly integrates with a history library, allowing users to navigate, recall, and edit previously entered commands using arrow keys (Up/Down).35

- **void add_history(const char *string);** [34]
  - Adds the string to the end of the command history list.
  - Typically, you call this after successfully reading a non-empty line and before executing it.
  - The 42 subject requires a "working History".[1]
- **History File (.bash_history equivalent):**
  - Readline can persist history across sessions by reading from and writing to a history file.
  - int read_history(const char *filename);: Loads history from filename (e.g., ~/.minishell_history).
  - int write_history(const char *filename);: Saves the current in-memory history to filename.
  - int append_history(int nevents, const char *filename);: Appends the last nevents to filename.
  - Bash uses environment variables like HISTFILE, HISTSIZE, and HISTFILESIZE to manage this.[37] For Minishell, implementing persistent history might be a bonus or an advanced feature, but in-session history via add_history and Readline's navigation is fundamental. The 42 subject for Minishell does not explicitly require saving history to a file, but a "functional history" implies at least in-memory history accessible via arrow keys.
- **Other Readline Functions (as per 42 allowed list):**
  - rl_clear_history(): Clears the in-memory history list.
  - rl_on_new_line(): Tells Readline to move to a new line before printing the prompt (useful after a signal like Ctrl-C).
  - rl_replace_line(const char *text, int clear_undo): Replaces the contents of the current Readline buffer.
  - rl_redisplay(): Redraws the current input line.

The use of GNU Readline significantly elevates the user-friendliness of the Minishell. It abstracts away the complexities of terminal raw mode, character-by-character input handling, and line editing, allowing the developer to focus on the core shell logic of parsing and execution. The history mechanism, even if only in-memory for the basic Minishell, is a crucial usability feature that users expect from any interactive shell. Bash's sophisticated history management, involving variables like HISTFILE (default ~/.bash_history), HISTSIZE (in-memory commands), and HISTFILESIZE (commands in file) [37], provides a model for how a more advanced shell handles command history persistence and control. While Minishell might not implement all these nuances, understanding them informs the design of a robust history system.

C. Command Input Parsing: From String to Actionable Structure

Once a line of input is read from the user, the next critical step is **parsing**. Parsing transforms the raw command string into a structured representation that the shell can understand and act upon. This process typically involves two main phases: lexical analysis (tokenization) and syntactic analysis (building a command structure).[41]

1. Lexical Analysis (Tokenization):

Lexical analysis, or tokenization, is the process of breaking down the input string into a sequence of meaningful units called tokens. Tokens can be command names, arguments, operators (like |, <, >), or quoted strings.15

- **Delimiters:** Whitespace characters (spaces, tabs) are the primary delimiters that separate tokens.
- **C Functions for Tokenization:**
  - **strtok(char *str, const char *delim)** [15]:
    - A standard C library function for tokenizing a string.
    - On the first call, str is the string to tokenize, and delim is a string of delimiter characters. strtok finds the first token, replaces the delimiter following it with a null byte (\0), and returns a pointer to the token.
    - On subsequent calls for the same string, str must be NULL. strtok continues from where it left off.
    - Returns NULL when no more tokens are found.
    - **Limitations for Shells:**
      - **Destructive:** strtok modifies the input string by inserting null bytes. A copy must be made if the original is needed.
      - **Not Reentrant/Thread-Safe:** It uses a static internal pointer to keep track of its position, making it unsuitable for concurrent or nested tokenization scenarios.[44]
      - **Consecutive Delimiters:** Treats multiple consecutive delimiters as a single one, which might not always be desired.
      - **Quote Handling:** strtok itself does not inherently understand quotes. Logic to handle quoted strings (where delimiters inside quotes are ignored) must be built around it. The 42 Minishell subject specifies that the shell should not interpret unclosed quotes [1], simplifying this somewhat but still requiring quote-aware tokenization.
  - **strtok_r(char *str, const char *delim, char **saveptr)** [44]:
    - A reentrant version of strtok.
    - The saveptr argument is a pointer to a char * variable that strtok_r uses to

store its context between calls, instead of a static internal pointer.

    - This makes strtok_r thread-safe and allows for multiple, interleaved tokenization operations on different strings, or nested tokenization on parts of a string.[44] For complex parsing where a token might itself need to be tokenized (e.g., variable assignments like VAR="value1 value2"), strtok_r is essential.
    - Given its robustness, strtok_r is generally preferred over strtok for non-trivial parsing tasks, especially in a program like a shell that might evolve to handle more complex inputs.

C

```
// Example using strtok_r
#include <string.h>
#include <stdio.h>

void tokenize_line(char *line) {
   char *token;
   char *saveptr; // Context for strtok_r

   // Delimiters: space and tab
   const char *delimiters = " \t";

   token = strtok_r(line, delimiters, &saveptr);
   while (token!= NULL) {
      printf("Token: [%s]\n", token);
      // Here, you would add the token to your command structure
      token = strtok_r(NULL, delimiters, &saveptr);
   }
}
```

This example is basic and does not handle quotes or special shell operators. A real shell tokenizer would need to be more sophisticated, iterating character by character when necessary to correctly identify tokens boundaries, especially around quotes and operators.

- **Manual Tokenization:** For full control, especially over quote handling and special characters, a custom tokenizer can be written. This involves iterating through the input string character by character, maintaining state (e.g., "am I inside single quotes?", "am I inside double quotes?"), and accumulating characters into tokens. This approach avoids the limitations of strtok/strtok_r regarding quote-awareness. The shlex module in Python [41] provides a good conceptual model for such a lexical analyzer, handling various quoting and escaping rules.

2. Syntactic Analysis (Parsing into Command Structures):

Once the input line is tokenized, syntactic analysis (or simply parsing) takes over. This phase interprets the sequence of tokens to build an internal representation of the command(s) to be executed. This representation is often a data structure, such as a linked list of commands or an Abstract Syntax Tree (AST).2

- **Data Structures for Parsed Commands:**

    - A common approach is to define structs to represent commands and their components. For simple commands without pipes or logical operators, a structure might hold:
      C
      ```
      typedef struct s_simple_command {
         char **argv;           // Array of arguments (token is command name)
         char *input_redir_file;  // Filename for input redirection (<)
         char *output_redir_file; // Filename for output redirection (>, >>)
         int   append_output;     // Flag for >>
         char *heredoc_delimiter; // Delimiter for <<
         // Potentially pointers to next command in a pipeline or conditional execution
         struct s_simple_command *next_piped_command;
         //... other fields for pipes, logical ops etc.
      } t_simple_command;
      ```
    - For pipelines (cmd1 | cmd2 | cmd3), these t_simple_command structures can be linked together in a list.[46]
    - For more complex syntax involving logical operators (&&, ||) or command grouping, an Abstract Syntax Tree (AST) is often more suitable.[2] In an AST, operators form internal nodes, and commands form leaf nodes. explicitly mentions using an asymmetric binary tree for command parsing in one Minishell implementation.
    - [42] describes a parser that builds a "command table" (an array of structs) where each row represents a simple command in a pipeline, storing command, options, arguments, and I/O redirection details.
- **Parsing Logic:**
    - The parser iterates through the tokens generated by the lexical analyzer.
    - It identifies command names, arguments, and redirection operators (<, >, >>, <<, |).
    - It populates the chosen data structure(s) with this information.
    - For redirections, it extracts the associated filenames.
    - For pipes, it links the simple command structures.
    - The parser must handle syntax errors gracefully (e.g., a redirection operator without a filename, an unexpected token). The 42 Minishell project requires that the shell does not interpret unclosed quotes or unspecified special characters like \ or ; [1], which simplifies error conditions related to these.
    - One approach to parsing is using a state machine or a recursive descent parser, where different functions handle different parts of the shell grammar (e.g., a function to parse a simple command, a function to parse a pipeline). [42] provides an example of grammar rules (q00: NEWLINE {return 0;} | cmd q1 q0 | error;) for a YACC-based parser, illustrating a more formal approach.

The choice between strtok (or strtok_r) and a fully custom tokenizer depends on the complexity of the syntax to be supported. While strtok_r is reentrant and can handle basic

tokenization, it lacks built-in awareness of shell quoting rules. For a Minishell that needs to correctly parse quoted arguments (e.g., echo "hello world" as two arguments: echo and hello world), a custom tokenizer that manages quote states is often necessary. This custom tokenizer would then feed its tokens to the syntactic analyzer.
The parsing stage is often considered one of the most challenging parts of the Minishell project.[8] A robust parser is crucial because the correctness of all subsequent execution steps depends on an accurate internal representation of the user's command. The data structures chosen (e.g., linked lists of command structs, or an AST) will significantly influence how the executor module traverses and processes the commands.

D. Handling Quotes and Special Characters

Shells use quoting mechanisms to control the interpretation of special characters and to group words containing whitespace into single arguments. The Minishell project requires specific handling for single quotes ('), double quotes ("), and the dollar sign ($) for variable expansion.[1] Backslashes (\) and semicolons (;) are explicitly *not* to be interpreted as special characters in the mandatory part.[1]

1. Single Quotes ('):

- **Behavior:** Enclosing text within single quotes preserves the literal value of *all* characters within the quotes.[48] No character inside single quotes has any special meaning to the shell. This includes spaces, tabs, wildcard characters, and even the dollar sign ($) and backslash (\).
- **Minishell Requirement:** "Handle ' (single quote) which should prevent the shell from interpreting the metacharacters in the quoted sequence".[4]
- **Implementation:** During tokenization, when a single quote is encountered, the tokenizer should switch to a "single-quoted state." It then accumulates all characters (including spaces, $, etc.) as part of the current token until a matching closing single quote is found. The quotes themselves are typically removed from the final token value.[48]
    - Example: echo 'Hello $USER, this is * a test' should result in one token: Hello $USER, this is * a test.

2. Double Quotes ("):

- **Behavior:** Double quotes also preserve the literal value of most characters within them, but with some crucial exceptions.[48]
    - **Variable Expansion ($):** The dollar sign ($) retains its special meaning for variable expansion (e.g., $VAR, ${VAR}).
    - **Command Substitution:** Backticks (`command`) and $(command) (though backticks might not be required for Minishell) would typically be interpreted for command substitution. Minishell's scope might limit this.
    - **Backslash (\):** Within double quotes, a backslash (\) retains its special meaning only when followed by certain characters: $, `, ", \, and newline. This allows these characters to be included literally within a double-quoted string.[50] For Minishell, since general backslash interpretation is out of scope for the mandatory part, its

    behavior within double quotes might be simplified to primarily affect $, ", and perhaps \ itself if needed to escape a literal " or $.

- **Minishell Requirement:** "Handle " (double quote) which should prevent the shell from interpreting the metacharacters in the quoted sequence except for $ (dollar sign)".[4] This means variable expansion *must* occur within double quotes. Other metacharacters (like *, |, < if they were inside) should be treated literally.
- **Implementation:** When a double quote is encountered, the tokenizer enters a "double-quoted state." It accumulates characters. If a $ is encountered (and not escaped, if escape handling within double quotes for $ is implemented), it triggers variable expansion logic (see section III.H). Other characters are generally taken literally. The process continues until a matching closing double quote is found. The quotes themselves are removed.
    - Example: If USER is "jdoe", echo "Hello $USER, this is * a test" should result in one token: Hello jdoe, this is * a test.

3. Dollar Sign ($):

- **Behavior:** The dollar sign initiates variable expansion (e.g., $VAR, $?) and, in more advanced shells, other forms of expansion.
- **Minishell Requirement:**
    - Expand environment variables ($VAR).[4]
    - Expand the exit status of the last executed foreground pipeline ($?).[4]
    - Expansion occurs within double quotes but not single quotes.
- **Implementation:** This is a complex part, often called "expansion," which might happen after initial tokenization or as part of a more sophisticated parsing phase.[5]
    - When a $ is found outside single quotes (or inside double quotes), the parser/expander must identify the variable name (e.g., alphanumeric characters following $, or the special character ?).
    - It then retrieves the variable's value (from the environment or internal shell state for $?).
    - The $VARIABLENAME sequence in the token is replaced by the variable's value.
    - If a variable is unset, it typically expands to an empty string.
    - Example: echo $HOME/file with HOME=/Users/me becomes echo /Users/me/file.

4. Unclosed Quotes and Other Special Characters:

- **Minishell Requirement:** "Not interpret unclosed quotes or unspecified special characters like \ or ;".[1]
    - **Unclosed Quotes:** If the input ends with an unclosed single or double quote, Minishell is not required to prompt for more input (like bash does with dquote>). The behavior might be to treat the rest ofthe line as part of the quoted string up to the end of the line, or to report a syntax error. The exact expected behavior should be clarified by 42 project guidelines or by observing bash's behavior in simple non-interactive cases if that's the reference. However, [4] explicitly says "does not interpret unclosed quotes", which might mean they are treated as literal characters if not forming a valid pair, or lead to a parsing error.

  - **Backslashes (\):** Generally not interpreted as an escape character in the mandatory part, except possibly within double quotes to escape $, ", or \ itself, depending on how strictly "except for $" is interpreted for double quotes.
  - **Semicolons (;):** Not interpreted as command separators in the mandatory part.

Integration into Parsing:
Handling quotes and expansions correctly requires the tokenizer and parser to be stateful.

1. **Tokenizer:**
   - When not in a quoted state, whitespace delimits tokens. Special characters like |, <, > also act as delimiters and form their own tokens.
   - Upon encountering ' or ", the tokenizer enters a corresponding quoted state.
   - In a quoted state, it consumes characters, treating them literally (for single quotes) or performing limited interpretation like $ expansion (for double quotes), until the matching closing quote is found. The accumulated string (minus the outer quotes) becomes a single token.
2. **Expander (Post-Tokenization or Integrated):**
   - After initial tokenization, or as part of it if the tokenizer is sophisticated enough, tokens that are eligible for expansion (not single-quoted, or double-quoted and containing $) are processed.
   - $VAR and $? are replaced by their values.
   - This might result in a token changing (e.g., $USER becomes jdoe) or even splitting into multiple tokens if the expanded value contains whitespace and was not itself within quotes (e.g., VAR="a b"; echo $VAR might lead to echo, a, b). However, the Minishell subject is usually simpler and might expect echo $VAR where VAR="a b" to pass "a b" as a single argument if the expansion happens before field splitting, or two arguments a and b if field splitting occurs after expansion on unquoted variables. Bash's behavior here is complex (word splitting and pathname expansion after parameter expansion on unquoted results). Minishell usually simplifies this: an expanded token that was originally unquoted might be subject to word splitting, while one from within double quotes ("$VAR") is not.

The shell's quoting mechanism is fundamental for users to control how their input is interpreted, allowing special characters to be used as literal data or for their syntactic meaning. For the Minishell developer, this means the parser must meticulously track quoting state to correctly delineate arguments and identify parts of the command line that require expansion. The clear distinction between single quotes (no interpretation) and double quotes (only $ interpretation for Minishell) provides a defined set of rules to implement.[4]

E. Command Execution: External Programs and Built-in Commands

After parsing the command line into a structured representation, the shell must execute the command(s). Unix shells distinguish between two primary types of commands: **external programs** (executable files residing in the file system) and **built-in commands** (implemented directly within the shell itself).[52]

1. Executing External Programs:

This is the common case for most commands like ls, grep, cat, etc. The execution typically follows the fork-exec-wait pattern 9:

- **fork():** The shell first creates a new child process using fork(). The child process is a near-duplicate of the parent shell.
- **Child Process Setup:** Before executing the new program, the child process is responsible for setting up its execution environment. This is where I/O redirections (<, >, >>, <<) and pipe connections are established using system calls like dup2(), open(), and close(). Signal handling might also be adjusted (e.g., resetting certain signals to their default behavior for the child).
- **execve():** The child process then calls execve(pathname, argv, envp) to replace its current image with the new program.[9]
    - pathname: The absolute or relative path to the executable file. The shell needs to resolve the command name to a full path. This involves:
        - Checking if the command given by the user is already an absolute path (starts with /) or a relative path (contains /).
        - If it's a simple command name (e.g., ls), the shell must search for the executable in the directories listed in the PATH environment variable.[1] This involves tokenizing the PATH string (colon-separated), appending the command name to each directory, and checking for the existence and executability of the resulting path using access(path, X_OK). The first valid path found is used.
    - argv: A NULL-terminated array of strings representing the arguments to the command. argv is conventionally the command name itself. These arguments are derived from the parsing phase.
    - envp: A NULL-terminated array of strings for the child's environment, usually a copy of the shell's current environment (accessible via extern char **environ;). If execve is successful, it does not return. If it fails (e.g., command not found, permission denied), it returns -1, and the child process should then print an error message and exit with an appropriate status.
- **Parent Process (waitpid()):**
    - After forking, the parent shell (if the command is run in the foreground) typically calls waitpid(child_pid, &status, 0) to wait for the child process to terminate.[9]
    - Once the child terminates, waitpid returns, and the parent can retrieve the child's exit status from the status variable using macros like WIFEXITED() and WEXITSTATUS().[11] This exit status is then used to set the shell's $? special variable.

2. Implementing Built-in Commands:
Certain commands are implemented directly within the shell for several reasons 52:

- **Modifying Shell State:** Commands like cd (changes the shell's current working directory), exit (terminates the shell), export and unset (modify the shell's environment) *must* be built-in. If they were external programs, they would run in a separate child process. Any changes they make (e.g., chdir() in a child) would only affect that child's environment, and the parent shell's state would remain unchanged after the child exits.[54]

- **Efficiency:** For very simple and frequently used commands like pwd or echo, implementing them as built-ins avoids the overhead of forking a new process and executing an external program.
- **Special Functionality:** Commands like history or alias inherently operate on the shell's internal data structures.

Minishell Required Built-ins 1:
echo (with -n), cd (relative/absolute paths only), pwd (no options), export (no options when setting, lists all exported vars if no args), unset (no options), env (no options/args), exit (no options).
Implementation Strategy for Built-ins:

- During command parsing or before execution, the shell checks if the command name matches one of the built-in commands.
- If it's a built-in, the shell *does not* fork().
- Instead, it directly calls a corresponding C function within the shell's own process to perform the built-in's action.
- This function will have access to and can modify the shell's internal state (e.g., current directory, environment variables).
- The built-in function should return an integer status (0 for success, non-zero for failure) which the shell uses to set its $? variable.

The separation between fork() for creating a new process context and execve() for loading a new program is a defining characteristic of Unix process management. This two-step process provides immense flexibility, particularly for the shell. The child process, being a copy of the shell, inherits the shell's environment, including its file descriptors. This allows the child to perform setup tasks, such as redirecting its standard input, output, or error file descriptors using dup2(), *before* it calls execve(). The program loaded by execve() then transparently uses these (potentially redirected) standard file descriptors without needing any special logic for redirection itself. This elegant mechanism is what enables the powerful I/O redirection features of Unix shells.
The execve() system call is, in a sense, a point of no return for the process that calls it successfully. It completely overwrites the calling process's memory image with the new program. If execve() fails, however, it returns to the caller (the child process in the fork-exec model), which must then handle the error, typically by printing an error message to standard error and exiting with a non-zero status to signal failure to the parent shell.
Built-in commands are essential because they operate directly on the shell's internal state or environment. If cd were an external command, it would change the directory of a new child process, but the parent shell's directory would remain unaffected once that child exits.[54]
Similarly, export needs to modify the parent shell's list of environment variables that will be passed to subsequent child processes. This direct manipulation of the shell's context necessitates their implementation as internal functions rather than external executables.

F. Implementing I/O Redirection: <, >, >>, <<

I/O redirection is a fundamental shell feature that allows users to control where a command's input comes from and where its output (both standard output and standard error) goes. The Minishell project requires the implementation of input redirection (<), output redirection (>), append output redirection (>>), and here-documents (<<).[1] These are achieved by manipulating file descriptors within the child process before it executes the command, primarily using the open(), close(), dup(), and dup2() system calls.[13]

1. Output Redirection (>): command > filename

- **Goal:** Send the standard output (FD 1) of command to filename. If filename does not exist, it is created. If it exists, its contents are overwritten (truncated).
- **Implementation Steps (in the child process before execve):**
    1. Open filename for writing, creating it if it doesn't exist, and truncating it if it does: int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644); (Permissions like 0644 grant read/write to owner, read to group/others).
    2. If open() fails (e.g., permission denied), print an error and exit the child.
    3. Redirect standard output to the opened file: dup2(fd, STDOUT_FILENO); (or dup2(fd, 1);) This call atomically closes STDOUT_FILENO if it was open and makes STDOUT_FILENO a duplicate of fd. Now, anything written to FD 1 by the command will go to filename.
    4. Close the original file descriptor fd, as it's no longer needed (FD 1 now points to the file): close(fd);
    5. The child then calls execve() to run the command.

2. Append Output Redirection (>>): command >> filename

- **Goal:** Send the standard output of command to filename, appending to the file's existing content. If filename does not exist, it is created.
- **Implementation Steps (in child):**
    1. Open filename for writing in append mode, creating it if it doesn't exist: int fd = open(filename, O_WRONLY | O_CREAT | O_APPEND, 0644); The key difference from > is the O_APPEND flag.
    2. If open() fails, print an error and exit child.
    3. Redirect standard output: dup2(fd, STDOUT_FILENO);
    4. Close fd: close(fd);
    5. Child calls execve().

3. Input Redirection (<): command < filename

- **Goal:** Take standard input (FD 0) for command from filename.
- **Implementation Steps (in child):**
    1. Open filename for reading: int fd = open(filename, O_RDONLY);
    2. If open() fails (e.g., file not found, permission denied), print an error and exit child.
    3. Redirect standard input from the opened file: dup2(fd, STDIN_FILENO); (or dup2(fd, 0);) Now, any attempt by the command to read from FD 0 will read from filename.

4. Close fd: close(fd);
5. Child calls execve().

4. Here-Documents (<<): command << DELIMITER... DELIMITER

- **Goal:** Feed lines of input, typed directly into the shell or script, as standard input to command until a line containing only DELIMITER is encountered. The history does not need to be updated for these input lines.[4]
- **Implementation Approaches:**
  - **Using a Temporary File (Common in bash, zsh [57]):**
    1. The parent shell reads lines from the user (e.g., using readline in a loop) until a line matching DELIMITER is found.
    2. These lines (excluding the final delimiter line) are written to a temporary file. mktemp() or a similar mechanism can be used to create a unique temporary filename.[58]
    3. After forking the child process for command:
       - The child opens this temporary file for reading.
       - The child uses dup2() to redirect its STDIN_FILENO to the temporary file's FD.
       - The child closes the original FD for the temporary file.
       - The child calls execve().
    4. The parent shell, after the child is set up or has completed, should delete the temporary file (unlink()). [57] details that bash often unlink()s the temp file immediately after opening it for reading in the child, ensuring it's cleaned up even if the shell crashes, as the file is removed when its last FD is closed.
  - **Using a Pipe (Common in dash [57]):**
    1. The parent shell creates a pipe: pipe(pipefd);
    2. The parent shell forks a child process for command.
       - **Child (command):**
         - Closes pipefd (write end).
         - Redirects its STDIN_FILENO to pipefd (read end) using dup2(pipefd, STDIN_FILENO);.
         - Closes pipefd.
         - Calls execve().
       - **Parent Shell:**
         - Closes pipefd (read end).
         - Reads lines from the user (e.g., using readline) until DELIMITER is found.
         - Writes each of these input lines to pipefd (write end of the pipe).
         - After writing all lines, the parent closes pipefd. This signals EOF to the child reading from the pipe.
         - Waits for the child process to complete.

The 42 Minishell subject allows readline for input , which can be used to read the lines for the

here-document. The choice between a temporary file and a pipe for here-documents involves trade-offs. Temporary files might be simpler to manage for very large inputs but involve disk I/O. Pipes are in-memory and avoid disk I/O but require careful synchronization of writing by the parent and reading by the child, and ensuring the write end is closed to signal EOF. Given that here-documents are often small, the pipe approach can be more elegant and efficient.[57]
The dup2(oldfd, newfd) system call is particularly powerful because it performs the close-of-newfd (if open) and the duplication of oldfd onto newfd as a single, atomic operation. This atomicity is important to avoid race conditions, especially in multithreaded programs, though for a single-threaded child process setting up its FDs, the main benefit is convenience and ensuring newfd is correctly reassigned. If oldfd is the same as newfd, dup2 does nothing and returns newfd, which is a safe behavior.

G. Implementing Pipes (|)

Pipes are a powerful Unix feature that allows the standard output of one command to be connected directly to the standard input of another command, forming a pipeline (e.g., ls -l | grep ".c" | wc -l). The Minishell project requires the implementation of pipes (| character).[1] This is achieved by using the pipe(), fork(), dup2(), and close() system calls in a coordinated manner for each command in the pipeline.[16]
Consider a simple two-command pipeline: cmd1 | cmd2.
Implementation Steps:

1. **Parse the Pipeline:** The shell first parses the input line and identifies the commands (cmd1, cmd2) and the pipe operator separating them.
2. **Create a Pipe:** The parent shell calls pipe(pipefd); to create a pipe. pipefd will be the read end, and pipefd will be the write end. Error checking for pipe() is essential.
3. **Fork for cmd1 (Left side of the pipe):**
   - The parent shell fork()s a child process (Child1).
   - **In Child1:**
     - Close the read end of the pipe: close(pipefd); (Child1 will write to the pipe, not read from it).
     - Redirect Child1's standard output (STDOUT_FILENO) to the write end of the pipe: dup2(pipefd, STDOUT_FILENO);. Now, anything cmd1 writes to its standard output will go into the pipe.
     - Close the original write-end file descriptor pipefd, as it's now redundant (FD 1 serves the same purpose).
     - Execute cmd1 using execve(). If execve fails, Child1 should print an error and exit.
4. **Fork for cmd2 (Right side of the pipe):**
   - The parent shell fork()s another child process (Child2).
   - **In Child2:**
     - Close the write end of the pipe: close(pipefd); (Child2 will read from the pipe, not write to it).
     - Redirect Child2's standard input (STDIN_FILENO) to the read end of the pipe: dup2(pipefd, STDIN_FILENO);. Now, when cmd2 reads from its

        standard input, it will get data from the pipe.
        - Close the original read-end file descriptor pipefd, as it's now redundant (FD 0 serves the same purpose).
        - Execute cmd2 using execve(). If execve fails, Child2 should print an error and exit.
5. **Parent Shell Actions:**
    - The parent shell has created two children and set up the pipe between them. Now, the parent must close *both* its copies of the pipe file descriptors: close(pipefd); and close(pipefd);.
        - **This is extremely important.** If the parent does not close pipefd (the write end), Child2 might never see an EOF when reading from pipefd, because the kernel sees that at least one process (the parent) still has the write end open. This can cause Child2 to hang.
        - Closing pipefd is also good practice to free up the descriptor and avoid potential issues.
    - The parent shell then waits for both Child1 and Child2 to terminate, typically using waitpid() for each. The exit status of the pipeline is usually the exit status of the *last* command in the pipeline (cmd2 in this case).

Handling Multi-Command Pipelines (e.g., cmd1 | cmd2 | cmd3):
The logic extends for longer pipelines. For a command in the middle of a pipeline (like cmd2 in cmd1 | cmd2 | cmd3), it needs to:

- Read its standard input from the pipe connected to the previous command.
- Write its standard output to a new pipe connected to the next command.

This often involves a loop in the parent shell that iterates through the commands in the pipeline. For each command except the last, a new pipe is created. The input for the current command (if not the first) is taken from the read end of the *previous* pipe. The output for the current command (if not the last) is directed to the write end of the *current* pipe. Careful management of file descriptors is crucial:

- Each child process must close all pipe file descriptors it doesn't use.
- The parent must close all pipe file descriptors after all necessary children have been forked and have duplicated the required ends.

A common pattern for managing FDs in a loop for pipelines:

C

```
// Simplified conceptual loop for pipelines
// Assumes 'commands' is an array/list of command structures
// int num_commands =...;
// t_command *commands =...;

int in_fd = STDIN_FILENO; // Input for the first command is stdin
int pipefd;
```

```
for (int i = 0; i < num_commands; i++) {
   if (i < num_commands - 1) { // If not the last command, create a pipe for its output
      if (pipe(pipefd) == -1) {
         perror("pipe");
         // Handle error: cleanup, exit children etc.
         exit(EXIT_FAILURE);
      }
   }

   pid_t pid = fork();
   if (pid == -1) {
      perror("fork");
      // Handle error
      exit(EXIT_FAILURE);
   }

   if (pid == 0) { // Child process
      if (in_fd!= STDIN_FILENO) { // If not the first command
         dup2(in_fd, STDIN_FILENO); // Redirect stdin from previous pipe's read end
         close(in_fd);            // Close original previous pipe's read end
      }
      if (i < num_commands - 1) { // If not the last command
         close(pipefd);        // Close current pipe's read end (child writes to it)
         dup2(pipefd, STDOUT_FILENO); // Redirect stdout to current pipe's write end
         close(pipefd);        // Close original current pipe's write end
      }
      // execve(commands[i]...);
      perror("execve failed");
      exit(EXIT_FAILURE); // Should not be reached if execve succeeds
   } else { // Parent process
      if (in_fd!= STDIN_FILENO) {
         close(in_fd); // Close parent's copy of previous pipe's read end
      }
      if (i < num_commands - 1) {
         close(pipefd); // Close parent's copy of current pipe's write end
         in_fd = pipefd; // Save current pipe's read end for the next command's input
      }
      // Store pid to wait for it later
   }
}

// After the loop, parent waits for all children
// for (int i = 0; i < num_commands; i++) {
//     waitpid(child_pids[i], &status, 0);
//     if (i == num_commands - 1) last_status = status;
// }
```

This pseudocode illustrates the iterative setup. The in_fd variable carries the read end of the pipe from one iteration to the next, becoming the standard input for the subsequent command. The parent must diligently close the write ends of pipes (pipefd) after the child that writes to it is forked, and the read ends (in_fd) once the child that reads from it has inherited it and the parent no longer needs it for the *next* iteration.
The successful implementation of pipes is a significant milestone in building a shell, as it demonstrates a core understanding of process cooperation and file descriptor manipulation.

H. Environment Variable Management in Minishell

Environment variables are a fundamental part of the Unix process model, providing a mechanism to pass configuration data and operational context to processes. A shell plays a crucial role in managing these variables, both by using them (e.g., PATH for command lookup) and by providing ways for the user to inspect and modify them. The Minishell project requires handling environment variable expansion ($VAR, $?) and implementing built-in commands like export, unset, and env.[1]

1. Accessing and Expanding Environment Variables:

- **extern char **environ;**: As discussed in Section II.F, this global variable points to the array of environment strings. It can be used directly to iterate through all environment variables, which is necessary for implementing the env built-in.[27]
- **getenv(const char *name)**: This is the standard C library function to retrieve the value of a specific environment variable name.[29] Your shell will use this when it needs to expand a variable like $HOME or $USER.
- **Expansion of $VAR**:
    - When the parser encounters an unquoted $ followed by a valid variable name (e.g., $PATH, ${USER}), or within double quotes, the shell must:
        1. Extract the variable name.
        2. Call getenv() with this name.
        3. If getenv() returns a value, replace the $VAR sequence in the command string/token with this value.
        4. If getenv() returns NULL (variable is unset), $VAR typically expands to an empty string.
- **Expansion of $?**:
    - The special variable $? expands to the exit status of the most recently executed foreground command or pipeline.[4]
    - The shell must maintain an internal variable that stores this exit status. After each foreground command completes (i.e., after waitpid() returns for an external command, or after a built-in command finishes), this internal variable is updated.

        - When $? is encountered during expansion, its value is substituted from this internal variable.

2. Implementing Built-in Commands for Environment Management:

- **export [name[=value]...]**:
    - Marks shell variables for export to the environment of subsequently executed child processes. If name=value is given, it first sets the shell variable name to value and then exports it.
    - **With arguments (e.g., export VAR=value or export VAR if VAR is already set):**
        - The shell needs to use setenv(name, value, 1) to add or modify the variable in its own environment.[30] The overwrite flag should be 1.
        - If only name is given, and name is an existing shell variable, export marks it for inclusion in environ without changing its value. If it's not an existing shell variable, the behavior can vary (bash creates it as an exported variable with no value, but its value in the environment might be empty). For Minishell, export VAR where VAR is unset might not be a required behavior, or it might mean to export it with an empty value.
    - **Without arguments (export)**:
        - The shell must list all environment variables that are currently marked for export.
        - The output format is typically declare -x NAME="VALUE" for each variable, which is itself valid shell input.[61] This involves iterating through environ or an internal list of exported variables and printing them in this format.
        - The 42 Minishell subject specifies "export without any options" [1], which usually implies this listing behavior when no arguments are given.
- **unset name...**:
    - Removes the variable name from the shell's environment.
    - Implementation uses unsetenv(name).[30]
    - The 42 subject specifies "unset without any options" [1], meaning it takes variable names as arguments.
- **env**:
    - Prints the current environment variables, one per line, in the format NAME=VALUE.
    - Implementation involves iterating through the environ array and printing each string until NULL is encountered.[55]
    - The 42 subject specifies "env without any options or arguments".[1]

3. Managing the $? Exit Status Variable:
The shell needs a mechanism to store the exit status of the last executed foreground command. This is typically an integer variable within the shell's main data structure.

- **For External Commands:**
    1. After fork(), the parent shell calls waitpid() for the child process.
    2. waitpid() returns the child's PID and fills a status integer (e.g., int child_status; waitpid(pid, &child_status, 0);).
    3. The shell then uses macros to interpret child_status [11]:

        - If WIFEXITED(child_status) is true, the command exited normally. The exit status is WEXITSTATUS(child_status).
        - If WIFSIGNALED(child_status) is true, the command was terminated by a signal. The effective exit status is often 128 + WTERMSIG(child_status). For example, if terminated by $SIGINT (signal 2), $? becomes 130.
    4. This derived value is stored as the shell's internal representation of $?.
- **For Built-in Commands:**
    1. Built-in functions (e.g., for cd, export) should be designed to return an integer status: 0 for success, non-zero for failure (e.g., 1 for a generic error, or specific codes as per POSIX for some built-ins).
    2. After the built-in function executes, the shell takes its return value and updates the internal $? variable directly.
    3. For example, cd to a non-existent directory should make $? non-zero. exit itself will terminate the shell, but if it's given an argument (e.g., exit 42), that argument becomes the shell's exit status.

The environ variable is the canonical representation of the environment that is passed to child processes via execve. While getenv reads from this effective environment, setenv and unsetenv directly modify it (or a copy that then becomes the new environ). The shell's built-ins provide a user interface to these underlying C library functions. The accurate management and expansion of $? is critical for shell scripting and conditional execution (e.g., with && and || operators in the bonus part).

I. Implementing Built-in Commands

Built-in commands are integral to a shell's functionality, providing capabilities that either cannot be implemented as external programs or are more efficiently handled internally. The Minishell project requires the implementation of a specific set of built-ins: echo, cd, pwd, export, unset, env, and exit.[1] These commands are executed directly by the shell process without creating a new child process via fork().[52]

General Implementation Strategy:

1. **Identification:** During the parsing phase, if the first token of a simple command matches a known built-in name, it's flagged for built-in execution.
2. **Direct Execution:** Instead of the fork-exec-wait sequence, the shell calls a dedicated C function that implements the built-in's logic.
3. **State Modification:** Built-ins like cd, export, unset, and exit modify the shell's own state (current directory, environment, or termination status). This is precisely why they *must* be built-in.[54]
4. **Return Status:** Each built-in implementation should return an integer status (typically 0 for success, non-zero for failure), which the shell uses to set its internal $? variable.
5. **I/O Redirection:** Built-ins, like external commands, should respect I/O redirections (>, <, >>) if specified on the command line. This means before calling the built-in's C function, the shell might need to temporarily dup2 standard output/input if redirections are present, and then restore them after the built-in completes.

Specific Built-in Implementations:

- **echo [-n][string...]**
  - **Functionality:** Prints its arguments to standard output, separated by spaces, followed by a newline.
  - **-n option:** If present as the first argument, suppresses the trailing newline.[4]
  - **Implementation:**
    - Check if the first argument is "-n".
    - Iterate through the subsequent arguments, writing each to STDOUT_FILENO using write(). Print a space between arguments.
    - If -n was not present, print a final newline character (\n).
    - Return 0.
- **cd [directory]**
  - **Functionality:** Changes the shell's current working directory.
  - **Minishell Scope:** Only relative or absolute paths are required; tilde expansion (~) or cd - might be bonus features.[4]
  - **Implementation:**
    - If no argument is given, try to change to the directory specified by the HOME environment variable (use getenv("HOME")). If HOME is not set or empty, it's an error (or specific behavior like staying in the current directory, bash prints an error).
    - If an argument is given, use it as the target path.
    - Call chdir(path).
    - If chdir() returns 0, the directory change was successful. Update any internal shell variable that might store the current path (like PWD). Return 0.
    - If chdir() returns -1, an error occurred (e.g., directory not found, permission denied). Print an error message to STDERR_FILENO using perror("cd") or a custom message. Return 1 (or another non-zero status).
    - The shell should also update the PWD environment variable. The OLDPWD environment variable handling is usually a more advanced feature.
- **pwd**
  - **Functionality:** Prints the absolute pathname of the current working directory to standard output, followed by a newline.
  - **Minishell Scope:** No options are required.[4]
  - **Implementation:**
    - Use char *getcwd(char *buf, size_t size);.
    - Allocate a buffer of sufficient size (e.g., PATH_MAX).
    - Call getcwd(buffer, PATH_MAX).
    - If successful, getcwd returns a pointer to the buffer (or buffer itself). Write the buffer contents and a newline to STDOUT_FILENO. Free the buffer if dynamically allocated by getcwd (if buf was NULL). Return 0.
    - If getcwd fails (e.g., buffer too small, or read permission denied on a parent directory), print an error message. Return 1.

- **export [name[=value]]** (Simplified for Minishell, "without any options" [1])
  - **Functionality (with arguments):** Sets an environment variable name to value and marks it for export to child processes. If value is omitted but name is given, and name is an existing shell variable, it's marked for export.
  - **Functionality (without arguments):** Lists all currently exported environment variables in a reusable format.
  - **Implementation (with arguments name=value or name):**
    - Parse name and value (if name=value form).
    - Use setenv(name, value, 1) to set/update the variable in the environment. The 1 ensures overwriting if it exists.
    - Return 0 on success, 1 on failure (e.g., invalid variable name).
  - **Implementation (without arguments):**
    - Iterate through the environ array (extern char **environ;).
    - For each string (e.g., VAR=value), print it in the format declare -x VAR="value" (or a simpler VAR=value if declare -x is too complex for the initial version, though bash uses declare -x).[61]
    - Return 0.
- **unset name**
  - **Functionality:** Removes the environment variable name.
  - **Minishell Scope:** No options required.[4]
  - **Implementation:**
    - For each name argument provided:
      - Call unsetenv(name).
    - Return 0 (even if a variable didn't exist, unsetenv often succeeds).
- **env**
  - **Functionality:** Prints the current environment variables.
  - **Minishell Scope:** No options or arguments.[4]
  - **Implementation:**
    - Iterate through extern char **environ;.
    - For each non-NULL string, write it to STDOUT_FILENO, followed by a newline.
    - Return 0.
- **exit [n]**
  - **Functionality:** Causes the shell to exit.
  - **Minishell Scope:** No options. An optional numeric argument n can be the exit status.
  - **Implementation:**
    - If an argument n is provided, parse it as an integer. This integer will be the exit status of the shell (modulo 256).
    - If no argument is provided, the exit status is that of the last command executed (the current value of $?).
    - Perform any necessary cleanup (e.g., free allocated memory, clear readline

        history if managing it).
    - Call exit(status_code);. This function does not return.

The reason these commands are built-in is primarily because they need to affect the shell's own execution environment or state. An external command runs in a child process, and a child process cannot change its parent's current working directory, environment variables, or cause the parent to terminate.[54] Thus, cd, export, unset, and exit must be part of the shell itself. Others like pwd and echo could theoretically be external, but are often built-in for efficiency and to ensure consistent behavior without relying on external files being present or having correct permissions.

J. Signal Handling Implementation

Proper signal handling is crucial for a shell to behave predictably and interactively, especially in response to user actions like pressing Ctrl-C, Ctrl-D, or Ctrl-\. The Minishell project requires specific behaviors for these signals, akin to bash.[1] The sigaction() system call is the preferred and allowed method for setting up signal handlers.[1]

Required Signal Behaviors (Interactive Mode):

- **Ctrl-C ($SIGINT):**
    - **Behavior:** Display a new prompt on a new line. The currently typed command line should be cleared or abandoned. If a child process is running in the foreground, $SIGINT should typically be sent to that child process instead of being handled by the shell itself (though the shell might then also redisplay its prompt). The 42 subject states "displays a new prompt on a new line" [4], which primarily refers to the shell's own handling when it's waiting for input.
    - **Implementation:**
        - Install a handler for $SIGINT using sigaction().
        - Inside the handler:
            - Write a newline character to standard output/error.
            - Call rl_on_new_line() to tell Readline the cursor is on a new line.
            - Call rl_replace_line("", 0) to clear the current Readline buffer.
            - Call rl_redisplay() to refresh the prompt and empty line.
            - Set the shell's internal $? variable to an appropriate value (e.g., 130 for $SIGINT).
        - **Important Consideration:** The shell needs to differentiate between when it should handle $SIGINT itself (e.g., when waiting for input at the prompt) and when a child process should handle it (e.g., when a command is running in the foreground).
            - When no child is running in the foreground, the shell's handler acts as described.
            - When a child is running, the shell might temporarily ignore $SIGINT or set up a handler that does nothing, allowing the signal to propagate to the foreground child process group. The child process will then terminate by default or handle it as programmed. After the child

terminates due to $SIGINT, the parent shell (after waitpid) would then update $? and redisplay the prompt.

- **Ctrl-D (EOF - End of File):**
  - **Behavior:** Exit the shell. This typically happens when readline() returns NULL, indicating EOF was received (often when Ctrl-D is pressed on an empty line).
  - **Implementation:**
    - When readline() in the main loop returns NULL:
      - Print "exit" (to mimic bash behavior).
      - Perform cleanup (free memory, etc.).
      - Call exit(last_status_code); where last_status_code is the current value of $?.
- **Ctrl-\ ($SIGQUIT):**
  - **Behavior:** Do nothing in the interactive shell.[4] If a child process is running, $SIGQUIT should be sent to it, typically causing it to terminate and dump core.
  - **Implementation:**
    - Install a handler for $SIGQUIT using sigaction() that is set to $SIG_IGN (ignore the signal) for the shell itself when it's interactive and no foreground child is running.
    - When a child process is forked to run a command, the signal disposition for $SIGQUIT in the child should ideally be reset to $SIG_DFL (default action, which is terminate and core dump) before execve is called. This allows Ctrl-\ to quit running commands.

General sigaction() Setup:

C

```
#include <signal.h>
#include <stdio.h>
#include <readline/readline.h> // For rl_on_new_line, etc.
#include <stdlib.h> // For exit()

// Global variable to store exit status (as per 42's one global var rule, if used for this)
// Or, more commonly, part of a shell_state struct.
// For simplicity here, let's assume a mechanism to update $? exists.
// int g_last_exit_status = 0;

void handle_sigint(int signum) {
  (void)signum; // Unused parameter
  // Set $? to 130 (128 + SIGINT)
  // update_exit_status(130); // Hypothetical function
  write(STDOUT_FILENO, "\n", 1);     // Newline
  rl_on_new_line();              // Tell Readline we're on a new line
```

```
    rl_replace_line("", 0);          // Clear current input buffer
    rl_redisplay();                  // Redisplay the prompt and cleared line
}

void setup_signal_handlers(void) {
    struct sigaction sa_int;
    struct sigaction sa_quit;

    // Handler for SIGINT (Ctrl-C)
    sa_int.sa_handler = handle_sigint;
    sigemptyset(&sa_int.sa_mask); // Do not block any other signals during handler
    sa_int.sa_flags = SA_RESTART; // Restart interrupted system calls if possible
    if (sigaction(SIGINT, &sa_int, NULL) == -1) {
        perror("sigaction SIGINT failed");
        exit(EXIT_FAILURE);
    }

    // Handler for SIGQUIT (Ctrl-\) - ignore it
    sa_quit.sa_handler = SIG_IGN;
    sigemptyset(&sa_quit.sa_mask);
    sa_quit.sa_flags = 0;
    if (sigaction(SIGQUIT, &sa_quit, NULL) == -1) {
        perror("sigaction SIGQUIT failed");
        exit(EXIT_FAILURE);
    }
}
```

Interaction with Child Processes:

- When the shell forks a child to run an external command:
    - The child process should typically reset its signal handlers for $SIGINT and $SIGQUIT to $SIG_DFL (default action) before calling execve(). This ensures that Ctrl-C and Ctrl-\ will terminate the child command as expected, rather than being ignored or handled by the shell's own handlers (which the child would otherwise inherit).
    - The parent shell, while waitpid() is waiting for a foreground child, might need to temporarily ignore $SIGINT and $SIGQUIT itself, or have handlers that simply record the signal and allow waitpid to be interrupted, so that the signals are effectively passed to the child. If the child terminates due to a signal, waitpid will report this, and the parent shell can then set $? accordingly (e.g., 130 for $SIGINT, 131 for $SIGQUIT).

Interaction with Readline:

Signal handlers, especially for $SIGINT, can interrupt readline(). The Readline library provides functions like rl_on_new_line(), rl_replace_line(), and rl_redisplay() that are essential for

correctly resetting the prompt and input line after a signal.34 Using these functions helps ensure that the terminal state remains consistent. The SA_RESTART flag in sigaction is also important, as it can help readline (which internally uses system calls like read) to recover from interruptions.
Careful management of signal dispositions in both the parent shell and its child processes is key to achieving the standard interactive shell behavior.

K. Error Handling and Exit Status ($?)

Robust error handling is a hallmark of a well-written shell. This involves detecting errors from system calls and internal operations, reporting them informatively to the user, and setting an appropriate exit status ($?) that reflects the outcome of commands.

1. Detecting and Reporting Errors from System Calls:

- **Check Return Values:** Most system calls and many C library functions return a special value (e.g., -1 for integers, NULL for pointers) to indicate an error. Always check these return values.
- **errno Global Variable:** When a system call fails, it usually sets the global integer variable errno (declared in <errno.h>) to a positive value indicating the specific error.[63] errno is only meaningful immediately after a call fails; subsequent successful calls or other library calls might reset or change it.
- **perror(const char *s)** [63]:
    - Declared in <stdio.h>.
    - Prints the string s (if not NULL), followed by a colon, a space, and a human-readable error message corresponding to the current value of errno. Output is to STDERR_FILENO.
    - Example: if (open("file", O_RDONLY) == -1) { perror("minishell: open failed"); }
- **char *strerror(int errnum)** [63]:
    - Declared in <string.h>.
    - Returns a pointer to a string containing the human-readable error message for the error number errnum (usually, you pass errno here).
    - Allows for more customized error messages.
    - Example: fprintf(stderr, "minishell: Error accessing %s: %s\n", filename, strerror(errno));
    - The string returned by strerror might be static and overwritten by subsequent calls, so use it immediately or copy it. strerror_r is a thread-safe alternative.

2. Setting the Shell's Exit Status Variable ($?):
The special shell variable $? must always reflect the exit status of the most recently executed foreground command or pipeline.4 Your Minishell needs an internal integer variable to store this status.

- **For External Commands:**
    1. After a child process is forked and execved, the parent shell calls waitpid() to wait for its termination.
    2. waitpid(pid, &status, 0) stores the child's termination status in the status integer.

    3. This status is then decoded [11]:
        - If WIFEXITED(status) is true: The child exited normally. The exit code is WEXITSTATUS(status). This value (0-255) becomes $?.
        - If WIFSIGNALED(status) is true: The child was terminated by a signal. The value for $? is conventionally 128 + WTERMSIG(status). For example, if killed by $SIGINT (signal 2), $? becomes 130.
    4. This calculated value is stored in the shell's internal variable representing $?.
- **For Built-in Commands:**
    1. Each C function implementing a built-in command should return an integer: 0 for success, and a non-zero value (typically 1 for general errors, or specific codes if applicable, e.g., cd failing might return 1) for failure.
    2. The shell directly uses this return value to update its internal $? variable.
    3. Example: If cd /nonexistent fails, the cd built-in function returns 1, and $? becomes 1. If echo hello succeeds, it returns 0, and $? becomes 0.
- **For Syntax Errors or Other Shell Errors:**
    - If the shell itself encounters an error during parsing (e.g., syntax error like an unexpected token, bad redirection) or execution setup (e.g., pipe fails before forking), it should print an error message to STDERR_FILENO.
    - The $? variable should be set to a non-zero value, often 2 for syntax errors or general shell errors, or 127 if a command is not found during PATH lookup (as bash often does). The exact values for specific shell errors can be guided by bash behavior.

Example of setting $? after waitpid:

C

```c
// In parent shell, after forking child_pid
// int child_actual_status;
// int shell_exit_status; // This will be our internal $?

// waitpid(child_pid, &child_actual_status, 0);

// if (WIFEXITED(child_actual_status)) {
//     shell_exit_status = WEXITSTATUS(child_actual_status);
// } else if (WIFSIGNALED(child_actual_status)) {
//     shell_exit_status = 128 + WTERMSIG(child_actual_status);
// }
// update_internal_dollar_question(shell_exit_status); // Update shell's $?
```

Consistent and accurate error reporting and exit status management are vital for a shell to be usable, especially for scripting, where the outcome of one command often determines the execution of subsequent commands (e.g., using && and || in the bonus part).

IV. Advanced Features (Bonus Part)

The 42 Minishell project often includes optional bonus features that allow students to explore more complex shell functionalities. Successfully implementing these bonuses typically requires a solid and extensible architecture from the mandatory part. Key bonus features often include logical operators (&&, ||) and wildcard expansion (globbing).[1] Job control is a more ambitious bonus that delves deeper into process group management.

A. Implementing Logical Operators (&& and ||)

- **Functionality:**
  - command1 && command2: command2 is executed if and only if command1 terminates with an exit status of 0 (success).
  - command1 | | command2: command2 is executed if and only if command1 terminates with a non-zero exit status (failure).
- **Parsing Considerations:**
  - The parser must be able to recognize && and || as distinct operators that separate commands or pipelines.
  - This often necessitates a more sophisticated parsing strategy than a simple linear list of commands. An Abstract Syntax Tree (AST) is a common way to represent the command structure, where && and || become internal nodes with left and right children representing the commands/pipelines they connect.[2] explicitly mentions one team's use of an asymmetric binary tree for this.
- **Execution Logic:**
  - The executor module must traverse the AST (or other structure).
  - When an && node is encountered:
    1. Execute the left child (command/pipeline).
    2. Check its exit status ($?).
    3. If the status is 0, proceed to execute the right child. The overall status of the && expression will be the status of the right child if executed, or the status of the left child if the right child was skipped.
    4. If the status is non-zero, skip the right child. The overall status is that of the left child.
  - When an || node is encountered

1. Minishell: As Beautiful As A Shell | PDF - Scribd, https://www.scribd.com/document/537050499/minishell
2. Minishell: A lightweight custom Unix shell project for 42 School. - GitHub, https://github.com/MarkosComK/42-Minishell
3. Minishell | Guide, https://42-cursus.gitbook.io/guide/rank-03/minishell
4. 42-Student-Teams/42-Minishell - GitHub, https://github.com/42-Student-Teams/42-Minishell
5. madebypixel02 / minishell - GitLab, https://gitlab.com/madebypixel02/minishell
6. Unix Opearting System - Architecture and Its Properties - ElProCus, https://www.elprocus.com/unix-architecture-and-properties/

7. What is Unix?Unix Architecture - MetEd - University Corporation for Atmospheric Research, https://www.meted.ucar.edu/ucar/unix/navmenu.php?tab=1&page=2-1-0&type=flash
8. minishell-42 : r/42_school - Reddit, https://www.reddit.com/r/42_school/comments/1fcljcr/minishell42/
9. fork() , exec() , wait() , and errno - GitHub, https://github.com/SFU-CMPT-201/lecture-notes/blob/main/02-fork-exec-wait.md
10. wait, https://www.opengroup.org/onlinepubs/007908775/xsh/wait.html
11. Exit status of a child process in Linux | GeeksforGeeks, https://www.geeksforgeeks.org/exit-status-child-process-linux/
12. Job control (Unix) - Wikipedia, https://en.wikipedia.org/wiki/Job_control_(Unix)
13. www.cs.cmu.edu, https://www.cs.cmu.edu/afs/cs/academic/class/15213-f22/www/activities/system-io-soln.pdf
14. File I/O, https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/fileio.html
15. Shell 1 (C-Shell) - Brown Computer Science, https://cs.brown.edu/courses/cs033/shell1/
16. pipe() System Call in Operating System- Scaler Topics, https://www.scaler.com/topics/pipes-in-os/
17. pipe() System call | GeeksforGeeks, https://www.geeksforgeeks.org/pipe-system-call/
18. Signal Handling In Linux Through The signal() Function ..., https://www.geeksforgeeks.org/signal-handling-in-linux-through-the-signal-function/
19. sigaction() — Examine or change a signal action - IBM, https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-sigaction-examine-change-signal-action
20. nenieiri-42Yerevan/Minishell: This is a project about creating a simple shell (Mini bash) - GitHub, https://github.com/nenieiri-42Yerevan/Minishell
21. Signal and Sigaction (The GNU C Library), http://www.gnu.org/s/libc/manual/html_node/Signal-and-Sigaction.html
22. sigaction() System Call in Linux - Tutorialspoint, https://www.tutorialspoint.com/unix_system_calls/sigaction.htm
23. sigaction(2) - Linux manual page - man7.org, https://www.man7.org/linux/man-pages/man2/sigaction.2.html
24. 07 04 Linux Sigaction System Call Programming - YouTube, https://www.youtube.com/watch?v=_1TuZUbCnX0
25. GETENV - NV5 Geospatial Software, https://www.nv5geospatialsoftware.com/docs/GETENV.html
26. environ(7) — Arch manual pages, https://man.archlinux.org/man/environ.7.en
27. CMPSC 311, Environment Variables,

https://www.cse.psu.edu/~deh25/cmpsc311/Lectures/Environment-Variables/Environment-Variables-2.html
28. How do I access environment variables in C? - Jim Fisher,‹ https://jameshfisher.com/2017/02/02/how-do-i-access-environment-variables-in-c/
29. getenv,‹ http://www.opengroup.org/onlinepubs/009695399/functions/getenv.html
30. getenv(3) - OpenBSD manual pages,‹ https://man.openbsd.org/OpenBSD-5.6/getenv.3
31. setenv(3) - FreeBSD Manual Pages,‹ https://man.freebsd.org/cgi/man.cgi?query=setenv&sektion=3
32. unsetenv(3) - Arch Linux manual pages,‹ https://man.archlinux.org/man/unsetenv.3.en
33. opensolaris unsetenv man page on unix.com,‹ https://www.unix.com/man_page/opensolaris/3c/unsetenv
34. readline — GNU readline interface — Python 3.13.3 documentation,‹ https://docs.python.org/3/library/readline.html
35. The GNU Readline Library,‹ https://tiswww.case.edu/php/chet/readline/rltop.html
36. Tab completion in GNU Readline - Thoughtbot,‹ https://thoughtbot.com/blog/tab-completion-in-gnu-readline
37. How to manage your Linux command history - Red Hat,‹ https://www.redhat.com/en/blog/history-command
38. How To Use Bash History Commands and Expansions on a Linux VPS | DigitalOcean,‹ https://www.digitalocean.com/community/tutorials/how-to-use-bash-history-commands-and-expansions-on-a-linux-vps
39. Bash history features for Linux server management - Liquid Web,‹ https://www.liquidweb.com/blog/bash-history/
40. How to Set an Unlimited Bash History | Baeldung on Linux,‹ https://www.baeldung.com/linux/bash-configure-unlimited-history
41. shlex — Simple lexical analysis — Python 3.13.3 documentation,‹ https://docs.python.org/3/library/shlex.html
42. Developing a Linux based shell | GeeksforGeeks,‹ https://www.geeksforgeeks.org/developing-linux-based-shell/
43. String Tokenization in C | GeeksforGeeks,‹ https://www.geeksforgeeks.org/string-tokenization-in-c/
44. How to Use `strtok` and `strtok_r` in C - Systems Encyclopedia,‹ https://systems-encyclopedia.cs.illinois.edu/articles/c-strtok/
45. strtok_r - CS50 Manual Pages,‹ https://manual.cs50.io/3/strtok_r
46. Section 10: Command line representation – CS 61 2021,‹ https://cs61.seas.harvard.edu/site/2021/Section10/
47. Concepts and Guidance on a Minishell Project in C : r/C_Programming,‹ https://www.reddit.com/r/C_Programming/comments/1jq0qx6/concepts_and_guidance_on_a_minishell_project_in_c/
48. CST8207-19W – GNU/Linux Operating Systems I,‹ https://teaching.idallen.com/cst8207/15w/notes/440_quotes.html

49. Special Characters and Quoting - Learning the bash Shell, Second ...,، https://www.oreilly.com/library/view/learning-the-bash/1565923472/ch01s09.html
50. Uwe Waldmann - A Guide to Unix Shell Quoting - Teaching,، https://rg1-teaching.mpi-inf.mpg.de/unixffb-ss98/quoting-guide.html
51. Bash Reference Manual,، https://stuff.mit.edu/afs/sipb/project/linerva/no-backup/root/bash-2.05b-2/build-bash/doc/bashref.pdf
52. Bash Reference Manual - GNU,، https://www.gnu.org/software/bash/manual/bash
53. Day 2 Linux: What are shell built-in commands? - thiru's blog,، https://tirucloud.hashnode.dev/day-2-linux-what-are-shell-built-in-commands
54. [C] After implementing my own shell, using fork() and exec(), why do all commands work except for cd? - Reddit,، https://www.reddit.com/r/learnprogramming/comments/2cdq5w/c_after_implementing_my_own_shell_using_fork_and/
55. www.cs.purdue.edu,، https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/Book/Chapter5-WritingYourOwnShell.pdf
56. تم الوصول بتاريخ 1970 ,1 يناير، https://unix.stackexchange.com/questions/120821/why-are-cd-export-and-other-commands-built-in
57. Shells Use Temp Files to Implement Here Documents - Oil Shell,، https://www.oilshell.org/blog/2016/10/18.html
58. How create a temporary file in shell script? - Unix & Linux Stack Exchange,، https://unix.stackexchange.com/questions/181937/how-create-a-temporary-file-in-shell-script
59. Exit Status($?) variable in Linux | GeeksforGeeks,، https://www.geeksforgeeks.org/exit-status-variable-in-linux/
60. Shell Command Language,، https://pubs.opengroup.org/onlinepubs/009696599/utilities/xcu_chap02.html
61. The Set Builtin (Bash Reference Manual) - GNU,، https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html
62. Bash Builtins (Bash Reference Manual) - GNU,، https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html
63. Error Handling in C | GeeksforGeeks,، https://www.geeksforgeeks.org/error-handling-in-c/
64. C Error Handling - Tutorialspoint,، https://www.tutorialspoint.com/cprogramming/c_error_handling.htm
65. Error Messages (The GNU C Library),، http://www.gnu.org/software/libc/manual/2.32/html_node/Error-Messages.html
66. Error Messages (The GNU C Library),، https://www.gnu.org/s/libc/manual/html_node/Error-Messages.html
67. Error Messages (The GNU C Library),، https://www.gnu.org/software/libc/manual/html_node/Error-Messages.html