CSCE 313
Introduction to Computer Systems

Texas A&M University • Fall 2026

LAB 2

The Process Abstraction and Creating Processes

Student Companion Guide

Step-by-step walkthrough, explanations,
and the reasoning behind every command

This guide accompanies the official lab handout. Where the two differ,
the handout governs the grade — but read the notes here first.

September 9, 2026

Contents

1 How to use this guide **3**

1.1 The colour key . . . 3

1.2 What you need before you start . . . 4

1.3 The one rule about scope . . . 4

2 Setup and pre-flight **6**

2.1 What is in the repository . . . 6

2.2 Installing the toolchain . . . 6

2.3 The Makefile picks the IR file for you . . . 7

2.4 First build . . . 7

2.5 Running the servers by hand . . . 8

3 The process abstraction **9**

3.1 A process is a program plus its private world . . . 9

3.2 `fork()` is called once and returns twice . . . 9

3.3 What the child inherits, and what it does not . . . 10

3.4 `execvp()` replaces the program, not the process . . . 10

3.5 The channel is a pair of named pipes . . . 11

4 Task 1 — Running the servers as child processes **12**

4.1 Task 1 — Fork three children and replace each with a server . . . 12

4.2 The file server's argument list . . . 13

5 Task 2 — Printing process details **17**

5.1 Task 2 — Fill in `print_process_info` so each line carries a real value . . . 17

6 Task 3 — Client–server communication **21**

6.1 Task 3 — Open the channels and fill in every menu action . . . 21

6.2 Opening the channels . . . 21

6.3 The two upload traps . . . 23

7 Task 4 — Closing the channels **26**

7.1 Task 4 — Send `QUIT` down all three channels before returning 26

8 Verifying your work **29**

8.1 Run the grader yourself 29

8.2 Reproduce the reference session by hand 30

8.3 Reading a failure 31

9 Deliverables and how points get lost **32**

9.1 What you submit 32

9.2 The complete list of ways to lose points 32

9.3 Final checklist 33

10 Quick reference **35**

10.1 The request types 35

10.2 The two structs 35

10.3 Server command lines 36

10.4 Commands you will use 36

10.5 Symptom to chapter 36

10.6 Getting help 37

1 How to use this guide

This document sits between the Lab 2 handout and your editor. The handout tells you what to build; this guide tells you how the pieces work, what to type, and which mistakes will cost you an afternoon. Everything in it was verified by building and running the actual code, on the actual toolchain, including every failure mode described.

1.1 The colour key

Every task in this guide is presented the same way, in the same order. Once you learn the colours you can skim for exactly the part you need.

Violet — Commands

A short table of the commands and functions this task uses, with a one-line description of each. Read this first if you have forgotten the syntax.

Maroon — Task

What the assignment is asking for, in plain language, plus the clue the handout buried. Read this to know when you are finished.

Green — How to solve it

Exact code and exact commands. Copy-pasteable. If you only want to finish the lab, the green boxes are enough.

Blue — What is actually happening

The mechanism one layer below what you can see, and the consequence of it. This is the part the exam asks about.

Amber — Where you will use this for real

Concrete situations outside this course where the same mechanism shows up.

Violet — The intuition

The mental model that survives after you have forgotten the syntax.

Maroon — Takeaway

Three or four sentences with the specific values, messages and rules. Reading only the takeaway boxes should let you reconstruct the whole lab.

Four support boxes appear wherever they are needed:

Red — Common mistake

Something that costs points or wastes an afternoon. Where possible these quote real terminal output, so that pasting your error into a search finds this page.

Grey — Note

Something that looks wrong but is expected. Read these before asking on Discord; several of them describe behaviour that is not a bug in your code.

Blue — TA tip

Optional depth for the curious. Never a required step.

Green — Checkpoint

Commands that confirm a section actually works before you move on.

1.2 What you need before you start

- A Linux machine — x86-64 or ARM. A lab machine, WSL, or a VM are all fine. macOS is not: the lab uses named pipes and `/proc`-style process semantics that this code assumes.
- `clang` (to assemble the provided channel implementation) and `g++` (for everything else). Chapter 2 covers the version requirements, which differ between x86-64 and ARM.
- Your accepted Lab 2 repository, cloned.
- Comfort with a terminal, and roughly four to six hours. The lab is not long, but two of its failure modes present as a program that hangs in silence, and those are slow to diagnose if you have not been warned. You have been warned in Chapters 4 and 7.

1.3 The one rule about scope

You modify `client.cpp` and nothing else. The three servers, the channel implementation, the `Makefile` and the test script are complete and correct. If a server appears to be misbehaving, the cause is in the request your client sent it. This guide points out the three places where that is most likely.

On reading the server source

You are not required to read `finance.cpp`, `file.cpp` and `logging.cpp`, but each is under a hundred lines and each answers a question this lab will raise. Where a server's behaviour

> matters, this guide quotes the relevant lines directly, so you do not have to go looking.

2

Setup and pre-flight

2.1 What is in the repository

Path	What it is
`client.cpp`	The only file you modify. Every `TODO` is here.
`finance.cpp`	Finance server. Complete — do not modify.
`file.cpp`	File server. Complete — do not modify.
`logging.cpp`	Logging server. Complete — do not modify.
`channel.h`	The `RequestChannel` interface you call.
`channel.ll`, `channel.ll.arm`	Its implementation, as LLVM IR, for x86-64 and ARM. Never rename these.
`common.h`, `common.cpp`	`Request`, `Response`, `RequestType`.
`Makefile`	Builds all four programs.
`lab2-tests.sh`	The same tests the autograder runs.
`storage/example_execution.txt`	A full correct session.
`storage/example.log`	The log that session produced.

2.2 Installing the toolchain

The `RequestChannel` implementation ships as LLVM intermediate representation rather than as C++ source, so `clang` assembles it. You do not need `clang` for anything else, and you never edit the IR.

```
$ sudo apt update
$ sudo apt install clang
```

Which version you need depends on your architecture, and this is the single most common way to lose an evening before writing any code at all.

Architecture	IR file used	clang requirement
x86-64	`channel.ll`	Any clang 18 or newer. This is what `apt install clang` gives you on Ubuntu 24.04.
ARM (aarch64)	`channel.ll.arm`	clang **20 or newer**. Ubuntu 24.04's default clang 18 fails.

Check which machine you are on:

```
$ uname -m
$ clang --version
```

ARM with clang 18: `error: expected type`

If `uname -m` prints `aarch64` and your clang is older than 20, the build stops while assembling the IR with a message of the form `error: expected type`. The IR file was generated by a newer compiler and uses syntax clang 18 cannot parse. Nothing is wrong with your code — you have not written any yet.
Install a newer clang and tell `make` to use it:

```
$ sudo apt install clang-20
$ make CLANG=clang-20
```

If no newer clang is available to you, move to a lab machine or any x86-64 Linux box, and tell the instructor.

2.3 The Makefile picks the IR file for you

```
ARCH := $(shell uname -m)
ifeq ($(filter aarch64 arm64,$(ARCH)),)
CHANNEL_LL = channel.ll
else
CHANNEL_LL = channel.ll.arm
endif
```

Never rename `channel.ll` or `channel.ll.arm`

In previous terms, students on ARM renamed `channel.ll.arm` to `channel.ll` by hand, then had to remember to rename it back before pushing. Forgetting meant the grader — which runs on x86-64 — assembled the ARM IR and failed every single test.
The selection above is automatic. There is nothing to rename and nothing to undo. If you have already renamed something, restore both original names before you push.

2.4 First build

```
$ make
```

A successful build compiles six objects and links four executables. Note the line that uses `clang` rather than `g++` — that is the IR step:

```
g++ -std=c++17 -g -Wall -Wextra ... -c client.cpp -o build/obj/client.o
g++ -std=c++17 -g -Wall -Wextra ... -c common.cpp -o build/obj/common.o
clang -Wall -Wextra -x ir -c channel.ll.arm -o build/obj/channel.o
g++ build/obj/client.o build/obj/common.o build/obj/channel.o -o client
g++ ... -o finance
g++ ... -o fileserver
g++ ... -o logging
```

You now have four executables: `client`, `finance`, `fileserver` and `logging`.

The file server's executable is `fileserver`, its channel is `"file"`

> The executable cannot be called `file`, because `file` is already a standard Unix command. Its *channel*, however, is named `"file"`. Both spellings are correct in their own place, and mixing them up is the subject of a warning in Chapter 4.

2.5 Running the servers by hand

Your finished client starts the servers itself. While you are still building Task 1, you can run each server in its own terminal and drive it from a fourth:

```
$ ./finance -m 1000 # accounts 0 through 1000
$ ./logging -f system.log # append log lines to system.log
$ ./fileserver .txt .h # allow uploads of .txt and .h only
$ ./client
```

This is a debugging aid and not the deliverable. The autograder fails every test if the client does not start the servers itself, so implement Task 1 first and use this only when you are isolating a problem.

Confirm your environment before writing code

```
$ uname -m && clang --version | head -1
$ make && ls -l client finance fileserver logging
```

All four executables should exist. If they do, the toolchain problem class is behind you and every later failure is in your own code.

3

The process abstraction

Tasks 1 and 2 are two halves of one idea, and they are much easier to write once that idea is clear. This chapter is the background. It contains no deliverables.

3.1 A process is a program plus its private world

A program is a file on disk. A process is that program *running*, with its own memory, its own open files, and its own identity. The identity is a number, the PID, and every process also records the PID of the process that created it, its PPID.

The word "private" is doing real work in that sentence. Two processes running the same program do not share variables. They do not share a heap. They do not even share the meaning of an address — and that last point is the one Task 2 makes you prove.

3.2 `fork()` is called once and returns twice

```c
pid_t pid = fork();
if (pid < 0) { /* fork failed */ }
if (pid == 0) { /* the child */ }
if (pid > 0) { /* the parent */ }
```

After `fork()` there are two processes. Both are sitting on the same line of the same program with the same variables holding the same values. The *only* difference between them is the value `fork()` handed back: the child receives `0`, and the parent receives the child's PID.

That asymmetry is the entire mechanism by which a process learns which one it is. There is no other signal. If you do not branch on the return value, both processes run whatever comes next — which is precisely the bug described in Chapter 4.

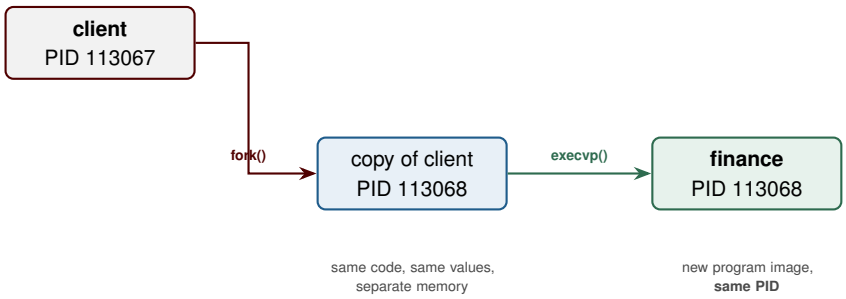

Figure 3.1: One `fork()` and one `execvp()`. Notice that the PID does not change across `execvp()`: the process survives, only the program running inside it is replaced. Your client does this three times.

3.3 What the child inherits, and what it does not

Inherited (a copy, or shared)	Not shared
The values of every variable at the moment of the fork	Any change made after the fork — writes are private to whoever made them
Open file descriptors, and the current working directory	The PID; the child gets a new one
The program image and the next line to execute	The parent's later view of memory

The subtlety is that inheritance is a snapshot, not a link. The child begins with your `global_var` equal to 100 because that is what it was when the fork happened. The instant either process writes to it, the two values part company forever.

Copy-on-write, if you are curious

> The kernel does not actually duplicate the parent's memory during `fork()`. It marks the pages read-only and shares them, then copies an individual page only when one of the two processes writes to it. The behaviour you observe — private memory — is exactly as if a full copy had been made, which is why the abstraction is worth having. Nothing in this lab depends on knowing this, but it explains why forking a large process is fast.

3.4 `execvp()` replaces the program, not the process

```
char* args[] = {(char*)"./program", (char*)"arg1", (char*)"arg2", nullptr};
execvp(args[0], args);
perror("execvp"); // only reached if execvp FAILED
exit(1);
```

Three properties matter, and each one causes a distinct bug when forgotten.

The array must end with `nullptr`. That is how `execvp` knows where the argument list stops. Without it, it reads past the end of your array into whatever happens to be next in memory.

`args[0]` is the program's own name, by convention. The operating system does not require this, but every program in the world assumes it — which is why the servers' own argument parsing loops all start at `i = 1`. Look at `finance.cpp`:

```
22    for(int i = 1; i < argc; i++) {
23       string arg = argv[i];
24       if(arg == "-m" && i + 1 < argc) {
25          max_accounts = atoi(argv[++i]) + 1;
26       }
27    }
```

If you put `"-m"` in `args[0]`, the loop never sees it and the server silently uses its default of 100 accounts.

A successful `execvp` never returns. There is no code left to return to — it was replaced. So every line you write after `execvp` runs only on failure, and that is exactly why the `perror` and `exit(1)`

above are there rather than being dead code.

Keep the strings alive

A temporary `std::string` is destroyed at the end of the statement that created it, so this leaves `args[2]` pointing at freed memory:

```
char* args[] = {..., (char*)std::to_string(n).c_str(), nullptr};
```

Store it in a named variable first, then take `.c_str()`:

```
string max_account_str = to_string(max_account);
char* args[] = {(char*)"./finance", (char*)"-m",
                (char*)max_account_str.c_str(), nullptr};
```

This is worth avoiding precisely because it usually appears to work: the freed bytes are typically still intact when `execvp` reads them microseconds later. It fails on a different machine, under a different compiler, or on the grader.

3.5 The channel is a pair of named pipes

`RequestChannel` hides its implementation, but knowing the shape of it explains two behaviours you will otherwise find baffling. Start your client and look at the directory while it runs:

```
$ ls -l fifo_*
prw-rw-r-- 1 user user 0 Sep 9 00:32 fifo_file_1
prw-rw-r-- 1 user user 0 Sep 9 00:32 fifo_file_2
prw-rw-r-- 1 user user 0 Sep 9 00:32 fifo_finance_1
prw-rw-r-- 1 user user 0 Sep 9 00:32 fifo_finance_2
prw-rw-r-- 1 user user 0 Sep 9 00:32 fifo_logging_1
prw-rw-r-- 1 user user 0 Sep 9 00:32 fifo_logging_2
```

The leading `p` in `prw-rw-r-` means *named pipe*, not regular file. Each channel is two of them — one for each direction — and the channel's name is literally part of the filename. Two consequences follow immediately:

- **The names must match exactly.** A client that opens `"Finance"` and a server that opens `"finance"` are using different files and will never meet.
- **Opening a pipe blocks until the other end opens it too.** This is why a client whose server failed to start does not report an error — it stops, silently, inside the `RequestChannel` constructor. Chapter 4 shows exactly what that looks like.

Leftover `fifo_*` files

If a run is killed part-way, the pipe files stay behind. They are harmless, and `make distclean` removes them. The test script removes them between tests for you.

4

Task 1 — Running the servers as child processes

45 points — the largest single block in the lab, and nothing else can be tested until it works.

4.1 **Task 1** — Fork three children and replace each with a server

Commands used in this task

Call	What it does
`fork()`	Duplicates the calling process. Returns `0` in the child, the child's PID in the parent, `-1` on failure.
`execvp(f, argv)`	Replaces the current process image with program `f`, passing the `nullptr`-terminated array `argv`. Never returns on success.
`perror(s)`	Prints `s`, then a colon, then the human-readable text of the last error.
`exit(n)`	Terminates this process with status `n`.

What this task asks you to do

Make `./client` bring the whole system up on its own: three `fork()` calls, each child replacing itself with one of the three servers, the parent carrying on to start the next.
The clue the handout buries is in the starter file's *indentation*. Under each `TODO` there is a block of lines that bumps `global_var`, `stack_var` and `*heap_var` and then calls `print_process_info`. That block is indented as though it were already inside an `if`. It is not — there is no `if` in the file yet. Putting one around it is your job, and forgetting is the single most expensive mistake in this lab.

How to solve it

Each of the three follows the same five-step shape. Fork, check for failure, open a child-only block, keep the given lines inside it, then exec.
The finance server, at the first `TODO`:

```
75      // Start finance server
76      pid_t finance_pid = fork();
77      if (finance_pid < 0) {
78          perror("fork (finance)");
79          exit(1);
```

```
    }
    if (finance_pid == 0) {
        // Modify these values to demonstrate memory isolation
        global_var++;
        stack_var++;
        (*heap_var)++;

        // Do not change this string
        empty_file("finance_attributes.txt");
        print_process_info("Finance process before exec:", heap_var,
                           stack_var, &stack_var, "finance_attributes.txt");

        string max_account_str = to_string(max_account);
        char* args[] = {(char*)"./finance",
                        (char*)"-m",
                        (char*)max_account_str.c_str(),
                        nullptr};
        execvp(args[0], args);
        perror("execvp (finance)");
        exit(1);
    }
```

The logging server is the same shape. Its filename is already a named `std::string`, so no temporary is involved:

```
    char* args[] = {(char*)"./logging",
                    (char*)"-f",
                    (char*)log_file_name.c_str(),
                    nullptr};
```

The file server differs only in how `args` is built — see the next section.
The parent writes no code at all in this task. It falls out of each `if` block and carries on to the next server.

4.2 The file server's argument list

The other two servers take a fixed two arguments. The file server takes one argument per allowed extension, and the count is whatever the user typed seconds earlier. A fixed-size array cannot express that, so build it on the heap:

```
    // ./fileserver + one slot per extension + the terminating nullptr
    char** args = new char*[extensions.size() + 2];
    args[0] = (char*)"./fileserver";
    for (size_t i = 0; i < extensions.size(); i++) {
        args[i + 1] = (char*)extensions[i].c_str();
    }
    args[extensions.size() + 1] = nullptr;

    execvp(args[0], args);
    perror("execvp (fileserver)");
    delete[] args;
    exit(1);
```

The `+ 2` is the part to get right: one slot for the program name and one for the terminating `nullptr`. The strings themselves are safe because `extensions` is a live `vector<string>` that outlives the

call.

The `delete[]` is unreachable on success, and that is fine

`execvp` does not return when it works, so the `delete[]` runs only on the failure path. Freeing it there is tidy rather than necessary — the `exit(1)` on the next line would release it anyway. The leak that would matter is a leak in a process that keeps running, and this one does not.

`./file` is not `./fileserver`

The channel is named `"file"` and the executable is called `fileserver`. Writing the channel name into `args[0]` produces this, buried in the middle of the extension prompts:

```
Enter allowed extensions (including the dot, e.g. .txt):
1: execvp: No such file or directory
```

and then the client **hangs forever with no further output**. It never reaches the menu.
The hang is the confusing part, and it is worth understanding rather than memorising. The exec failed, so that child exited; no file server exists. Your client then reaches `RequestChannel file("file", CLIENT_SIDE)`, and opening a named pipe blocks until something opens the other end. Nothing ever will. Measured on the reference machine: the client was still blocked when a 25-second timeout killed it, exit status 124.
The lesson generalises. For the rest of this lab, "the menu never appeared" means a server did not start; it almost never means the menu code is broken.

What is actually happening

Look again at what the three blocks have in common: each calls `print_process_info` *after* `fork()` but *before* `execvp()`. That window is the only moment in the program's life when the child exists as a copy of the client. One instruction later, `execvp` throws that copy away and loads `./finance` over the top of it. The attributes files are written from inside that window, which is why they can report the client's variables at all.
This is also why the PID does not change across the exec. `fork()` creates a process; `execvp()` changes what is running inside one. They are different operations, and Unix keeps them separate deliberately — the gap between them is where a shell sets up redirections, changes directory, or drops privileges before the new program ever starts. Your client uses that gap to write a file.
The consequence to hold on to: **everything the child does between fork and exec is invisible to the parent, and everything after exec is a different program entirely.**

The most expensive mistake in the lab

If you put the `if (pid == 0)` around only the `execvp` and leave the given lines above it unguarded, the parent runs them too. What makes this so costly is that it does *not* look broken:

```
=== Process Information Tests ===
[PASS] Finance process info test passed: 5/5 points
  Details: Correct parent-child relationship and memory isolation
[FAIL] Logging process info test failed: 0/5 points
  Error: Incorrect process info relationships
[FAIL] File process info test failed: 0/5 points
  Error: Incorrect process info relationships
```

Finance passes. Both later servers fail. The reason is cumulative corruption: the parent incremented `global_var` to 101 along with the first child, so when it forks the second child that child starts from 101 and reports **103** where the grader expects 102. The third reports 104 instead of 103.

Occasionally you will also see `finance_attributes.txt` come out with fourteen lines instead of seven, because parent and child raced to write the same file. That intermittency is a hallmark of this bug.

Symptom to remember: the first server passes and the other two fail. The fix is to move the opening brace up, above `global_var++`.

Where you will use this for real

- Every process on your machine was created this way. Your shell forks and execs for every command you type; that is why `cd` has to be a shell builtin rather than a program, since a child cannot change its parent's directory.
- Web servers of the pre-thread era — and PostgreSQL today — fork a worker per connection, so that one crashed request cannot corrupt the others' memory.
- Container runtimes fork, then adjust namespaces and cgroups in the gap before exec. Docker's isolation lives entirely in that window.
- CI runners, build systems and `make -j` all fan out work by forking and exec'ing compilers, then waiting on the results.
- When a deployment script hangs with no output, the reason is very often the one you met above: a child died and the parent is blocked on a pipe waiting for it.

The intuition

`fork()` is photocopying yourself; `execvp()` is the copy changing clothes. The copy is identical at the instant it is made — same memory, same place in the code, same everything but its name — and the only way it can tell it is the copy is that `fork()` handed it a zero. Then `execvp` replaces its entire body with a different program, while keeping the same name badge: the PID never changes.

The reason Unix splits one operation into two is that the seam between them is useful. Everything a shell does to prepare a command — redirect output, change directory, close file descriptors — happens in that seam, in a process that is about to stop being the shell. Your client writes a file in that seam. A container runtime builds an entire sandbox in it.

Takeaway

Three forks, three children, one `execvp` each; the parent writes nothing but the forks themselves. The child is identified only by `fork()` returning `0`, and the given block that bumps

`global_var`, `stack_var` and `*heap_var` must be inside that `if` — if it is not, the finance test passes and the logging and file tests fail with `Incorrect process info relationships`. Argument arrays end with `nullptr`, `args[0]` is the program's own name, and the file server's array is heap-allocated at `extensions.size() + 2` because its length is not known until run time. Anything after `execvp` runs only on failure, so that is where `perror` belongs. A client that prints `execvp: No such file or directory` and then hangs before the menu has a server that never started.

Checkpoint

After writing all three, rebuild and start the client. You should reach the menu, which means all three channels opened, which means all three servers are alive:

```
$ make && ./client
$ ps -o pid,ppid,comm -C finance,logging,fileserver
```

The `PPID` column should show your client's PID three times.

5

Task 2 — Printing process details

15 points — and the part of the lab that shows you the process abstraction directly.

5.1 Task 2 — Fill in `print_process_info` so each line carries a real value

Commands used in this task

Call	What it does
`getpid()`	Returns this process's own PID.
`getppid()`	Returns the PID of this process's parent.
`&x`	The address of `x`. Streaming a pointer to `ofstream` prints it in hex.
`*p`	The value `p` points at.

What this task asks you to do

The starter's `print_process_info` writes the right labels with nothing after them. Fill in each one so it prints a real PID, a real PPID, and an address followed by a value for each of the global, stack and heap variables.
Do not change the label strings and do not reorder the lines. The test script reads these files positionally — by line index, not by searching for a label — so a reordered file fails even though it looks correct to you.

How to solve it

The function already receives everything it needs. `global_var` is a global so it is simply in scope; `heap_var` is the pointer itself; `stack_var` is the value and `stack_var_addr` the address.

```
22 void print_process_info(const string& label, int* heap_var, int stack_var,
23                         int* stack_var_addr, const string& outfile) {
24     ofstream f(outfile, ios::app);
25     f << label << endl;
26     f << "PID: " << getpid() << endl;
27     f << "PPID: " << getppid() << endl;
28     f << "Global variable address: " << &global_var
29       << " value: " << global_var << endl;
30     f << "Stack variable address: " << stack_var_addr
31       << " value: " << stack_var << endl;
32     f << "Heap variable address: " << heap_var
33       << " value: " << *heap_var << endl;
34     f << "----------------------------------------" << endl;
```

```
35      f.close();
36  }
```

Note the asymmetry in the last three lines, because it is easy to get wrong: the global needs `&` to take its address, the heap pointer needs `*` to read its value, and the stack variable arrives pre-split into an address and a value as two separate parameters.

Running the client now produces four files. Here is the parent's first block next to the finance child's, from a real run:

```
parent_attributes.txt finance_attributes.txt
Parent process before fork: Finance process before exec:
PID: 113067 PID: 113068
PPID: 113066 PPID: 113067
Global ... 0xb1c2859d0010 value: 100 Global ... 0xb1c2859d0010 value: 101
Stack ... 0xfffff999082c value: 200 Stack ... 0xfffff999082c value: 201
Heap ... 0xb1c28c739020 value: 300 Heap ... 0xb1c28c739020 value: 301
```

Three things to check, and the third is the point of the whole task:

1. Every child's PPID equals the parent's PID. They really are its children.
2. Every child's values differ from the parent's and from each other's — 101/201/301 for finance, 102/202/302 for logging, 103/203/303 for the file server.
3. Every child's addresses are **identical** to the parent's.

Two processes, one address, two different values

That third observation should look impossible. `0xb1c2859d0010` holds `100` in one process and `101` in another, at the same instant.
It is possible because the number is not an address in RAM. It is a *virtual* address, and every process has its own table mapping virtual addresses to physical ones. The hardware consults that table on every single memory access, and the table is swapped when the kernel switches processes. So the same number means different physical memory depending on who is asking.
Two consequences follow, and both are worth more than the fifteen points:
Isolation is the default, not an achievement. A wild pointer in one process cannot corrupt another, because there is no arrangement of bits it could hold that names another process's memory. Isolation is not enforced by checking — it is a property of the address not existing.
Nothing needed to be copied. The addresses match because the child began as a copy-on-write duplicate of the parent's page tables. The single increment of `global_var` is what forced the kernel to make a private copy of that one page. Had the child never written, no copy would have been made at all.

Your addresses will not match the ones printed here

Linux randomises where the stack, heap and globals land on every run, as a security measure. Two runs of your own client will disagree, and yours will not match this guide's. What must hold is the *relationship*: the parent and its children agree with each other within a single run. If they do, the task is right.

The handout hedges on the addresses; here is what actually happens

The handout says each child's addresses are "the same as the parent's, or nearly so." On the reference machine all three matched the parent *exactly*, in every run, for all three children — there was nothing approximate about it. That is what you should expect too, because the child inherits the parent's page tables wholesale.

The hedge only becomes relevant if you move the `print_process_info` call to a different call depth, which would change where `stack_var` lives. Leave the calls where the starter puts them and all three addresses will agree.

Where you will use this for real

- Every debugger you will use prints virtual addresses. When two `gdb` sessions on two processes show the same pointer, they are looking at different memory — a genuinely common confusion when debugging a forking server.
- Address space layout randomisation is virtual memory used as a defence: an attacker who cannot predict where the stack landed cannot hard-code a return address into an exploit.
- `mmap` of the same file into two processes gives them different virtual addresses for the same physical pages — the basis of shared libraries, which is why one copy of libc in RAM serves every running program.
- Out-of-memory kills on a server are usually about physical pages, not virtual ones. A process can reserve far more address space than the machine has RAM, and frequently does.
- "It works on my machine, crashes on the server" is very often a pointer bug whose damage lands on a different page under a different allocator.

The intuition

A street address is not a house. "12 Oak Street" means a different building in every town, and knowing the number tells you nothing about which building without also knowing the town. A virtual address is the number; the process is the town; the page table is the map from one to the other.

This inverts the assumption most people arrive with — that a pointer is a location in the machine. It is not. It is a location in a *story the kernel tells this process*, and each process is told a different story with the same words. That is why your four files can print one address and four different values, and why nothing in this lab could have leaked from one server into another even if you had tried.

Takeaway

`getpid()` and `getppid()` fill the first two lines; the global needs `&global_var` for its address, the heap variable needs `*heap_var` for its value, and the stack variable arrives already split across two parameters. Labels and line order are fixed, because `lab2-tests.sh` reads these files by line index. A correct run gives every child a PPID equal to the parent's PID, values of 101/201/301, 102/202/302 and 103/203/303, and addresses matching the parent's exactly for the global and the heap. Same address, different value, is not a bug — it is virtual memory, and it is the reason process isolation costs nothing.

Checkpoint

```
$ head -6 parent_attributes.txt
$ for f in finance logging file; do sed -n '2,6p' ${f}_attributes.txt; echo;
    done
```

Confirm the PPIDs, then confirm the values step 101, 102, 103 across the three children. If finance is right and the other two are too high, re-read the red box in Chapter 4.

6

Task 3 — Client–server communication

30 points — seven menu actions, three servers, one trap.

6.1 Task 3 — Open the channels and fill in every menu action

Commands used in this task

Call	What it does
`RequestChannel(name, side)`	Opens one end of a named channel. Use `RequestChannel::CLIENT_SIDE`.
`send_request(req)`	Sends a `Request` and blocks until the `Response` comes back. The only channel call you make.
`Request(type, uid, amt, fname, data)`	All but `type` default, so `Request(QUIT)` and `Request(BALANCE, 7)` are both valid.

`receive_request` and `send_response` are the servers' half of the conversation. You never call them.

What this task asks you to do

Create the three client-side channels after the forks, then fill in each menu action so it sends the right request to the right server and stores the reply in the `resp` variable the starter already tests.
Almost every action talks to *two* servers: the one that owns the resource, and the logging server that records that it happened. The audit request always goes second, and always inside the `if (resp.success)` branch that is already written for you — you log what happened, not what was attempted.

6.2 Opening the channels

```
RequestChannel finance_channel("finance", RequestChannel::CLIENT_SIDE);
RequestChannel file_channel("file", RequestChannel::CLIENT_SIDE);
RequestChannel logging_channel("logging", RequestChannel::CLIENT_SIDE);
```

The names come from the servers' own source and must match character for character. Note again that the file server's channel is `"file"` while its executable is `fileserver`. Each name is opened exactly once from each side: one `CLIENT_SIDE` here, one `SERVER_SIDE` in the server. Opening the same side twice is undefined behaviour.

How to solve it

Menu action	To finance or file	To logging
Login	—	`LOGIN`
Deposit	`DEPOSIT` with `amount`	`DEPOSIT`, same amount
Withdraw	`WITHDRAW` with `amount`	`WITHDRAW`, same amount
Balance	`BALANCE`	`BALANCE` with the *returned balance* in `amount`
Upload	`UPLOAD_FILE` with filename and data, to **file**	`UPLOAD_FILE` with filename
Download	`DOWNLOAD_FILE` with filename, to **file**	`DOWNLOAD_FILE` with filename
Logout	—	`LOGOUT`

Login and logout are pure bookkeeping — there is no finance request, because nothing in the finance server has any notion of a session.

```
// Login
logging_channel.send_request(Request(LOGIN, current_user));

// Deposit
Response resp = finance_channel.send_request(
                Request(DEPOSIT, current_user, amount));
if (resp.success) {
   cout << "Deposit successful. New balance: " << resp.balance << endl;
   logging_channel.send_request(Request(DEPOSIT, current_user, amount));
}

// Upload
Response resp = file_channel.send_request(
                Request(UPLOAD_FILE, current_user, 0.0, filename, content));
if (resp.success) {
   cout << "File upload successful\n";
   logging_channel.send_request(
                Request(UPLOAD_FILE, current_user, 0.0, filename));
}
```

Withdraw mirrors deposit, and download mirrors upload with `DOWNLOAD_FILE` and no `data` field. Replace the starter's placeholder `Response resp;` with the assignment — if you leave it, you are testing a default-constructed response whose `success` is `false`, and every action reports failure.

`viewed balance: 0` — the trap that costs the most points here

The logging server prints `r.amount` for a `BALANCE` request, not `r.balance`. Look at `logging.cpp`:

```
43          case BALANCE:
44             logfile << "viewed balance: " << r.amount;
45             break;
```

A `Request` has no `balance` field at all — only a `Response` does. So after the finance server tells you the balance, you must copy that number into the `amount` field of the request you send to the logging server:

```
logging_channel.send_request(Request(BALANCE, current_user, resp.balance));
```

Send a default-constructed request instead and your log reads `[1]: viewed balance: 0` while your screen correctly shows 300. The screen is right, the log is wrong, and the log is what is graded.

Read every response, always

`send_request` writes a request and then blocks reading the reply. If you ever send a request without consuming its response, the reply stays in the pipe and the *next* read returns it — so from then on every answer you get belongs to the previous question. The failure appears one action late, which makes it painful to diagnose.
Calling `send_request` and discarding its return value is fine: the read still happened. What is not fine is a code path that writes to the channel without going through `send_request`.

6.3 The two upload traps

Both of these are limits of the provided channel. Neither is something you can fix in `client.cpp`, and both were confirmed by running them.

A | in the file silently truncates it

The wire format separates fields with `|`, so file content is cut at the first one. A 29-byte file:

```
$ cat pipe.txt
alpha|beta gamma
second line
$ ./client # ... upload pipe.txt ...
File upload successful
$ wc -c storage/pipe.txt
5 storage/pipe.txt
```

Five bytes survived — `alpha` — and the client still reported success. Avoid `|` in anything you upload. `channel.h`, which the reference session uploads, contains none.

An upload over about 1 KB kills the file server

A whole request must fit in a 1024-byte buffer, and that budget covers the type, user id, amount, filename *and* contents. Uploading a 3000-byte file, then attempting any second file operation, gives this:

```
Enter choice: 5
Enter filename to upload: big.txt
File upload successful
Enter choice: 5
Enter filename to upload: small.h
$ echo $?
13
```

The first upload reports success and stores a truncated 1009 bytes. The second produces no output at all: the client **terminates** with exit status 13, mid-menu, and the file server is gone.

The mechanism is that the oversized request overflowed the buffer, leaving bytes behind in the pipe. The file server parsed that leftover as its next request, could not make sense of it, read it as a `QUIT`, and exited. Your client then wrote into a pipe with no reader.
Files of 900, 950 and 990 bytes were all uploaded, stored and followed by further operations with no trouble. Keep test uploads under roughly 1 KB; `channel.h` at 579 bytes is a good choice.

The handout is wrong about what happens next

The handout states that after the file server dies, "every later upload or download does nothing at all, with no error." That is not what happens. The client does not carry on quietly — it exits with status 13 and stops responding to the menu entirely. Reproduced five times out of five.
Practically the advice is the same (restart the client, use a smaller file), but if your client vanishes mid-session after a large upload, it is this and not a crash in your code. Worth mentioning to the instructor.

Uploaded files are one byte larger than the original

`channel.h` is 579 bytes; `storage/channel.h` comes back as 580. Test files of 900, 950 and 990 bytes stored as 901, 951 and 991. The channel's wire format appends a newline. A `diff` shows only "No newline at end of file". This is expected and costs no points — do not go hunting for it in your code.

What is actually happening

Every action in this lab is two requests to two servers, and the split is the point. The finance server owns balances and knows nothing about files. The file server owns `storage/` and knows nothing about money. Neither knows who is logged in, because sessions live only in your client's `current_user` variable.
The logging server is the one that ties them together, and it is deliberately the only one whose state survives the run. Kill the finance server and every balance is gone — the accounts live in that process's heap and die with it. Kill the logging server and the file it appended to is still on disk. That difference is the entire reason the log exists: it is the audit trail of a system whose components are individually disposable.
This also explains the ordering rule. The audit line goes inside `if (resp.success)` because a log that records attempted deposits indistinguishably from real ones is worse than no log — you could no longer reconstruct the balance from it. The log is a record of what happened, and what happened is determined by the server that owns the resource.

Where you will use this for real

- This is microservice architecture in miniature: separate processes, each owning one resource, communicating only by message. Your client is the API gateway.
- Financial and medical systems are legally required to keep exactly this kind of append-only audit log, written after the fact succeeds, and auditors read it rather than the database.
- The response-desynchronisation bug — one unread reply and every subsequent answer is off by one — is the classic failure mode of HTTP keep-alive connections and database

connection pools. Its signature in production is the same: user A sees user B's data.
- Fixed-size buffers silently truncating oversized input is how a large share of real security vulnerabilities begin.
- Separator characters appearing inside data is why CSV parsing is harder than it looks and why SQL injection exists. The `|` trap above is that bug class, in a form you can watch happen.

The intuition

A request is a letter, not a phone call. You put everything the recipient needs into the envelope, because once it is sealed you cannot add to it and they cannot ask you a follow-up. That is why a `Request` carries five fields most of which go unused, and why the balance you learned from the finance server has to be physically copied into the letter you send to the logging server. The logging server has no way to ask the finance server anything — they have never met.
The corollary is the one that catches people. A `Response` is not a `Request`: it has no `user_id`, no `amount`, no `filename`. It does not need them. You are the one who asked, so you already know what the answer is about — and if you have forgotten, that is your bookkeeping problem, not the server's.

Takeaway

Three channels named `"finance"`, `"file"` and `"logging"`, opened `CLIENT_SIDE` after the forks. Every successful action sends a second, auditing request to the logging server from inside the existing `if (resp.success)` branch. The one non-obvious field mapping is `BALANCE`: the logging server prints `r.amount`, so pass `resp.balance` as the request's amount or the log reads `viewed balance: 0`. Login and logout touch only the logging server. Keep uploads under about 1 KB and free of `|` characters — over the limit the client exits with status 13 and the file server dies, and a `|` truncates the content while still reporting `File upload successful`. Every reply must be read, or every later reply is the wrong one.

Checkpoint

Reproduce the reference log exactly. Run the session in `storage/example_execution.txt` — login as 1, deposit 500, withdraw 200, view balance, upload `channel.h`, download `example.log`, logout, exit — then:

```
$ diff system.log storage/example.log
```

The only acceptable difference is `\ No newline at end of file` on the reference side. In particular line 4 must read `[1]: viewed balance: 300`, not `300` on screen and `0` in the file.

7

Task 4 — Closing the channels

10 points — four lines of code, and the failure mode that produces the most confused Discord posts.

7.1 Task 4 — Send `QUIT` down all three channels before returning

Commands used in this task

Call	What it does
`Request(QUIT)`	A request with only its type set. Every server exits on receiving one.
`wait(NULL)`	Blocks until any one child exits; returns that child's PID, or `-1` when there are no children left.

What this task asks you to do

Before `main` returns, tell all three servers to shut down. The starter already ends with `while (wait(NULL) > 0);`, which returns only once every child has exited — and a server exits only when it receives `QUIT`. Miss one and the loop never ends.

How to solve it

At the cleanup `TODO`, after the menu loop:

```
    finance_channel.send_request(Request(QUIT));
    file_channel.send_request(Request(QUIT));
    logging_channel.send_request(Request(QUIT));

    while(wait(NULL) > 0);
```

All three. The order does not matter. `Request(QUIT)` needs no other fields — every other parameter defaults.

`QUIT` is the one request that gets no reply

Each server checks for it before doing anything else and exits immediately:

```
27      if (r.type == QUIT) exit(0);
```

So no response is ever written back. You are calling `send_request`, which normally blocks waiting for a reply — but the server's exit closes the pipe, the read ends, and the call returns. The value it returns is meaningless; ignore it. This is the one place in the lab where not reading a real response is correct.
It is also why `QUIT` never appears in your log file: the logging server exits on line 27, before

reaching any of the code that writes a line.

A missed `QUIT` hangs forever, in complete silence

This does not crash, print a warning, or time out. The client stops. Here is a real run with the logging server's `QUIT` removed, at a 20-second timeout:

```
6. Download File
7. Logout
0. Exit
Enter choice: 0
$ echo $?
124
```

The output simply ends. No prompt returns, no error appears. Exit status 124 is `timeout` killing it, not the program deciding anything.
The grader is blunter about it:

```
=== Channel Closing Tests ===
[FAIL] Channel closing test failed: 0/10 points
  Error: client never exited: it is still waiting on a server that was
  never sent QUIT
```

If your program stops responding at exit, this is why. It is also worth knowing that a missing `QUIT` makes the whole test suite crawl — every test that starts a client now waits out its timeout, and a run that normally takes about a minute took over five.

Why silence rather than an error

Nothing is wrong from the operating system's point of view, and that is exactly the problem. `wait(NULL)` asks the kernel to block until a child changes state. A server that never received `QUIT` is sitting in its own blocking read on a pipe, waiting for a request that will never come. Neither process is spinning, neither is failing — both are asleep, each waiting for the other to act. This is a deadlock, and a deadlocked program consumes no CPU and produces no diagnostic. It is indistinguishable from a program doing slow work, which is why you can lose twenty minutes to it.
The reason it hangs at exit rather than during the session is that the pipes are only closed when a process exits. Right up until the end, the servers are happily blocked on reads and your client is happily driving a menu. The deadlock is created by your last four lines and detected by the line after them.
The consequence worth carrying forward: a hang is a shape of bug, not an absence of one. When a concurrent program stops, ask what each participant is waiting for — there is always an answer, and it is usually a message somebody forgot to send.

Confirming it from another terminal

While the client is hung, look at what is still alive:

```
$ ps -o pid,ppid,stat,comm -C finance,logging,fileserver
```

The server still running is the one you forgot. `STAT` of `S` means interruptible sleep — blocked on that read. Clean up leftovers with `pkill -x logging` and remove stale `fifo_*` files before

rerunning.

Where you will use this for real

- Graceful shutdown is a first-class feature in real systems: Kubernetes sends `SIGTERM` and waits before `SIGKILL`, precisely so processes can finish and exit on their own terms.
- A database connection pool that is not closed leaves server-side connections open until they time out; do it under load and the database refuses new clients.
- Zombie processes are the mirror image of this bug — a child that exited whose parent never called `wait`, so the exit status is never collected and the entry stays in the process table.
- Message queue consumers that never acknowledge a message cause the broker to redeliver it forever. Same shape: one side is waiting for a message the other side never sends.
- "The deploy hangs at the last step and CI times out after an hour" is this bug, in production, costing real money.

The intuition

Think of the wait loop as locking up an office at the end of the day. You cannot leave until everyone has gone home, and the only way anyone knows to go home is if you tell them. Tell two of the three and you will stand at the door all night — not because anything went wrong, but because you are waiting for someone who is patiently waiting for you.
The reason this is worth more than ten points is the asymmetry it teaches. Failures in sequential code announce themselves: a crash, an exception, a wrong number. Failures in concurrent code frequently announce nothing at all, because every individual participant is behaving correctly. Your only evidence is that nothing is happening. Learning to read that absence as data is most of what makes concurrent debugging possible.

Takeaway

Three `Request(QUIT)` sends, one per channel, before `while (wait(NULL) > 0);`. `QUIT` takes no other fields and receives no response, and it is the one request the logging server never writes a line for — it exits first. A missed `QUIT` does not crash or warn: the client simply stops after `Enter choice: 0` and never returns, exit status 124 under `timeout`, and the grader reports `client never exited: it is still waiting on a server that was never sent QUIT`. A hang at exit is always this.

Checkpoint

```
$ printf '1000\nsystem.log\n1\n.txt\n0\n' | ./client >/dev/null; echo $?
$ ps -C finance,logging,fileserver
```

The first command must print `0` and return promptly. The second must list no processes.

8

Verifying your work

8.1 Run the grader yourself

`lab2-tests.sh` is the same set of checks the autograder runs, out of 100. Run it before every push — it is far faster than waiting for the grader.

```
$ make test
```

A complete, correct run:

```
=== Process Information Tests ===
[PASS] Finance process info test passed: 5/5 points
[PASS] Logging process info test passed: 5/5 points
[PASS] File process info test passed: 5/5 points

=== Server Creation Tests ===
[PASS] Fork implementation test passed: 15/15 points
[PASS] Finance server execve test passed: 10/10 points
[PASS] Logging server execve test passed: 10/10 points
[PASS] File server execve test passed: 10/10 points

=== Client-Server Communication Tests ===
[PASS] Financial operations test passed: 10/10 points
[PASS] Logging operations test passed: 10/10 points
[PASS] File operations test passed: 10/10 points

=== Channel Closing Tests ===
[PASS] Channel closing test passed: 10/10 points

========== Final Score ==========
Total Score: 100/100
```

`make test` deletes your executables when it finishes

The last thing the script does is `make -s distclean` (`lab2-tests.sh:306`). So immediately after a test run:

```
$ ./client
bash: ./client: No such file or directory
```

Nothing is broken and your build did not fail. Run `make` again. The handout does not mention this, and it reliably convinces people that a passing test run destroyed their work.

How the grader checks Task 1

It runs the client under `strace` and looks for `fork/clone` calls and for three `execve` calls with the right arguments. This is why the servers must be started *by your client*: starting them yourself in other terminals produces no such syscalls in the client's trace, and every test fails at once.

8.2 Reproduce the reference session by hand

`storage/example_execution.txt` is a full correct run and `storage/example.log` is the log it produced. Reproducing them is a stronger check than the grader, because it compares your output line by line.

```
$ make
$ rm -f system.log example.log storage/channel.h
$ printf '1\nsystem.log\n1\n.h\n1\n1\n2\n500\n3\n200\n4\n5\nchannel.h\n6\nexample.
    log\n7\n0\n' | ./client > mine.txt
$ diff system.log storage/example.log
$ diff mine.txt storage/example_execution.txt
```

The inputs, in order, are: max account 1, log file `system.log`, one extension `.h`; then login as user 1, deposit 500, withdraw 200, view balance, upload `channel.h`, download `example.log`, logout, exit.

Your log should match exactly:

```
[1]: logged in
[1]: deposited 500
[1]: withdrew 200
[1]: viewed balance: 300
[1]: uploaded file: channel.h
[1]: downloaded file: example.log
[1]: logged out
```

Two differences that are expected

The log's last line. `diff` reports `\ No newline at end of file` against the reference. The reference file simply lacks a trailing newline; the content is identical.
Your session transcript is missing the typed numbers. When input is piped rather than typed, the terminal does not echo it, so `Enter choice: 1` appears in the reference as `Enter choice:` in yours. Every line your *program* prints will match. If you want a character-identical transcript, type the session by hand.

8.3 Reading a failure

What the grader says	Where to look
Everything fails at once	The client is not starting the servers. Task 1, before anything else.
Finance process info passes, logging and file fail	The given mutation block is outside `if (pid == 0)`. Chapter 4.
`client never exited: ... never sent QUIT`	A missing `QUIT`. Chapter 7.
Logging operations fail, financial pass	Usually `viewed balance: 0`: the balance was not copied into `amount`. Chapter 6.
File operations fail	Check `./fileserver` versus `./file`, and that the channel is `"file"`.
The suite takes many minutes	Something is hanging and every test is waiting out its timeout. Almost always a missing `QUIT`.

Before you push

```
$ make test
$ git status --short
```

Expect `100/100`, and expect `git status` to show `client.cpp` and nothing else that you built or generated.

9

Deliverables and how points get lost

9.1 What you submit

SUBMIT: `client.cpp`

That is the entire submission. One file, containing all four tasks.

```
$ git add client.cpp
$ git commit -m "lab 2"
$ git push
```

The autograder runs on every push, so push as often as you like and read the result.

Do not commit generated files or executables

The programs' output files — `*_attributes.txt`, your log file, and anything uploaded into `storage/` — are produced when the grader runs your code. So are the four executables and the `build/` directory. All are already covered by `.gitignore`. Check before you push:

```
$ git status --short
```

If anything other than `client.cpp` appears, do not `git add .`.

9.2 The complete list of ways to lose points

Everything in this table was either verified by running it or is quoted from the handout's own list.

Symptom	Cause
Every test fails at once	The servers are not being started by the client. Task 1 first.
Client prints nothing and never exits	A `QUIT` was not sent to one of the three servers.
`viewed balance: 0` in the log	The balance was not copied into the request's `amount` field.
Finance process info passes, logging and file fail	The `if (pid == 0)` block does not wrap the given lines.
`execvp: No such file or directory`, then a hang	The file server is `./fileserver`, not `./file`.
Log file empty	The logging server started, but no requests were sent to it.
Deposits succeed but nothing is logged	The audit request is outside the `if (resp.success)` branch.
Every action reports failure	The starter's placeholder `Response resp;` was left in place, shadowing the real reply.
Works alone, fails in the grader	You renamed `channel.ll`. Do not — the Makefile picks the right one.
Uploads and downloads stop; client exits with status 13	An earlier upload was over about 1 KB and killed the file server.
Uploaded file is truncated at a strange place	It contains a `|`, the wire format's field separator.
`error: expected type` from clang	You are on ARM with clang 18. ARM needs clang 20 or newer.
`./client: No such file or directory` right after a passing test run	`make test` ends with `distclean`. Run `make` again.

9.3 Final checklist

☐ `make` builds all four executables with no warnings.
☐ All three `fork()` calls check for `< 0`.
☐ Each given mutation block sits *inside* `if (pid == 0)`.
☐ Every `args` array ends with `nullptr`, and `args[0]` is the program name.
☐ The file server's `argv` is heap-allocated at `extensions.size() + 2`.
☐ `print_process_info` prints real values, with the labels and line order unchanged.
☐ Three channels, named `"finance"`, `"file"`, `"logging"`, opened `CLIENT_SIDE`.
☐ Every menu action stores its reply in `resp`; no placeholder `Response resp;` remains.
☐ Every audit request is inside `if (resp.success)`.
☐ The `BALANCE` audit passes `resp.balance` as the request's `amount`.
☐ Three `Request(QUIT)` sends before the wait loop.
☐ `make test` reports 100/100.
☐ `system.log` matches `storage/example.log`.

☐ `git status` shows only `client.cpp`.

10

Quick reference

10.1 The request types

Type	Server	Fields it reads	Log line written
`LOGIN`	logging	`user_id`	`[7]: logged in`
`LOGOUT`	logging	`user_id`	`[7]: logged out`
`DEPOSIT`	finance, logging	`user_id`, `amount`	`[7]: deposited 500`
`WITHDRAW`	finance, logging	`user_id`, `amount`	`[7]: withdrew 200`
`BALANCE`	finance, logging	`user_id`; log reads `amount`	`[7]: viewed balance: 300`
`UPLOAD_FILE`	file, logging	`filename`, `data`	`[7]: uploaded file: channel.h`
`DOWNLOAD_FILE`	file, logging	`filename`	`[7]: downloaded file: example.log`
`QUIT`	all three	—	none; the server exits first

10.2 The two structs

```
struct Request {
   RequestType type; // the only required field
   int user_id;
   double amount; // the log reads THIS for BALANCE
   std::string filename;
   std::string data;
   Request(RequestType t, int uid = 0, double amt = 0.0,
         std::string fname = "", std::string d = "");
};

struct Response {
   bool success; // read this first
   double balance; // finance server only
   std::string data; // file contents, on a download
   std::string message; // explanation, useful when success is false
};
```

A `Response` has no `user_id`, no `amount` and no `filename`. If you need to know which user or file a reply refers to, you already do — you asked.

10.3 Server command lines

Server	Command line	Notes
finance	`./finance -m <max>`	Accounts 0 through `max` inclusive. Created on first use with balance 0. Withdrawing exactly the balance succeeds — the test is `>=`.
logging	`./logging -f <file>`	Appends; the file survives between runs.
fileserver	`./fileserver <ext>...`	Owns `storage/`. With **no** extension arguments it performs no check at all and accepts every upload.

10.4 Commands you will use

Command	What it does
`make`	Build all four executables.
`make CLANG=clang-20`	Build using a specific clang (ARM).
`make test`	Run the grader locally. Ends with `distclean`.
`make clean`	Remove executables and `build/`.
`make distclean`	Also remove logs, FIFOs and `*_attributes.txt`.
`uname -m`	Your architecture; decides which IR file is used.
`ps -o pid,ppid,stat,comm -C finance,logging,fileserver`	Which servers are still alive, and their parent.
`pkill -x logging`	Kill a leftover server after a hang.
`rm -f fifo_*`	Remove stale named pipes.
`diff system.log storage/example.log`	Compare against the reference log.

10.5 Symptom to chapter

What you are seeing	Go to
Build fails with `error: expected type`	Chapter 2, ARM/clang
Menu never appears	Chapter 4, failed exec
First server passes, other two fail	Chapter 4, unguarded block
Same address, different values	Chapter 5 — expected
`viewed balance: 0`	Chapter 6, `BALANCE`
Client exits with status 13	Chapter 6, oversized upload
Uploaded file truncated	Chapter 6, `|` separator
Hangs after `Enter choice: 0`	Chapter 7, missing `QUIT`
`./client: No such file or directory`	Chapter 8, `distclean`

10.6 Getting help

Ask on Discord, and bring the exact command you ran and the exact output — not a description of it. Office hours and TA contact details are in the syllabus. The Canvas inbox is not monitored.