Assignment 3

A Miniature Unix Shell with N-Stage Pipelines and a Shared-Memory Session Ledger

Five system calls, eight pipeline stages, and nowhere for a leaked descriptor to hide.

Version 1.0 October 14, 2026

Version history appears in Section 20.

Contents

1 Overview . . . 2
2 Learning Objectives . . . 3
3 Assignment Structure . . . 4
4 Where to Start . . . 4
5 Part-1: The Read–Parse–Execute Loop . . . 6
6 Part-2: N-Stage Pipelines . . . 8
7 Part-3: Shared-Memory Session Ledger . . . 11
8 Three Invariants That Explain Every Bug in This Assignment . . . 13
9 Part-4: File-Descriptor Forensics . . . 14
10 Personalized Verification . . . 15
11 Required Use of Claude Code . . . 15
12 Deliverables . . . 16
13 Command-Line and Output Contract . . . 17
14 Grading . . . 17
15 How We Grade Your Claude Code Prompt Logs . . . 19
16 Sample Test Cases (Autograder Preview) . . . 20
17 Late Submission . . . 21
18 The spirit of this assignment . . . 21
19 How AI Was Used in Developing This Assignment . . . 21
20 Version History . . . 22
A The Published Header: `ramsh.h` . . . 23
B The Starter Makefile . . . 26
C How the Autograder Works . . . 27

1 Overview

A shell is a very small program with a very large reputation. Strip away history, tab completion, and forty years of accumulated syntax, and what remains is a loop: read a line, split it into programs, launch those programs with `fork()` and `exec()`, connect them with `pipe()`, and collect their exit statuses with `waitpid()`. In this assignment you will build that loop yourself, and then build a second capability alongside it: a *session ledger*, kept in System V shared memory, that records every pipeline your shell runs and survives to be read by a completely separate program.

There is some history hiding inside that small loop. Early Unix already had a program called `sh`, and a Unix programmer from the 1970s would recognize much of the process lifecycle you are about to implement. The machinery around shells has become vastly more elaborate; the underlying relationship among process creation, program replacement, descriptors, and waiting has proved remarkably durable. That is part of what makes this project interesting: you are not simulating an operating-systems abstraction. You are using Linux's actual process and IPC machinery to rebuild a small piece of an architecture that has survived for more than half a century.

Your shell is called `ramsh`.[1] It supports pipelines of up to eight stages (`seq 100000 | grep 7 | wc -l`), a small set of builtins, and a companion viewer program, `ramview`, that attaches to the ledger segment and prints its contents. Everything in this assignment traces back to five system calls: `fork`, `exec`, `waitpid`, `pipe`, and the `shmget` family. There are no threads, no semaphores, and no locks anywhere in this assignment, and Section 7.4 explains why their absence is a design decision you must be able to defend rather than an oversight.

The core theme of this assignment is *bookkeeping*. Calling `pipe()` is one line of code. Knowing which of the resulting descriptors must be closed, in which processes, and being able to predict precisely what breaks when a single `close()` is forgotten, is the actual content of this assignment. It is fairly straightforward to write a two-stage pipeline that appears to work. The harder and more important skill, and the one this assignment is actually grading, is writing a general N-stage executor whose descriptor hygiene you can prove, and then demonstrating that proof empirically in Part 4 by deliberately breaking it.

This is also where several ideas from Lectures 3–6 finally become one system rather than a collection of system calls. `fork()` creates the processes; `exec()` gives them new identities; `pipe()` gives them communication paths; `waitpid()` lets the parent observe their lifecycles; and shared memory gives a separate process access to state that outlives one execution step. A shell is small enough that you can still hold the architecture in your head, but rich enough that vague reasoning about ownership, inheritance, or cleanup becomes visible almost immediately. Operating systems can be rather unsentimental about vague reasoning.

[1] A portmanteau of our University mascot (RAM) and `sh`.

Grading methodology and the Quiz that you can't Google

This assignment is graded out of **15 points**, and those points answer two different questions, calling for two different kinds of evidence.
9 points (60%) answer the question a compiler already asks: *does the program work*? That question is settled for you, mechanically and without opinion, by the autograder described in Appendix C.
The remaining **6 points (40%)** are the *process-engineering* component, and they answer a harder question: *do you understand what you built, and did you build it?* **2 points** come from an analysis of your Claude Code prompt transcripts: whether they show the short, specific, iterative exchanges of someone building and debugging their own program, or the single large paste of someone outsourcing it. **4 points** come from a short quiz generated from your own code: your own function names, your own design choices, the changes that might introduce bugs, etc. Two students can submit shells that behave identically on every test case and still walk away from that quiz with very different scores, because the quiz was never testing the shell. It was testing you.
There is a reason for the split. A working three-stage pipeline is a nice thing to have. A student who can explain, unprompted, why the final stage of that pipeline hangs forever when one forgotten `close()` survives in the parent is a nice thing to become. Most of the hours you put into this assignment build the former. The prompt analysis and the quiz at the end are where we find out whether you also built the latter.

2 Learning Objectives

- Drive the complete `fork()`/`exec()`/`waitpid()` lifecycle of a child process, including exit-status and signal-death decoding with the `WIFEXITED`/`WIFSIGNALED` macro family.
- Construct pipelines of arbitrary depth with `pipe()` and `dup2()`, and reason precisely about file-descriptor inheritance, duplication, and end-of-file semantics.
- Predict, induce, and diagnose the failure modes of incorrect descriptor bookkeeping: the hung reader, the deadlocked writer, and the leaked descriptor visible in a child's own `/proc/self/fd`.
- Manage the full lifecycle of a System V shared memory segment (`shmget`, `shmat`, `shmdt`, `shmctl`) across a `fork()`/`exec()` boundary into an unrelated program.
- Design a shared data structure that is correct *by construction* under a single-writer discipline, and articulate exactly which assumption a future concurrent version would break.
- Design and defend a C program's interface: struct layout for shared memory, memory ownership, and a stable command-line and output contract that another program (the autograder) depends on.
- Use Claude Code productively on a project where the code is short but the reasoning is not, and where verifying descriptor hygiene, not producing code, is the hard part.

Item	Detail
Language	C23 (`-std=c23`), compiled with GCC 15 from `gcc-toolset-15`; must build cleanly with `-Wall -Wextra -Werror` (Section 4.1, Appendix B)
Platform	AlmaLinux 9.8 (Olive Jaguar): Linux kernel 5.14, glibc 2.34. All grading happens on this platform.
Starter files	`ramsh.h` (Appendix A), `Makefile` (Appendix B), and the public autograder tests (Appendix C)
Expected effort	Approximately 30 hours, including required use of Claude Code
Tooling	Claude Code is required for this assignment (see Section 11)

3 Assignment Structure

The assignment has four parts, and all four are required. The hour estimates for each of these elements are included in the table below. Section 4 tells you where to start.

Part	Title	Est. hours
1	The read–parse–execute loop and builtins	6–7
2	N-stage pipelines	9–10
3	Shared-memory session ledger and the `ramview` viewer	7–8
4	File-descriptor forensics lab	4–5
—	Design document, forensics report, and prompt log (written throughout, not at the end)	2–3

4 Where to Start

Four parts, two programs, and a shared-memory segment is a lot to hold in your head on the first day. This section tells you what you are given, where each piece of code goes, and an order of work in which every step can be run and checked before you build the next one.

There is something pleasingly direct about this architecture. The shell itself does surprisingly little privileged or mysterious work. It asks the kernel to create a process, arranges that process's environment, replaces its program image, and waits for the result. The program eventually running as `grep`, `sort`, or `wc` does not need to know that `ramsh` exists at all. That separation is one of Unix's enduring tricks: sophisticated compositions can emerge from programs that know almost nothing about one another.

4.1 Platform: AlmaLinux 9.8

Your code is graded on AlmaLinux 9.8 (Olive Jaguar), the operating system on the department's lab machines: Linux kernel 5.14 and glibc 2.34. Three facts about this platform will save you time.

- **The default compiler is too old for C23.** `/usr/bin/gcc` on AlmaLinux 9 is GCC 11, which rejects `-std=c23` (that flag arrived in GCC 14). Use GCC 15 from `gcc-toolset-15` by running

  ```
  source /opt/rh/gcc-toolset-15/enable
  ```

  in each new shell, or once in your `~/.bashrc`. After that, `gcc --version` reports 15. If you forget, the starter Makefile stops and prints this instruction.

- **Strict C23 hides POSIX.** `-std=c23` asks for ISO C and nothing else, so glibc hides POSIX declarations such as `getline`, `kill`, `clock_gettime`, and `CLOCK_MONOTONIC`. The Makefile passes

`-D_POSIX_C_SOURCE=200809L` to bring them back, and `ramsh.h` refuses to compile without it. Do not switch to `-std=gnu23` to make an error go away; the autograder builds with the flags in Appendix B, whatever your Makefile says.

- **Test where you are graded.** macOS has no `/proc`, and its pipes and System V limits behave differently. Ubuntu and WSL are close, but they ship a newer compiler and glibc, and under `-Werror` a different compiler can fail a build over a different warning. Write code wherever you like, but build and run the public tests on a lab machine before every submission. Everything this assignment uses (`valgrind`, `strace`, `ps`, `ipcs`, `ipcrm`) is packaged for AlmaLinux; on your own AlmaLinux 9 machine, `sudo dnf install gcc-toolset-15-gcc valgrind strace` adds what a minimal install lacks.

4.2 What you are given

- **`ramsh.h`** (Appendix A). Every constant, type, message format, and function signature the autograder relies on, with a comment on each function stating its contract. Read it before you write any code: it is the specification, written in C. Do not edit it. The autograder copies the published version over yours before it builds, so put declarations of your own in a separate header.
- **`Makefile`** (Appendix B). Builds `ramsh`, `ramview`, and `libramsh.a`, a static library holding every function declared in `ramsh.h`. The autograder links its own unit tests against `libramsh.a`. That is why no function declared in `ramsh.h` may live in the same file as a `main()`.
- **The public autograder tests** (Appendix C). `python3 grade.py --public .` runs them from your repository root, exactly as the grader will.

The file layout below is a suggestion. You may divide the code differently; the rule that matters is the last column.

File	Holds	In `libramsh.a`?
`parse.c`	`parse_line`, `parse_errmsg`, `free_pipeline`	yes
`builtins.c`	`is_builtin`, `run_builtin`	yes
`exec.c`	`decode_status`, `leak_target`, `run_pipeline`	yes
`ledger.c`	`ledger_create`, `ledger_attach`, `ledger_record`, `ledger_print`, `ledger_detach`, `ledger_destroy`	yes
`main.c`	`ramsh`'s `main()`: the read–parse–execute loop, `-c` mode, and the single cleanup path	no
`ramview.c`	`ramview`'s `main()`: attach, print, detach	no

4.3 A build order that works

Each step ends with something you can run. Commit when it works: small commits are good engineering, and they are also what the prompt-log grading in Section 15 checks your transcripts against.

Step	Build	Check it by hand	Tests
1	`parse_line`, `parse_errmsg`, `free_pipeline`, `is_builtin`	A throwaway `main()` that prints each stage's `argv`; clean under `valgrind`	P1.1–P1.2
2	`decode_status`; `run_pipeline` for one stage; `main()` with `-c`	`./ramsh -c "echo hello"`; `./ramsh -c nosuchprog; echo $?` prints 127	P1.3–P1.6
3	The interactive loop; `cd` and `exit`; one cleanup path for `exit`, end-of-file, and `-c`	`cd` then `pwd`; Ctrl-D exits with status 0	P1.7–P1.8
4	Two stages first, then the general N-stage loop	`seq 1000000 | wc -l`, `ls /proc/self/fd`, `yes | head -n 1`	P2.1–P2.8
5	The ledger functions, `ramview`, the `ledger` builtin	`ipcs -m` is empty after `exit`	P3.1–P3.7
6	`leak_target` and `RAMSH_LEAK`; the `forensics.md` experiments	`RAMSH_LEAK=0 timeout 5 ./ramsh -c "echo done | wc -l"`	P4.1–P4.3
7	Memory, zombies, and the shell's own descriptors	The public tests	Q1–Q3

Step 4 is where most of the 30 hours go. Resist writing the N-stage loop first: a two-stage version you understand completely is the fastest route to a general one.

5 Part-1: The Read–Parse–Execute Loop

`ramsh` runs a loop: print the prompt `ramsh>` (note the trailing space), read one line from `stdin`, parse it, execute it, repeat. The loop ends when the user types `exit` or when `stdin` reaches end-of-file (Ctrl-D). In both cases the shell must clean up its shared-memory segment (Section 7) before terminating with status 0.

The prompt goes to stderr, and this is not a typo. Everything `ramsh` says about itself, including the prompt, the fork/reap trace lines defined below, and all error messages, is written to `stderr`. `stdout` belongs exclusively to the programs your shell runs. This is the same contract `bash` honors, and it is what makes a shell *composable*: it is the reason `./ramsh -c "seq 5" | wc -l` can print `5` instead of a number polluted by your prompt. The autograder depends on this separation absolutely; a single stray `printf` to `stdout` will fail otherwise-correct submissions.

5.1 Parsing rules

Tokens are separated by spaces and tabs. The pipe character `|` is recognized only when it appears as a standalone token; `a|b` is a single (doomed) token, not a pipeline, and your parser is not required to rescue it. There is no quoting, no globbing, no environment-variable expansion, no `&&`, no background `&`, and no redirection (redirection is available as extra credit; see Section 6.4). Input lines may be up to 1023 characters. A line that is empty or contains only whitespace produces no output and no ledger entry; the shell simply prompts again.

The following are syntax errors, must execute nothing, must record nothing in the ledger, and must produce exactly the indicated message on `stderr`. When a line has more than one of them, the first one in this list is reported.

- A pipeline with an empty stage (a leading, trailing, or doubled `|`): `ramsh: syntax error near '|'`

- More than 8 stages: `ramsh: too many stages (max 8)`
- A builtin appearing as the command of any stage in a multi-stage pipeline: `ramsh: builtins cannot appear in a pipeline`

5.2 Builtins

Builtin	Forks?	Behavior
`cd [dir]`	No	`chdir()` to `dir`; with no argument, to `$HOME`. On failure: `ramsh: cd: <strerror>` on stderr.
`exit`	No	Detach and destroy the ledger segment, then terminate with status 0.
`ledger`	Yes	Fork and `execvp()` the viewer: `ramview <shmid>`. Wait for it. Never recorded in the ledger itself.

Table 1: Three builtins, one distinction. `cd` and `exit` *must not* fork, because a `chdir()` or an `exit()` performed in a child changes the child and then evaporates with it. `ledger` deliberately *must* fork, because its entire purpose is to prove the segment is readable from a different program (Section 7). Be prepared to explain both directions of this distinction; it is a standing quiz favorite.

5.3 Running a single external command

A one-stage pipeline is executed with one `fork()`, one `execvp()`, and one `waitpid()`. The shell prints a trace of this lifecycle to `stderr` in exactly this format (one line at fork time, one at reap time):

```
ramsh: stage 0 [pid 41213] echo
ramsh: stage 0 [pid 41213] exited 0
```

If the child dies from a signal, the reap line is instead `ramsh: stage 0 [pid 41213] killed by signal <N>`, and the recorded status (Section 7) is $128 + N$, matching the convention every real shell uses.

The exec-failure protocol. If `execvp()` fails (the program does not exist, is not executable, etc.), the child must print `ramsh: exec failed for <argv0>: <strerror>` to `stderr` and then call `_exit(127)`. Two details here are load-bearing, and both are quiz material. First, the status must be 127, the universal shell convention for "command not found." Second, the call must be `_exit()`, with the underscore, not `exit()`. The child inherited a copy of the parent's `stdio` buffers at `fork()` time, and `exit()` flushes them as if they were the child's own. For an output buffer, that prints data the parent will print again. In `ramsh` the buffer that bites is `stdin`'s. When the shell reads its commands from a file (`./ramsh < script.txt`), `exit()` in the child hands the unread part of its copy of the input buffer back to the file by moving the file offset backward, and that offset is shared with the parent (Invariant 1, Section 8). The parent's next `read()` then starts from the old position, and the shell runs the rest of the script a second time. The autograder tests exactly this (test P1.5). The one-character difference between a correct shell and a shell that occasionally does everything twice is worth remembering.

5.4 One-shot mode

`./ramsh -c "<line>"` parses and executes exactly one line, records it in the ledger, destroys the ledger, and exits with the **exit status of the last stage** (so `./ramsh -c "nosuchprog"` exits 127, and `./ramsh -c "nosuchprog | wc -l"` exits 0, because `wc` succeeded). A line with a syntax error exits 2, the status `bash` uses. A builtin exits with its own status: 0 for `exit`, 0 or 1 for `cd`, and `ramview`'s status for `ledger`. A blank line exits 0. The autograder uses this mode heavily, and so should you.

5.5 Required interface

Your parser and builtins implement the Part 1 section of `ramsh.h` (Appendix A). The autograder calls these functions directly from its unit tests, so the contracts in the header comments are graded, not just the shell's visible behavior.

```
parse_result_t parse_line(const char *line, pipeline_t *p);
const char    *parse_errmsg(parse_result_t rc);
void           free_pipeline(pipeline_t *p);
int            is_builtin(const char *argv0);
int            run_builtin(const pipeline_t *p, int shmid);
```

Two of those contracts shape your `main()`:

- `parse_line` prints nothing. It returns a `parse_result_t`; for an error, the caller prints `parse_errmsg(rc)`, and for `PARSE_EMPTY` it does nothing. A silent parser can be tested on its own.
- `run_builtin` never calls `exit()`. For the `exit` builtin it returns `RAMSH_EXIT_REQUESTED`, and `main()` runs the same cleanup (detach, destroy, return) that it runs at end-of-file and at the end of a `-c` run. One cleanup path is easier to get right than three.

6 Part-2: N-Stage Pipelines

This is the center of gravity of the assignment. Your executor must handle any pipeline from 1 to 8 stages with a single general loop; the autograder tests at 1, 2, 3, and 8 stages, and it pushes more data through the pipes than a pipe can buffer, which is precisely the condition under which the common bugs stop hiding.

This is also where the project becomes genuinely interesting. The stages of a pipeline are ordinary independent processes that happen to inherit a carefully engineered descriptor topology. No stage knows the whole pipeline exists. `grep` does not know who produced descriptor 0, and `wc` does not know who will eventually close the last write end. Your shell creates the appearance of cooperation by arranging the world correctly before `exec()`, and then largely gets out of the way. That is a remarkable amount of leverage from a handful of system calls.

The old "plumbing" metaphor for Unix pipelines is almost suspiciously accurate. Unfortunately, this includes the part where leaving one end open somewhere you cannot see it causes trouble somewhere else.

6.1 Required interface

The executor is the Part 2 section of `ramsh.h`: `run_pipeline`, plus two small helpers that the autograder unit-tests separately. `decode_status` is your `WIFEXITED`/`WIFSIGNALED` decoding, and `leak_target` parses the `RAMSH_LEAK` variable of Part 4. The comment above `run_pipeline` in Appendix A restates the requirements below as a five-step contract.

```
int32_t decode_status(int wstatus);
int     leak_target(int npipes);
int32_t run_pipeline(const pipeline_t *p, int32_t *pids_out,
                     int32_t *status_out, uint32_t *elapsed_ms_out);
```

6.2 Requirements of Task

1. An N-stage pipeline uses exactly $N - 1$ pipes. Pipe k connects the `stdout` of stage k to the `stdin` of stage $k + 1$.
2. **Fork every stage before waiting for any stage.** This is not a style preference. If you complete a full `fork()`/`waitpid()` cycle for stage 0 before forking stage 1, then `seq 1000000 | wc -l` deadlocks: `seq` fills the pipe's kernel buffer (64 KiB on Linux), blocks in `write()` waiting for a reader that you

have not forked yet, and your shell blocks in `waitpid()` waiting for `seq` to finish writing. Each side waits on the other forever. The pipeline stages of a shell are inherently concurrent processes, even though your shell contains no threads.

3. Each child, after `fork()` and before `execvp()`, uses `dup2()` to install the correct pipe ends onto descriptors 0 and 1, and then closes *every* pipe descriptor it holds, including the ones it just duplicated from. `dup2()` copies the descriptor; the original is a separate reference that must still be closed.

4. The parent closes all $2(N-1)$ pipe descriptors after the last fork and before the first `waitpid()`. The parent is not a stage; every pipe descriptor it retains is a leak. A leaked write end hangs the pipeline, and Part 4 makes you demonstrate this on purpose. A leaked read end is quieter: `seq 1000000 | wc -l` still works, but in `yes | head -n 1` the `yes` process never receives the `SIGPIPE` that should stop it, because a reader (your shell) still exists. Between pipelines, the shell's own descriptor table must be back to exactly 0, 1, and 2; the autograder checks this directly.

5. Reap with `waitpid()` on each stage's pid, **in stage order**, printing one reap line per stage in the trace format from Part 1. This discipline is what makes your trace deterministic and therefore gradable by exact match; a bare `wait()` loop reaps in completion order, which varies run to run.

6. Trace lines: one `ramsh: stage <k> [pid <P>] <argv0>` line per stage, printed by the parent in stage order at fork time, then one reap line per stage in stage order.

7. After all stages are reaped, record the pipeline in the ledger (Part 3) and only then print the next prompt.

8. Descriptor hygiene is graded directly: the autograder runs `ls /proc/self/fd` as the only, first, middle, and last stage of a pipeline, and the only descriptors visible in the child must be 0, 1, 2, and 3 (descriptor 3 being the one `ls` itself opened to read the directory; if you can explain why that fourth entry is not a bug in your shell, you have understood this part).

6.3 Worked example (use this to self-check before submitting)

The session below feeds five lines to an interactive `ramsh`. Prompts are omitted for clarity; the autograder strips the literal token `ramsh>` from `stderr` before comparing.

```
echo hello world
seq 3 | sort -r | head -n 1
nosuchcmd | wc -l
ledger
exit
```

`stdout` must be exactly:

```
hello world
3
0
LEDGER total=3 capacity=64 shmid=163842
#1 stages=1 ms=1 status=0 cmd=echo hello world
#2 stages=3 ms=4 status=0,0,0 cmd=seq 3 | sort -r | head -n 1
#3 stages=2 ms=3 status=127,0 cmd=nosuchcmd | wc -l
```

`stderr` must be exactly (after normalization; see the guarantee below):

```
ramsh: stage 0 [pid 41213] echo
ramsh: stage 0 [pid 41213] exited 0
ramsh: stage 0 [pid 41214] seq
ramsh: stage 1 [pid 41215] sort
ramsh: stage 2 [pid 41216] head
ramsh: stage 0 [pid 41214] exited 0
ramsh: stage 1 [pid 41215] exited 0
```

```
ramsh: stage 2 [pid 41216] exited 0
ramsh: stage 0 [pid 41217] nosuchcmd
ramsh: stage 1 [pid 41218] wc
ramsh: exec failed for nosuchcmd: No such file or directory
ramsh: stage 0 [pid 41217] exited 127
ramsh: stage 1 [pid 41218] exited 0
ramsh: stage 0 [pid 41219] ramview
ramsh: stage 0 [pid 41219] exited 0
```

Note what the third command demonstrates: `nosuchcmd` fails to exec and dies with 127, yet `wc` still runs, prints `0`, and exits cleanly, because the dying child's descriptors were closed by the kernel at `_exit()`, delivering the end-of-file that `wc` was waiting for. A pipeline is not a transaction; each stage succeeds or fails on its own, and the ledger records all of it.

If your reap lines for the three-stage pipeline appear in a different order on different runs, you are calling `wait()` in a loop instead of `waitpid()` on each stage's pid in stage order. If `seq 3 | sort -r | head -n 1` prints `3` but your shell never prints the next prompt, a write end of one of your pipes is still open in the parent, and `sort` or `head` is blocked in `read()` waiting for an end-of-file that can never arrive. That second failure is not a nuisance to be worked around; it is the subject of Part 4.

A guarantee about nondeterminism. Process IDs, elapsed milliseconds, and shared-memory segment IDs are inherently different on every run. The autograder normalizes all three (the tokens `[pid <P>]`, `ms=<M>`, and `shmid=<S>` are matched by pattern, never by value) before performing exact comparison. The `exec failed` line is printed by the *child*, concurrently with the parent's trace, so it can land anywhere, even ahead of its own stage's fork line; the autograder removes it before the exact comparison and checks it separately, for presence and text. The interleaving of the stages' own `stdout` is never inspected at all; only the final pipeline output is. Everything that *is* compared exactly, namely the fork lines, the reap lines and their order, and the ledger fields, is fully deterministic under the in-order `waitpid()` discipline required above. If your trace differs from run to run, treat that as a bug in your wait discipline, not as unfairness in the autograder.

6.4 Extra credit: redirection

Supporting `< infile` on the first stage and `> outfile` on the last stage, with correct interaction with pipes and correct error messages for unopenable files, is accepted as extra credit worth up to 0.75 points. It must be documented in your design document and covered by tests you wrote; an undocumented, untested implementation earns nothing, and any regression it introduces in the base contract costs you the base points. The base autograder never sends < or >.

7 Part-3: Shared-Memory Session Ledger

Everything your shell learns while reaping a pipeline (pids, statuses, timing) currently evaporates when the next prompt is printed. In this part you give it a memory, and, more importantly, you put that memory somewhere another process can find it.

This gives you a second, deliberately different view of interprocess communication. A pipe moves a byte stream through a kernel-managed channel; shared memory lets two processes inspect the same region of storage and leaves the interpretation of those bytes to them. Both mechanisms let processes communicate, but they make very different promises about ownership, lifetime, and coordination. Seeing both inside the same program is considerably more useful than memorizing two APIs in isolation.

At startup, `ramsh` creates one System V shared memory segment sized to hold a `ledger_t` (`shmget` with `IPC_PRIVATE`, permissions `0600`), attaches it, and zeroes it. After reaping each non-builtin pipeline, the shell writes one record. The `ledger` builtin then forks a child that execs `ramview <shmid>`, passing the segment ID *as a decimal string in argv*. The viewer attaches the same segment read-only (`shmat` with `SHM_RDONLY`), prints every surviving record, detaches, and exits. Section 8 explains why the ID crosses the exec boundary as a string when the shell's pointer to the very same memory could not.

An aside: "System V" is the name of an API, not an operating system

UNIX System V was AT&T's commercial Unix of 1983, and it shipped three interprocess-communication mechanisms that outlived it: message queues, semaphores, and the shared-memory calls this assignment uses (`shmget`, `shmat`, `shmdt`, `shmctl`). They were later standardized, and Linux has implemented them natively for decades, so there is nothing to emulate and nothing to install. The name simply stuck, the same way "BSD sockets" did.

Two things are worth knowing so the name does not mislead you. First, Linux also offers a newer and entirely separate *POSIX* shared-memory API (`shm_open` with `mmap`). A web search for "Linux shared memory" will turn it up first, and it is not what this assignment asks for: its objects never appear in `ipcs -m`, and the autograder does not look for them. Second, "SysV init", the boot system that `systemd` replaced, shares only the name. On the lab machines, `man 7 sysvipc` is the overview and `man 2 shmget` is where to start.

The API looks its age because it is. That is not entirely a disadvantage for this assignment: every stage of the object's lifecycle is explicit. You create it, attach it, detach it, and remove it. Nothing quietly tidies the segment away merely because your process has lost interest in it. The 1980s have, in this particular respect, delegated the cleanup to you personally.

7.1 Required interface

The ledger is the Part 3 section of `ramsh.h`:

```
typedef struct {
    uint64_t seq;                       /* 1-based; never reused        */
    int32_t  nstages;
    int32_t  pids[RAMSH_MAX_STAGES];    /* valid for k < nstages        */
    int32_t  status[RAMSH_MAX_STAGES]; /* decode_status() values       */
    uint32_t elapsed_ms;
    char     cmdline[LEDGER_CMD_MAX];  /* NUL-terminated copy of raw   */
} ledger_entry_t;

typedef struct {
    uint64_t       total;               /* pipelines ever recorded      */
    ledger_entry_t ring[LEDGER_CAP];    /* seq s lives in               */
                                        /*   ring[(s - 1) % LEDGER_CAP] */
} ledger_t;

int       ledger_create(void);
ledger_t *ledger_attach(int shmid, int readonly);
void      ledger_record(ledger_t *led, const pipeline_t *p,
                        const int32_t *pids, const int32_t *status,
                        uint32_t elapsed_ms);
int       ledger_print(const ledger_t *led, int shmid, FILE *out);
void      ledger_detach(ledger_t *led);
void      ledger_destroy(int shmid);
```

`ledger_print` writes the viewer's output to any `FILE *`. That keeps `ramview`'s `main()` down to attach, print, detach, and it lets the autograder test your ring arithmetic on an ordinary `ledger_t` in plain memory, with no shared memory involved.

7.2 Why the struct looks the way it does

Notice what `ledger_entry_t` does *not* contain: pointers. A `char *cmdline` would compile, run, and appear to work in the shell, and then dereference garbage in `ramview`, because a pointer is an address in one process's virtual address space and means nothing in another's. Worse, `shmat()` is free to attach the same segment at *different addresses in different processes*, so even a pointer aimed into the segment itself is not portable across the attachment. Everything in a shared struct must therefore be inline: fixed-size arrays, not pointers, and fixed-width types (`int32_t`, not `pid_t` or `int`) so the layout is not an accident of one compiler's mood. The `static_assert` lines in `ramsh.h` pin that layout down: an entry is exactly 336 bytes with `cmdline` at offset 80, so a layout change fails the build instead of silently corrupting what `ramview` reads. Command lines longer than 255 characters are silently truncated in the ledger; the design document must state this limit.

7.3 Ring semantics

The ledger is a ring of 64 entries. `total` counts every pipeline ever recorded and never decreases; when the ring is full, the oldest entry is overwritten. `ramview` prints the header line, then every surviving entry in ascending `seq` order, oldest first. After 70 recorded pipelines, the viewer must print `total=70` and its first entry line must be `#7`; entries 1 through 6 are gone, and that is the correct behavior, not data loss to apologize for.

7.4 Why this ledger needs no lock

You may have noticed that a shared, mutable data structure read by two processes usually comes with a semaphore attached, and this one does not. That is not an oversight, and "it seems to work" is not the

reason. It needs no lock because of two structural facts you can point to in your own code: **(1)** exactly one process, the shell, ever writes to the segment, and **(2)** the only reader, `ramview`, runs only while the shell is blocked in `waitpid()` waiting for that very reader to finish. Writer and reader are never inside the segment at the same time, not because the hardware promises it, but because your program's structure makes the overlap impossible. This is called a *single-writer discipline*, and being able to state it precisely, and to say exactly which of the two facts a background-job feature or a second concurrent shell would destroy, is worth more on the quiz than any amount of code. A later assignment in this course deliberately breaks assumption (2), and semaphores enter your life at that point and not before.

7.5 Cleanup

System V segments are kernel objects, not process memory: they outlive the process that created them. `exit`, end-of-file on `stdin`, and the end of a `-c` run must all detach (`shmdt`) and destroy (`shmctl` with `IPC_RMID`) the segment; the autograder checks for leftover segments after your shell exits, and a leftover segment fails the test. A shell killed by a signal never reaches its cleanup code, so every run that `timeout` kills in Part 4 strands one segment. That is expected, and the autograder removes those itself. During development, when your crashing shell strands segments (it will), `ipcs -m` lists them and `ipcrm -m <shmid>` removes them. Getting on first-name terms with both commands is part of the assignment.

8 Three Invariants That Explain Every Bug in This Assignment

Every plumbing bug you will meet in Parts 1 through 4 is a violation of one of three facts. It is worth being able to recite all three, and to say which one a given bug violates, since the quiz asks about the reasoning behind your code, not just whether the calls are typed correctly.

If you find yourself staring at a hang, resist the urge to add code until you can name the violated invariant. In this assignment, symptoms multiply faster than causes. A reader blocked forever, a writer that never receives `SIGPIPE`, and a viewer reading nonsense can look like unrelated failures until you reduce them to the kernel rules underneath. That reduction, from a messy symptom to a small invariant, is one of the most useful habits in systems debugging.

Invariant 1: `fork()` copies the descriptor table; `exec()` preserves it. A child starts life with a duplicate of every descriptor its parent held, and `execvp()` replaces the program without touching that table. The duplicates refer to the same open files as the originals, file offsets included, which is why a child can move the parent's place in a file (Section 5, the exec-failure protocol). This gap, after `fork()` and before `exec()`, is the only place shell plumbing can happen, and it is the entire mechanism by which `ramsh` controls programs like `sort` that have never heard of it. `sort` does not cooperate with your pipeline; it reads descriptor 0 because that is all it knows how to do, and descriptor 0 is a pipe because you made it one before `sort` existed.

Invariant 2: a pipe delivers end-of-file only when *every* write end is closed. The kernel maintains a reference count of open write descriptors per pipe, and a `read()` returns 0 only when that count reaches zero. The count tracks descriptors, not intentions: the parent shell, which never had any intention of writing, holds a write end just as firmly as the producer stage does. This is why one forgotten `close()` in the parent hangs a pipeline in which every child behaved perfectly, and it is the entire subject of Part 4.

Invariant 3: descriptors survive `exec()`; memory attachments do not. `exec()` keeps the descriptor table but demolishes and rebuilds the address space, which unmaps every `shmat()` attachment. This asymmetry dictates the design of Part 3: the shell cannot pass `ramview` a pointer to the ledger (per-process, destroyed by exec, and not even guaranteed to be the same address after re-attachment), but it can pass the segment *ID*, a system-global name, as an ordinary string argument, and let the viewer perform its own `shmat()`. A descriptor number could have crossed the same boundary alive; an address cannot.

9 Part-4: File-Descriptor Forensics

Part 2 required you to close descriptors correctly. This part requires you to demonstrate, on purpose and with evidence, what happens when you don't. Reading about the hung-reader bug is not the same as watching your own shell freeze because of a single line you deleted with your own hands; the second one is the version you will still remember at the final exam, and long after.

Think of this part as a controlled experiment rather than an exercise in breaking things. You begin with a causal claim, change exactly one condition, and then ask the kernel for evidence. The interesting achievement is not making the program fail; programs tend to volunteer for that service. The achievement is predicting the precise shape of the failure before you run it, including who blocks, where it blocks, and why the kernel has no reason to let it proceed.

9.1 The sabotage switch

Your shell must support the environment variable `RAMSH_LEAK`. When `RAMSH_LEAK` holds a decimal integer n with $0 \le n < N-1$, `run_pipeline` must skip exactly one `close()`: the *parent's* close of the *write end of pipe n*. Children behave normally; every other close happens normally. Any other value (unset, not a number, or out of range for the pipeline being run) is ignored. `leak_target` in `ramsh.h` is where that rule lives. This switch is autograded directly:

```
$ ./ramsh -c "echo done | wc -l"                  # control run
1                                                 # ...and exits 0
$ RAMSH_LEAK=0 timeout 5 ./ramsh -c "echo done | wc -l"
$ echo $?
124                                               # hung; timeout killed it
```

Be sure you can narrate the second run: `echo` writes and exits; `wc` consumes the data and then blocks in `read()`, because the parent still holds a write end of pipe 0 and the kernel's reference count cannot reach zero (Invariant 2); the shell meanwhile blocks in `waitpid()` for `wc`. Nobody crashed. Nobody erred. Everybody waits, forever, and `timeout` is the only party with an exit strategy.

9.2 The forensics report

Submit a file named `forensics.md` in the repository root containing three short experiments. Evidence in this report must be pasted from real runs, and Section 15 explains how it is checked; a beautifully written report describing runs that never happened scores as fabrication. The autograder awards the Part 4 points only when `forensics.md` is present; its content is evaluated with your transcripts and at the walkthrough.

1. **The descriptor map.** For `seq 3 | sort -r | head -n 1`, draw the complete descriptor table of all four processes (parent and three children) at the instant just after the last `fork()` and before any `exec()` or cleanup `close()`: every descriptor, what it refers to, and in which process. ASCII art is fine. Then list every `close()` that must occur, who performs it, and which line of your source performs it.

2. **Predict, then break.** Before running it, write down precisely what `RAMSH_LEAK=0 ./ramsh -c "echo done | wc -l"` will do and why, naming which process blocks in which system call. Then run it under `timeout 5`, and paste the exit code plus one piece of direct evidence of the blocking syscall: the `wchan` column of `ps -o pid,comm,wchan` on the stuck processes, or the first line of `cat /proc/<pid>/syscall`, or an `strace -p` excerpt. State whether your prediction survived contact with the kernel.

3. **A leak of your own.** Choose a *different* `close()` in your executor (a child-side one is more interesting), delete it temporarily by hand, predict the outcome, run an experiment that exposes it (hint: some child-side leaks are invisible on small inputs and fatal past 64 KiB; explain why), paste the evidence, and restore the line. Document the diff you tested.

10 Personalized Verification

Passing the autograder does not end your obligation on this assignment. After the submission deadline, you will be given a small modification specific to you, previously unseen, to make to your own already-submitted code, for example: a `!!` builtin that re-runs the most recent ledger entry, a `ledger -n <K>` flag that prints only the last K records, making `LEDGER_CAP` configurable through an environment variable, or `>` redirection on the final stage if you did not attempt the extra credit. You will make this change live, during the walkthrough described in Section 11, starting from your own submitted source and nothing else.

This is also closer to real systems work than a frozen specification. Codebases survive by absorbing changes without quietly invalidating the assumptions on which they were built. A small new requirement is therefore a useful test of understanding: you must decide where the change belongs, which existing mechanism it can reuse, and which invariant it might threaten. Passing yesterday's tests tells us the program worked yesterday. Extending it coherently tells us you understand why.

This is not a second, separately weighted assignment. It is the concrete form of the process-engineering component described in Section 11 and reflected in the grading breakdown: a static submission that compiles and passes every published test case is necessary but not sufficient, since none of that requires you to still be able to navigate, defend, or extend your own code once the requested change is one neither you nor any AI assistant could have seen in advance.

Why this exists. Every behavior in this handout is specified precisely enough to autograde by exact match or by behavioral check, which is exactly what makes rigorous grading possible. It is also, unavoidably, what makes this document alone reproducible from a single request to an AI coding tool. Personalized Verification exists because a static document, however complete, can only ever get you to a plausible-looking submission. Only your ability to work with your own code after the fact, on a change you have never seen before, can confirm the work behind it was actually yours.

11 Required Use of Claude Code

You are required to use Claude Code for this assignment and to log that usage, but the goal is for you to leave the assignment understanding every line you submit, and not just produce working code by any means available.

For this project, Claude Code is most useful as an unusually patient systems colleague: good for interrogating a descriptor map, challenging an ownership argument, generating a test for a hypothesis, or helping isolate why a process is stuck in `read()`. It is considerably less useful as a ghostwriter whose work you plan to meet for the first time at the walkthrough.

A few ground rules:

- Appropriate use: asking Claude Code to explain a concept, review your reasoning about descriptor lifetimes, help you debug a hang or a zombie process, discuss tradeoffs between implementation choices, or push back on a design you propose.
- Appropriate use: asking Claude Code to help you understand `dup2()` semantics, the `waitpid()` status macros, System V shared memory lifecycles, or why `_exit()` and `exit()` differ in a forked child, before you implement those parts yourself.
- Not appropriate: pasting this handout, or a section of it, into Claude Code and asking for a complete implementation to submit. Graders will compare submissions for structural fingerprints of this pattern, and a wholesale-generated shell tends to fail in recognizable ways anyway: the descriptor map in `forensics.md` will not match the code beside it, and the quiz will find the gap in minutes.
- You must submit the raw session transcript(s) Claude Code already saves for you, not a hand-written or `/export`'d summary. By default these live locally as JSONL files at `~/.claude/projects/<project>/<session-id>.jsonl`, one file per session, timestamped and recorded tool call by tool call. Copy the `.jsonl` file(s) for this assignment's project directory into your submission as-is; do not edit, reformat, retype, or hand-transcribe them. Work on this assignment in its own dedicated project

directory, both so this copy step is a single `cp`, and so the file doesn't contain unrelated conversations from other work.

- Every student, not a flagged subset, sits for the short mandatory live walkthrough described above after the deadline. It is not a bonus check triggered by a mismatch between your log and your code; it happens for everyone.
- Claude Code purges local session transcripts automatically after 30 days by default. If you start this assignment more than a few weeks before the deadline, your earliest sessions may already be gone by the time you submit unless you copy them out periodically or raise `cleanupPeriodDays` in your settings. Losing early transcripts to this default is not a valid excuse for an incomplete submission, plan around it.
- Build-and-run instructions are not a separate README. Fold them into the design document (see Section 12), so there is one authoritative written document rather than two that can drift apart.

A well-used 30 hours with Claude Code on this assignment looks like a lot of short, specific exchanges about one function or one bug at a time, not one long exchange that ends with a finished project pasted into your editor.

What "solve this" looks like in a transcript, and why it doesn't work. A session whose first substantive prompt is this handout, or a close paraphrase of it, followed by little or no iteration, is a visible and specific pattern in a raw transcript, not a vague impression. TAs are instructed to treat a single large opening prompt with minimal back-and-forth as a trigger for closer attention at the walkthrough, regardless of whether the resulting code passes the autograder. Passing the autograder was never the hard part of this assignment; it is worth restating that plainly here.

12 Deliverables

You should use Canvas to submit a single archive (`tar` or `zip`) of your entire project repository, including the `.git` directory, so its full commit history is present for the prompt-log grading described in Section 15. The archive must contain:

- The complete source tree: every `.c` and `.h` file you wrote, plus the unmodified `ramsh.h`. The `Makefile` by itself is not your source; the graders build both binaries from these files.
- The `Makefile` from Appendix B, extended with any files you add. It must build `ramsh`, `ramview`, and `libramsh.a` with `-std=c23 -Wall -Wextra -Werror` and no warnings, and it must support both `make all` and `make clean`. No function declared in `ramsh.h` may be defined in the file that holds a `main()`.
- A design document (1–2 pages) covering: your `ledger_entry_t` layout and why shared structs may not contain pointers, your descriptor close-plan for the general N-stage case (a table of who closes what is ideal), why the ledger needs no lock in your own words and precisely which assumption a concurrent feature would break, why `cd` must not fork and `ledger` must, the truncation limits you enforce, and the known limits of your parser. This document also carries the build-and-run instructions that used to live in a separate README, matching the I/O contract in Section 13 exactly; there is no separate README.
- The forensics report `forensics.md` described in Section 9, with pasted evidence from real runs.
- The raw Claude Code session transcript(s) (`.jsonl` files) for this project's directory, copied in as-is. Not a summary, not a `/export`, not retyped. Place these inside the repository before you archive it, so they travel with the `.git` history rather than as a loose attachment.

13 Command-Line and Output Contract

The autograder invokes your compiled binaries directly and parses both output streams, so the interface below must match exactly. Do not print anything to `stdout` other than what the executed programs and `ramview` produce; every word the shell says about itself goes to `stderr`. The message formats are also defined as macros in `ramsh.h`; use them rather than retyping the strings.

```
./ramsh                 interactive; reads pipelines from stdin
                        until EOF or the exit builtin
./ramsh -c "<line>"     one-shot; runs one pipeline, cleans up,
                        exits with the last stage's status
                        (2 for a syntax error; a builtin's own
                        status for cd, exit, and ledger)
./ramview <shmid>       attaches the segment read-only, prints
                        the ledger, detaches, exits 0
                        (exit 1 + message on stderr if the
                        argument is not a valid ID or the
                        segment cannot be attached)

Environment:
RAMSH_LEAK=<n>          sabotage mode (Part 4): the parent skips
                        closing the write end of pipe <n>

Shell trace lines (stderr):
ramsh: stage <k> [pid <P>] <argv0>
ramsh: stage <k> [pid <P>] exited <status>
ramsh: stage <k> [pid <P>] killed by signal <N>
ramsh: exec failed for <argv0>: <strerror>
ramsh: syntax error near '|'
ramsh: too many stages (max 8)
ramsh: builtins cannot appear in a pipeline
ramsh: cd: <strerror>

Viewer output (stdout):
LEDGER total=<T> capacity=64 shmid=<id>
#<seq> stages=<n> ms=<ms> status=<s0,s1,...> cmd=<line>
```

The autograder runs `ramsh` from the build directory with the build directory prepended to `PATH`, so the `ledger` builtin may simply `execvp("ramview", ...)`. Do the same when testing by hand: `PATH=$PWD:$PATH ./ramsh`. Resist the temptation to exec `./ramview` by relative path; the `cd` builtin exists, and it will be used before `ledger` in at least one test, which tells you exactly what that test is checking. The autograder also sets `LC_ALL=C`, so `<strerror>` is always the C-locale text, such as `No such file or directory`.

14 Grading

This assignment is worth **15 points**. **9 points (60%)** are for program correctness and are graded by the autograder. **6 points (40%)** are for process engineering: 2 points for an analysis of your Claude Code prompts, and 4 points for the quiz on your own code described in Section 11 and Section 10.

Before any test runs, the autograder builds your submission with the flags in Appendix B. A submission that does not build earns 0 of the 9 correctness points; the 6 process-engineering points are graded either way. Within the 9 correctness points, grading has two tiers, and the difference between them is not a grading technicality. It reflects a real property of concurrent processes that is worth understanding in its own right.

- **Tier 1: exact match after normalization.** Your shell's own trace, its error messages, the final output of each pipeline, and the ledger fields are all fully determined by your code once pids, timings, and segment

IDs are normalized away, *because this handout requires the discipline that makes them deterministic*: in-order `waitpid()`, stderr/stdout separation, and fixed message formats. These are compared exactly, in the same spirit as the worked example in Section 6.

- **Tier 2: behavioral.** Some properties of a correct shell cannot be read off a transcript at all: that an 8-stage pipeline moving a hundred thousand lines terminates, that a sabotaged run hangs rather than terminating, that a child's descriptor table contains nothing it shouldn't, that no shared-memory segment outlives the shell, and that valgrind finds no leaks. These are graded by running your program under a time limit, inspecting `/proc`, the System V segment table, and valgrind, and observing what it *does*. Two correct shells may differ internally in a hundred ways; they cannot differ here.

This split is also why the in-order `waitpid()` requirement exists at all: it converts the one genuinely nondeterministic thing about a pipeline, the order in which its stages happen to finish, into a deterministic reporting order that a machine can grade and a human can read. Appendix C lists every test, its points, and what it checks.

Component	Method	Points
Part 1: single commands and builtins	Unit tests of the parser; exact-match trace and output; exec-failure protocol, including the `_exit()` test; `cd`/`exit` semantics; `-c` exit statuses (P1.1–P1.8)	2
Part 2: pipelines	Exact-match traces at 1–8 stages, plus behavioral stress: large transfers under a time limit, `SIGPIPE`, reap order, descriptor hygiene via `/proc/self/fd`, syntax-error handling (P2.1–P2.8)	3
Part 3: session ledger	Viewer contract, ring wraparound at 70 entries, unit tests of `ledger_record`/`ledger_print`, read-only attach, `cd` before `ledger`, segment cleanup (P3.1–P3.7)	2
Part 4: forensics	`RAMSH_LEAK` hang-versus-control behavior and `leak_target`; requires `forensics.md` to be present (P4.1–P4.3)	1
Code quality	No memory errors or leaks under valgrind; no zombies; the shell's own descriptors back to 0, 1, 2 between pipelines (Q1–Q3)	1
Program correctness subtotal (60%)	*Autograded (Appendix C)*	*9*
Prompt analysis	Review of your raw Claude Code transcripts (Section 15) for genuine, iterative use rather than wholesale generation, corroborated against your repository	2
Quiz	Short quiz generated from your own submitted code, administered at the live walkthrough, including the personalized modification (Sections 10 and 11)	4
Process engineering subtotal (40%)		*6*
Total		**15**

The design document and `forensics.md` remain required deliverables. They carry no separate point

weight, but both are reviewed by the TA alongside your transcripts and are used to seed quiz questions; a missing or perfunctory descriptor map will cost you where it hurts, on the quiz, in front of a TA holding your own diagram. The same goes for structure and readability: no test can measure them, but you will navigate and extend your own code at the walkthrough, and code that is hard to read is hard to extend with a TA watching.

15 How We Grade Your Claude Code Prompt Logs

Two of the 15 points on this assignment come from your Claude Code transcripts. It is worth being precise about what those points measure, because they are easy to misread as a participation check that any submitted log passes. They are not. They measure whether your transcripts show the fingerprints of someone building and debugging their own program, and that is a claim we can check against your repository rather than take on faith.

The governing principle is simple, and everything below follows from it: **the grade is based on what can be corroborated against the repository, never on the log's own narrative**. A transcript can say anything. It can claim you caught a subtle bug, demanded a test, or rejected a bad design. What earns credit is not the claim but the evidence for it sitting in your git history, your source, and your tool output. A confident story with nothing behind it scores below a modest one that checks out.

15.1 Mechanism

The grader runs headless (`claude -p`) or as a slash command, with your repository checked out alongside the log, not the log alone. It reads your transcripts as a set of claims and then verifies those claims by inspection against the repo. Concretely, it asks questions of the form: does the commit a prompt refers to actually exist; does the fix for a diff you rejected in conversation show up later in history; does the test you told Claude Code you wanted actually appear in your test files; does the hang you debugged at, say, the parent's close loop in `exec.c:141` correspond to code that in fact changed around that line, and does the evidence pasted into `forensics.md` match tool output that a session actually produced? Episodes that cross-check against a commit, a test, or a diff count. Episodes that float free of any repository artifact do not, no matter how well written they are.

This is the same philosophy as the sabotage switch in Section 9 and the live walkthrough in Section 10, applied to your process instead of your output. A plausible-looking artifact that does not actually connect to real work is easy to produce and, deliberately, easy to catch.

15.2 The scale

Your transcripts are scored on the five-level scale below for technical substantiveness. The level determines the point value directly, in even steps of 0.4: level 5 earns both points, and each level down subtracts four tenths. A submission with no transcripts scores 0.

Level	Name	What it looks like	Points
5	**Verified engineering dialogue**	Prompts reference concrete artifacts (file and line, invariants, tool output); your pushback demonstrably changed outcomes; every required episode cross-checks against a commit, test, or diff.	2.0
4	**Substantive, partially corroborated**	The same quality of reasoning, but one or two episodes cannot be tied back to repository evidence.	1.6
3	**Concrete but shallow**	Real artifacts are cited, but the judgments are acceptance without technical reasoning, of the "looked good, kept it" variety.	1.2
2	**Generic**	Plausible transcripts with no verifiable anchors; the log could describe any student's project as easily as yours.	0.8
1	**Performative or contradicted**	The narrative conflicts with git history, or episodes appear fabricated.	0.4

15.3 What this means for how you work

You do not earn a high score by narrating impressively. You earn it by working in a way that leaves a verifiable trail: small commits, tests you actually asked for and actually wrote, real hangs pasted from real runs, and design decisions you can point to in the code. If you use Claude Code the way Section 11 describes, as a collaborator you argue with one function and one bug at a time, the corroboration falls out of your normal workflow and you will land at level 4 or 5 without effort. The only way to score low here is to submit a log that has little to do with the code beside it, and that is precisely the case these points exist to catch.

16 Sample Test Cases (Autograder Preview)

A few of the actual autograder cases, so you know what conformance looks like before you submit. The test ID in brackets matches the catalog in Appendix C.

- `./ramsh -c "echo hello"` [P1.3]: `stdout` is exactly `hello`, exit status 0, and the `stderr` trace matches the two-line format from Part 1 after pid normalization.
- `./ramsh < script.txt`, where `script.txt` holds `nosuchcmd`, `echo A`, and `echo B` on three lines [P1.5]: `stdout` is exactly `A` and `B`, once each. A child that calls `exit()` instead of `_exit()` prints them twice.
- `./ramsh -c "seq 1000000 | wc -l"` with a 10-second limit [P2.2]: prints `1000000` and exits 0. This one case catches both classic failures at once: the sequential fork-then-wait deadlock (requirement 2 of Part 2) and a leaked parent-side write end, either of which turns it into a timeout.
- `./ramsh -c "seq 100000 | cat | cat | cat | cat | cat | cat | wc -l"` [P2.3]: eight stages, prints `100000`. Nobody hand-wires seven pipes; this test exists to confirm your executor is a loop.
- `./ramsh -c "ls /proc/self/fd"` [P2.4]: `stdout` is exactly the four lines `0`, `1`, `2`, `3`. Descriptor 3 belongs to `ls` itself; anything above it is a pipe descriptor your shell failed to close before `exec`.
- `./ramsh -c "yes | head -n 1"` [P2.5]: prints `y`, reports stage 0 `killed by signal 13`, and exits 0. A read end left open anywhere turns it into a timeout.
- `./ramsh -c "nosuchprog | wc -l"` [P1.6, P2.8]: `stdout` is `0`, the trace shows stage 0 exiting 127, and the shell exits 0. Then `./ramsh -c "nosuchprog"` alone must exit 127 [P1.4].

- `RAMSH_LEAK=0 timeout 5 ./ramsh -c "echo done | wc -l"` [P4.1]: exits 124. The identical command without the environment variable prints `1` and exits 0.
- A scripted session of three pipelines followed by `ledger` and `exit` [P3.1, P3.7]: the viewer output must match the contract in Section 13 after normalization, and no segment owned by the test user may remain once the shell exits.
- A scripted session of 70 pipelines followed by `ledger` [P3.2]: the header reports `total=70` and the first entry printed is `#7`.

The public tests, marked in Appendix C, are posted with the starter files, together with the reference traces and a reference ledger transcript.

17 Late Submission

The late submission policy for CS370 is available from the course website at `https://www.cs.colostate.edu/~cs370`.

18 The spirit of this assignment

You are required to submit your own work, and you are required to use Claude Code to help you produce it. Those two requirements are not in conflict with each other. Your own work in the context of this assignment has never meant work built with no help; it means work that you understand well enough to defend, regardless of how much help you had along the way.

There is no reason to submit someone else's work on this assignment. Not as a shortcut. Not under deadline pressure. Not under different variations of "*you can't believe what I am going through right now*". Every tool that could make a borrowed solution look plausible is the same tool you were already handed on day one.

Whatever a classmate's code might save you in Part 2, it costs you in Part 4 and at the walkthrough. A descriptor map drawn from code you did not write, and a quiz built from it, is not a shortcut. It is a trap you built for yourself.

Do the assignment the way it is designed to be done. Use Claude Code as a real collaborator, not a copy-paste machine. There is nothing left to gain by doing it any other way.

If you do this assignment well, the artifact you keep is not merely `ramsh`. You will have built a compact mental model of process creation, inheritance, IPC, blocking, cleanup, and failure that transfers to much larger systems. A shell is a small program, but it gives the operating system remarkably few places to hide. By the end, you should be able to explain not only why your shell works when it works, but why it hangs when it hangs. The second explanation is usually the more valuable one.

19 How AI Was Used in Developing This Assignment

This assignment has been developed with assistance from AI tools. The assignment itself, including its structure, requirements, learning objectives, and overall design, is my work. I used AI as a second set of eyes. A *tireless* second set of eyes. It helped me interrogate the specification and verify technical details. Notably, it helped me identify places where what I intended and what I had actually written were not quite the same thing.

In particular, I iterated with Claude and Claude Code to check the correctness of the specification and prompts, identify inconsistencies, explore corner cases that were missing from earlier versions, and test whether the instructions would lead to the behavior that I actually intended. That last part mattered more than you might think. This assignment has several moving parts: a small ambiguity in one place can have large, downstream consequences several steps later. Claude Code was useful for finding those places before you did.

These tools also identified places where my exposition could be clearer. I used those observations selectively in revising the text. The examples, explanations, constraints, and progression of the assignment went through substantial human iteration as well. Quite a lot of it. Intermediate drafts of this assignment were vetted by the GTAs, who would also point places where things were underspecified.

Another thing that I tested was what happens when a long assignment specification gets summarized by an LLM. I deliberately tested what happens when a long, detailed specification is handed to a model with a prompt along the lines of, "Give me a bulleted list of what he's going on about." This turned out to be useful. It exposed places where an important constraint, qualification, or dependency could disappear in the summarization process, or where a concise summary could be technically correct but still lead you down the wrong path. I revised the specification to reduce those unintended consequences without trying to make the full document reducible to a small set of convenient bullets. Please read the entire assignment.

There is a certain appropriateness to all of this. I am asking you to use AI in this assignment as an engineering tool, not as a substitute for understanding. That is largely how I used it in creating the assignment too: to question, check, probe, and refine work that I was already doing. The goal was to make the assignment better before asking you to wrestle with it.

20 Version History

This section will reflect the change history for the assignment. It will list the version number, the date it was released, and the changes that were made to the preceding version. Changes to the first public release are made to clarify the assignment; the spirit or the crux of the assignment will not change.

Version	Date	Comments
1.0	10/14/2026	First public release of the assignment.

Appendix A: The Published Header: `ramsh.h`

This is the complete header, and it is the specification in C: every type, constant, message format, and function the autograder uses, each with its contract in a comment. It compiles cleanly under the flags in Appendix B. Do not edit it; the autograder copies the published version over yours before it builds. Take the file itself from the starter files rather than copying from this PDF.

```
/*
 * ramsh.h -- CS 370 Assignment 2: the published interface.
 *
 * DO NOT EDIT THIS FILE. The autograder copies the published version
 * over yours before it builds, then links its own unit-test programs
 * against your libramsh.a. Put any declarations of your own in a
 * separate header (for example, internal.h).
 *
 * Every function declared here must be defined in a source file that
 * the Makefile archives into libramsh.a. The two main() functions
 * (ramsh's and ramview's) must live outside libramsh.a, and nothing in
 * libramsh.a may depend on anything defined next to them.
 */
#ifndef RAMSH_H
#define RAMSH_H

#include <assert.h>     /* static_assert */
#include <stddef.h>     /* offsetof      */
#include <stdint.h>     /* int32_t, ...  */
#include <stdio.h>      /* FILE          */
#include <sys/types.h>  /* pid_t         */

#if !defined(_POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L
#error "Compile with -D_POSIX_C_SOURCE=200809L (the starter Makefile does)."
#endif

/* ================================================================
 * Limits
 * ================================================================ */
#define RAMSH_MAX_STAGES 8     /* stages in one pipeline               */
#define RAMSH_LINE_MAX   1024  /* input line, including the NUL        */
#define LEDGER_CAP       64    /* slots in the ledger ring             */
#define LEDGER_CMD_MAX   256   /* stored command line, including NUL   */

/* ================================================================
 * Output contract (handout: "Command-Line and Output Contract").
 * Everything below goes to stderr. Use these formats verbatim: the
 * autograder compares them exactly.
 * ================================================================ */
#define RAMSH_PROMPT    "ramsh> "
#define MSG_FORK        "ramsh: stage %d [pid %d] %s\n"
#define MSG_EXITED      "ramsh: stage %d [pid %d] exited %d\n"
#define MSG_SIGNALED    "ramsh: stage %d [pid %d] killed by signal %d\n"
#define MSG_EXEC_FAILED "ramsh: exec failed for %s: %s\n"
#define MSG_CD_FAILED   "ramsh: cd: %s\n"

/* ================================================================
 * Part 1: parsing
 * ================================================================ */
typedef struct {
    char **argv;   /* argc + 1 slots; argv[argc] == NULL; heap-owned  */
    int    argc;   /* at least 1 in every stage of a parsed pipeline  */
} command_t;

typedef struct {
    command_t stages[RAMSH_MAX_STAGES];
    int       nstages;             /* 0 unless parse_line gave PARSE_OK */
    char      raw[LEDGER_CMD_MAX]; /* the line as typed, up to its first
                                      '\n', cut to 255 bytes; filled for
```

```
                                    every parse_line() result          */
} pipeline_t;

typedef enum {
    PARSE_OK          =  0, /* 1..RAMSH_MAX_STAGES stages parsed       */
    PARSE_EMPTY       =  1, /* blank or whitespace only: do nothing    */
    PARSE_ERR_PIPE    = -1, /* leading, trailing, or doubled '|'       */
    PARSE_ERR_STAGES  = -2, /* more than RAMSH_MAX_STAGES stages       */
    PARSE_ERR_BUILTIN = -3  /* builtin as a command in 2+ stages       */
} parse_result_t;

/* Splits line on spaces, tabs, and '\n'. '|' is special only as a
 * whole token ("a|b" is one ordinary word). When a line has more than
 * one error, the first one in the enum order above is reported.
 * p need not be initialized; parse_line overwrites all of it. On any
 * result other than PARSE_OK, p owns no heap memory. On PARSE_OK the
 * caller must eventually call free_pipeline(p). Prints nothing. */
parse_result_t parse_line(const char *line, pipeline_t *p);

/* The exact stderr text (without the trailing newline) for an error,
 * or NULL for PARSE_OK and PARSE_EMPTY:
 *   PARSE_ERR_PIPE     "ramsh: syntax error near '|'"
 *   PARSE_ERR_STAGES   "ramsh: too many stages (max 8)"
 *   PARSE_ERR_BUILTIN  "ramsh: builtins cannot appear in a pipeline" */
const char *parse_errmsg(parse_result_t rc);

/* Frees every argv string and array and leaves p->nstages == 0.
 * Safe to call twice, and safe after any parse_line() result. */
void free_pipeline(pipeline_t *p);

/* ================================================================
 * Part 1: builtins
 * ================================================================ */

/* Nonzero if argv0 is "cd", "exit", or "ledger"; 0 otherwise. */
int is_builtin(const char *argv0);

#define RAMSH_EXIT_REQUESTED (-1)

/* Runs the builtin in p->stages[0]. Requires p->nstages == 1 and
 * is_builtin(p->stages[0].argv[0]).
 *   cd      chdir() in this process (no fork) to argv[1], or to $HOME
 *           when there is no argument. Returns 0, or prints
 *           MSG_CD_FAILED and returns 1.
 *   ledger  forks and execs "ramview <shmid>" through PATH, with the
 *           same trace lines as a one-stage pipeline (run_pipeline
 *           can do this for you). Returns ramview's decoded status.
 *   exit    returns RAMSH_EXIT_REQUESTED. It never calls exit(): the
 *           caller runs the single cleanup path it shares with
 *           end-of-file and -c mode.
 * Never records anything in the ledger. */
int run_builtin(const pipeline_t *p, int shmid);

/* ================================================================
 * Part 2: execution
 * ================================================================ */

/* Decodes a waitpid() status: the exit code for a normal exit, or
 * 128 + the signal number for a death by signal. */
int32_t decode_status(int wstatus);

/* Part 4 sabotage switch. Returns n if RAMSH_LEAK holds a decimal
 * integer n with 0 <= n < npipes; otherwise (unset, not a number, out
 * of range) returns -1. */
int leak_target(int npipes);

/* Runs p, which has 1..RAMSH_MAX_STAGES stages and no builtins:
 *  1. creates the N-1 pipes; pipe k joins stage k's stdout to stage
```

```
 *     k+1's stdin;
 *  2. forks every stage before waiting for any, and prints MSG_FORK
 *     for each stage, in stage order, right after its fork;
 *  3. in each child: dup2()s its pipe ends onto 0 and 1, closes every
 *     pipe descriptor, and calls execvp(); if that fails, prints
 *     MSG_EXEC_FAILED and calls _exit(127);
 *  4. in the parent: closes all 2(N-1) pipe descriptors before the
 *     first waitpid(), except the write end of pipe leak_target(N-1);
 *  5. calls waitpid() on each pid in stage order, printing MSG_EXITED
 *     or MSG_SIGNALED for each stage.
 * For k < p->nstages, fills pids_out[k] and status_out[k] (decoded).
 * Sets *elapsed_ms_out to CLOCK_MONOTONIC milliseconds from just
 * before the first fork to just after the last reap. Returns the last
 * stage's status. If pipe() or fork() fails, prints a message, closes
 * what it opened, reaps what it forked, and returns -1. */
int32_t run_pipeline(const pipeline_t *p, int32_t *pids_out,
                     int32_t *status_out, uint32_t *elapsed_ms_out);

/* ================================================================
 * Part 3: the shared-memory ledger
 * ================================================================ */
typedef struct {
    uint64_t seq;                      /* 1-based; never reused        */
    int32_t  nstages;
    int32_t  pids[RAMSH_MAX_STAGES];   /* valid for k < nstages        */
    int32_t  status[RAMSH_MAX_STAGES]; /* decode_status() values       */
    uint32_t elapsed_ms;
    char     cmdline[LEDGER_CMD_MAX];  /* NUL-terminated copy of raw   */
} ledger_entry_t;

typedef struct {
    uint64_t       total;              /* pipelines ever recorded      */
    ledger_entry_t ring[LEDGER_CAP];   /* seq s lives in               */
                                       /*   ring[(s - 1) % LEDGER_CAP] */
} ledger_t;

/* The layout is part of the contract: no pointers, fixed-width
 * fields, no padding. The build fails if it ever drifts. */
static_assert(offsetof(ledger_entry_t, pids)       == 12,  "layout");
static_assert(offsetof(ledger_entry_t, elapsed_ms) == 76,  "layout");
static_assert(offsetof(ledger_entry_t, cmdline)    == 80,  "layout");
static_assert(sizeof(ledger_entry_t)               == 336, "layout");
static_assert(sizeof(ledger_t) == 8 + LEDGER_CAP * 336,    "layout");

/* shmget(IPC_PRIVATE, sizeof(ledger_t), IPC_CREAT | 0600). Returns
 * the shmid, or -1 with errno set. New segments are zero-filled. */
int ledger_create(void);

/* shmat(). A nonzero readonly attaches with SHM_RDONLY, as ramview
 * must. Returns NULL (not shmat's (void *)-1) on failure, errno set. */
ledger_t *ledger_attach(int shmid, int readonly);

/* Appends one entry: seq = led->total + 1, stored in
 * ring[(seq - 1) % LEDGER_CAP]. Copies p->nstages, the first nstages
 * pids and statuses, elapsed_ms, and p->raw; then sets total = seq. */
void ledger_record(ledger_t *led, const pipeline_t *p,
                   const int32_t *pids, const int32_t *status,
                   uint32_t elapsed_ms);

/* Writes the viewer output to out: the header line, then every
 * surviving entry in ascending seq order:
 *   LEDGER total=<T> capacity=64 shmid=<id>
 *   #<seq> stages=<n> ms=<ms> status=<s0,s1,...> cmd=<line>
 * Reads led only. Returns 0, or -1 if writing to out failed.
 * ramview's main() is ledger_attach + ledger_print + ledger_detach. */
int ledger_print(const ledger_t *led, int shmid, FILE *out);

/* shmdt(). A NULL led is a no-op. */
```

```
void ledger_detach(ledger_t *led);

/* shmctl(shmid, IPC_RMID, NULL). A shmid of -1 is a no-op. */
void ledger_destroy(int shmid);

#endif /* RAMSH_H */
```

Appendix B: The Starter Makefile

The Makefile builds the two programs and `libramsh.a`, the static library the autograder links its unit tests against. Add your own `.o` files to `LIBOBJS` as you create them. The autograder runs

```
make CC=gcc CPPFLAGS=-D_POSIX_C_SOURCE=200809L \
     CFLAGS='-std=c23 -Wall -Wextra -Werror -g' all libramsh.a
```

with `gcc-toolset-15` enabled. Variables given on the `make` command line override the ones in your Makefile, so changing the flags in your copy does not change how you are graded. As with the header, take the file from the starter files: the recipe lines must begin with a tab, which a PDF does not preserve.

```
# CS 370 Assignment 2 -- starter Makefile (AlmaLinux 9.8)
#
# AlmaLinux 9's stock /usr/bin/gcc is GCC 11, which rejects -std=c23.
# Build with GCC 15 from gcc-toolset-15. Enable it once per shell:
#     source /opt/rh/gcc-toolset-15/enable
# (put that line in your ~/.bashrc so you never have to think about it)
#
# The autograder runs:  make CC=gcc CPPFLAGS=... CFLAGS=... all libramsh.a
# with exactly the values below. Variables given on the make command line
# override the ones in this file, so changing the flags here does not
# change how you are graded. Do not use the 'override' directive.

CC       := gcc
CPPFLAGS := -D_POSIX_C_SOURCE=200809L
CFLAGS   := -std=c23 -Wall -Wextra -Werror -g
AR       := ar

# Every function declared in ramsh.h is defined in one of these files.
# Add your own .o files here as you create them. main.o and ramview.o
# must never appear in this list.
LIBOBJS  := parse.o builtins.o exec.o ledger.o

.PHONY: all clean

all: ramsh ramview

ramsh: main.o libramsh.a
    $(CC) $(CFLAGS) -o $@ main.o libramsh.a

ramview: ramview.o libramsh.a
    $(CC) $(CFLAGS) -o $@ ramview.o libramsh.a

libramsh.a: $(LIBOBJS)
    $(AR) rcs $@ $^

%.o: %.c ramsh.h
    $(CC) $(CPPFLAGS) $(CFLAGS) -c -o $@ $<

clean:
    rm -f ramsh ramview libramsh.a *.o

# Fail early, with a useful message, if the compiler cannot do C23.
ifneq ($(MAKECMDGOALS),clean)
  C23 := $(shell echo | $(CC) -std=c23 -xc -E - >/dev/null 2>&1 && echo y)
```

```
  ifneq ($(C23),y)
    $(error $(CC) cannot compile C23; first run \
            source /opt/rh/gcc-toolset-15/enable)
  endif
endif
```

Appendix C: How the Autograder Works

The autograder settles the 9 correctness points. It does not grade the prompt logs or the quiz.

C.1 Where it runs

Each submission is graded in its own fresh AlmaLinux 9.8 container with the same compiler as the lab machines (GCC 15 from `gcc-toolset-15`), no network access, and limits on processes and memory. The container has its own System V namespace, so the segment checks see only your shell's segments. The grader runs as an ordinary user, not root: root ignores segment permissions, and one test depends on them.

C.2 What one run does

1. **Copy and pin.** Your repository is copied to a scratch directory and `ramsh.h` is replaced with the published copy.
2. **Build gate.** `make` runs with the canonical flags shown in Appendix B. If the build fails, every test scores 0, and the build log is included in your report.
3. **Interface check.** The unit tests are compiled against the published header and linked against your `libramsh.a`. If a function is missing or defined next to a `main()`, the link fails and the unit tests (P1.1, P1.2, P1.6, P3.3, P4.3) score 0; everything else still runs.
4. **Tests.** Each test runs `ramsh` or `ramview` from the build directory with `PATH` set to the build directory followed by `/usr/bin:/bin`, `HOME` set to a scratch directory, and `LC_ALL=C`. A test that expects a pipeline to finish gives it a time limit (3 to 10 seconds, or 120 under valgrind). If the limit expires, the grader kills the shell's whole process group and the test fails. The `RAMSH_LEAK` tests invert this: they pass only if the limit expires. After every test, the grader removes any segment the test left behind, so each test is judged on its own; only P3.7 scores leftover segments.
5. **Normalization.** Prompts are removed from `stderr`, `[pid N]`, `ms=`, and `shmid=` values are replaced by placeholders, and `exec failed` lines are checked separately (Section 6). Everything else is compared exactly.
6. **Report.** You receive a per-test pass or fail report, with a unified diff of expected and actual output for every exact-match failure.

The tests are written so that a bug costs points in the tests aimed at it, not everywhere: forgetting `SHM_RDONLY`, exec'ing `./ramview`, or never destroying the segment each fails exactly one test. The exception is the executor itself. A child that keeps its pipe descriptors open breaks every multi-stage pipeline, and it loses every test that runs one.

C.3 Test catalog

Tests marked **P** are public: they are posted with the starter files, and you can run them yourself.

ID	What it checks	Pts	Pub
P1.1	Unit: `parse_line` tokens, `argv` termination, `raw` line and its 255-byte cut, `is_builtin`	0.25	P
P1.2	Unit: blank lines, the three syntax errors and their precedence, `parse_errmsg` text	0.25	P
P1.3	`-c "echo hello"`: `stdout`, trace, status 0	0.25	P
P1.4	`-c nosuchprog`: `exec failed` message, status 127	0.25	P
P1.5	Script read from a file, with an exec failure first: every later line runs exactly once	0.25	
P1.6	Unit: `decode_status`; `-c` statuses for `false`, `nosuchprog | wc -l`, a syntax error, a failed `cd`	0.25	
P1.7	`cd dir`, bare `cd` (to `$HOME`), failed `cd`: no fork lines, correct message	0.25	
P1.8	Blank lines, the `exit` builtin (later lines never run), end-of-file: status 0	0.25	
P2.1	`seq 3 | sort -r | head -n 1`: exact trace	0.25	P
P2.2	`seq 1000000 | wc -l` within 10 seconds	0.5	P
P2.3	Eight stages, 100,000 lines: output and trace	0.5	P
P2.4	`ls /proc/self/fd` as the only, first, middle, and last stage: exactly `0 1 2 3`	0.5	P
P2.5	`yes | head -n 1` finishes; stage 0 `killed by signal 13`	0.25	
P2.6	`sleep 0.3 | true`: reap lines in stage order, not completion order	0.25	
P2.7	Syntax errors (all three kinds, and nine stages) run nothing and record nothing	0.5	P
P2.8	`nosuchcmd | wc -l`: output `0`, trace, statuses 127 and 0	0.25	
P3.1	The worked example of Section 6, exactly	0.5	P
P3.2	70 pipelines, then `ledger`: `total=70`, entries `#7` to `#70`	0.25	P
P3.3	Unit: `ledger_record` and `ledger_print` on plain memory, including wraparound	0.25	
P3.4	`ramview` on a segment the grader built with mode `0400` (read-only attach)	0.25	
P3.5	`ramview` with a bad or missing segment ID: status 1, message on `stderr`, nothing on `stdout`	0.25	
P3.6	`ledger` after `cd /`: `ramview` found through `PATH`	0.25	
P3.7	No segment remains after `exit`, end-of-file, or `-c` (including failures)	0.25	
P4.1	`echo done | wc -l`: finishes normally; hangs with `RAMSH_LEAK=0`	0.5	P
P4.2	`RAMSH_LEAK=1` hangs a three-stage pipeline; values `2`, `abc`, and `0` on one stage are ignored	0.25	
P4.3	Unit: `leak_target`	0.25	
Q1	Valgrind memcheck on a session and a `-c` run: no errors, no definite or indirect leaks	0.5	
Q2	No zombie children between pipelines (`ps --ppid` run by the shell itself)	0.25	
Q3	The shell's own `/proc/<pid>/fd` is exactly `0 1 2` between pipelines	0.25	
	Total	**9**	

Part 4 tests (P4.1–P4.3) score only when `forensics.md` is present in the repository root.

C.4 Running the public tests yourself

On a lab machine, from your repository root:

```
source /opt/rh/gcc-toolset-15/enable
python3 /path/to/grade.py --public .
```

The public run uses the same build flags, time limits, and normalization as the real one. A clean pass takes under ten seconds, most of it spent waiting out the deliberate hang in P4.1.