Lab 2 — The Process Abstraction and Creating Processes

CSCE 313 · Introduction to Computer Systems · Fall 2026

Due Monday, September 21, 2026, 11:59 PM CT · csce-313-fa26.github.io/lab-2/

What you are building

All labs this term build one project: a financial management system. Lab 2 starts it.

You write the client. All three servers are written for you.

- Run the finance, logging and file servers as child processes
- Print process context, and see what a child does and does not inherit
- Send requests over a RequestChannel and read the responses
- Shut the whole system down cleanly

Everything you write goes in client.cpp, at the TODO comments.

Important concepts

Program vs process

Program: executable code. It sits inert on disk as a bag of bits.

Process: an instance of a program in execution.

- — One program can have many processes running at once
- — Every program runs in the context of some process

Context = process ID, processor state, and address space.

Process context

Two parts of the context matter for this lab.

Process ID, and parent process ID

— Identifies the process, and who created it

Address space, which is private to each process

— Stack — local variables, function parameters

— Heap — whatever you new or malloc

— Data — globals and statics, allocated at start-up

— Code — the instructions themselves

Task 2 has you print these from parent and children and compare them.

The client-server model

A client handles the user. A server owns a resource and answers requests about it.

The client never touches the resource itself — it asks.

Why it is worth the trouble

- The resource lives in one place
- Each server can be understood, changed and crashed on its own

What it costs

- The server is a single point of failure
- Every exchange needs a protocol both sides agree on

Lab 2: an overview

One client, three servers

You run ./client. It starts all three servers itself.

— finance — owns the array of user accounts

— file — owns the storage/ directory

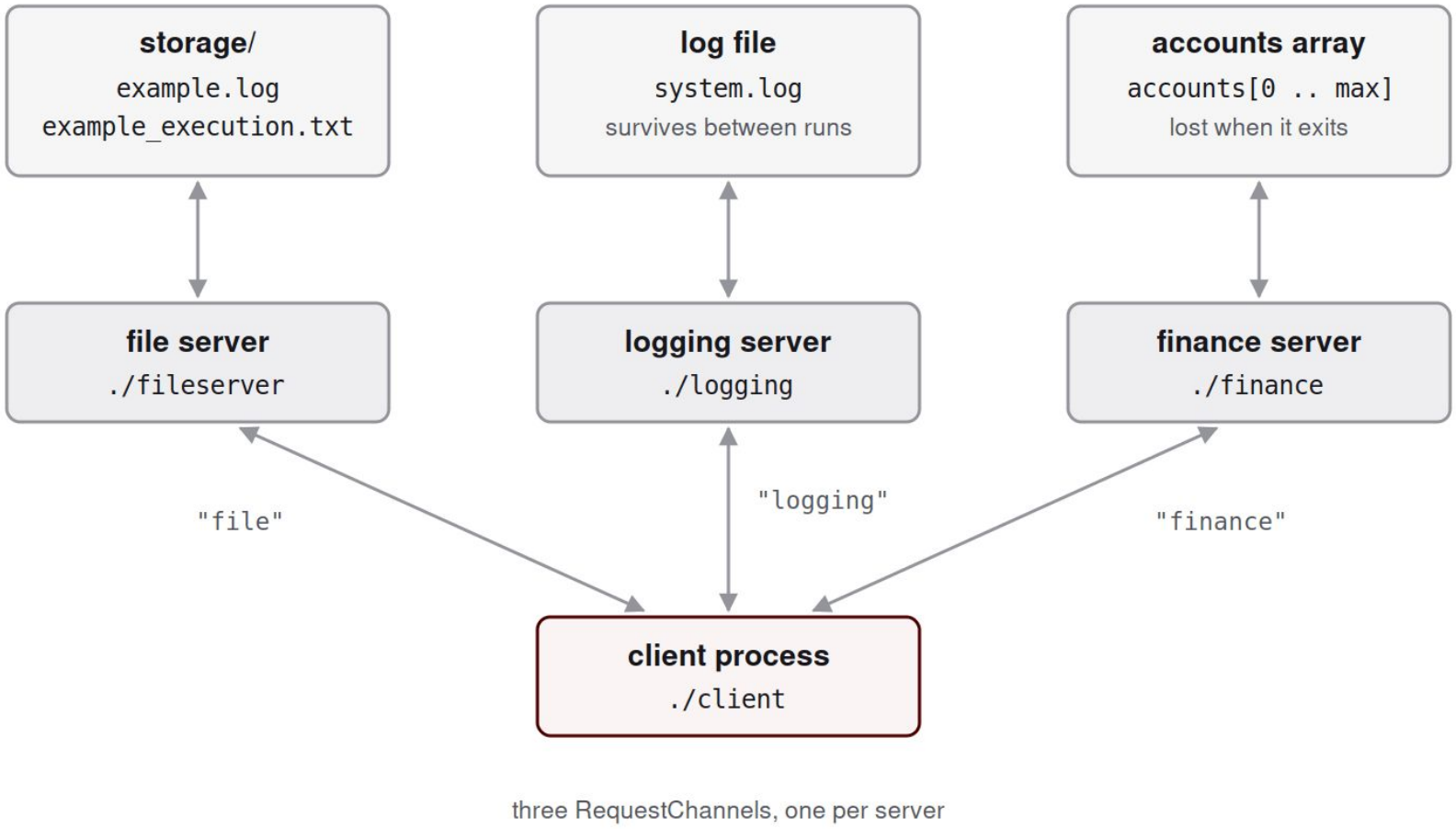
— logging — owns the log file

Each server owns something different, and that difference matters:

— the log file survives the run; the accounts do not, because they live

— in that process's memory and die with it.

Architecture



three RequestChannels, one per server

The client forks and execs all three servers, then talks to each over its own named channel.

Request and Response (common.h)

Request — what you send. Only type is required; the rest default.

```
RequestType type;   int user_id;   double amount;
std::string filename;   std::string data;
```

Response — what comes back. Four fields, and only some are filled.

```
bool success;   double balance;
std::string data;   std::string message;
```

A Response has no user_id, no amount and no filename.

You already know which user and which file you asked about. Read success first, and message when it is false.

The servers

All three are finished

You do not modify any server. You do need to know how to talk to them.

Every server answers QUIT by exiting — and sends nothing back.

Each server reads only the Request fields its own action needs.

Each server fills only the Response fields its own action produces.

Read the server source to see which fields those are. Beyond that you do not need to know how any of them is implemented.

Finance server

`./finance -m <max_account_num>`

Owns an array of accounts, 0 through max_account_num inclusive.

- — Requests from any other id fail with "Invalid account ID"
- — Accounts are created on first use, with a balance of zero

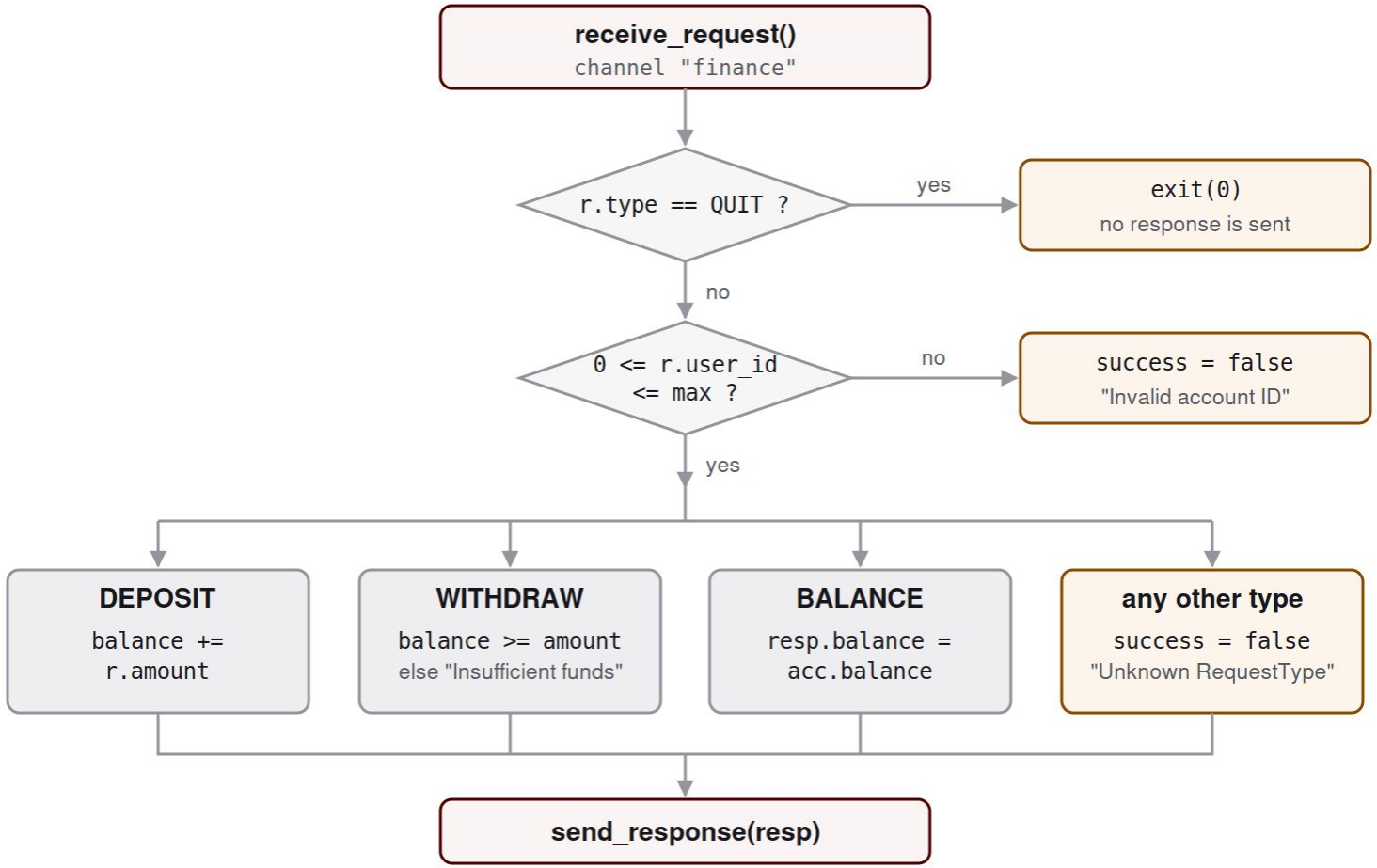
Handles three request types:

- — DEPOSIT — add amount to the balance
- — WITHDRAW — subtract it, if the balance is large enough
- — BALANCE — report the balance

A withdrawal of exactly the balance SUCCEEDS. The test is >=, not >.

resp.balance is set on all three, so print the new balance from the response.

Finance server



QUIT returns nothing at all. Any unrecognised type fails with "Unknown RequestType".

File server

```
./fileserver <ext_1> <ext_2> ... <ext_n>
```

The executable is fileserver, not file — file is already a Unix command.

Its channel, separately, is named "file". Both spellings are right in their own place.

Owns the storage/ directory.

Handles two request types:

— UPLOAD_FILE — needs the filename and the contents

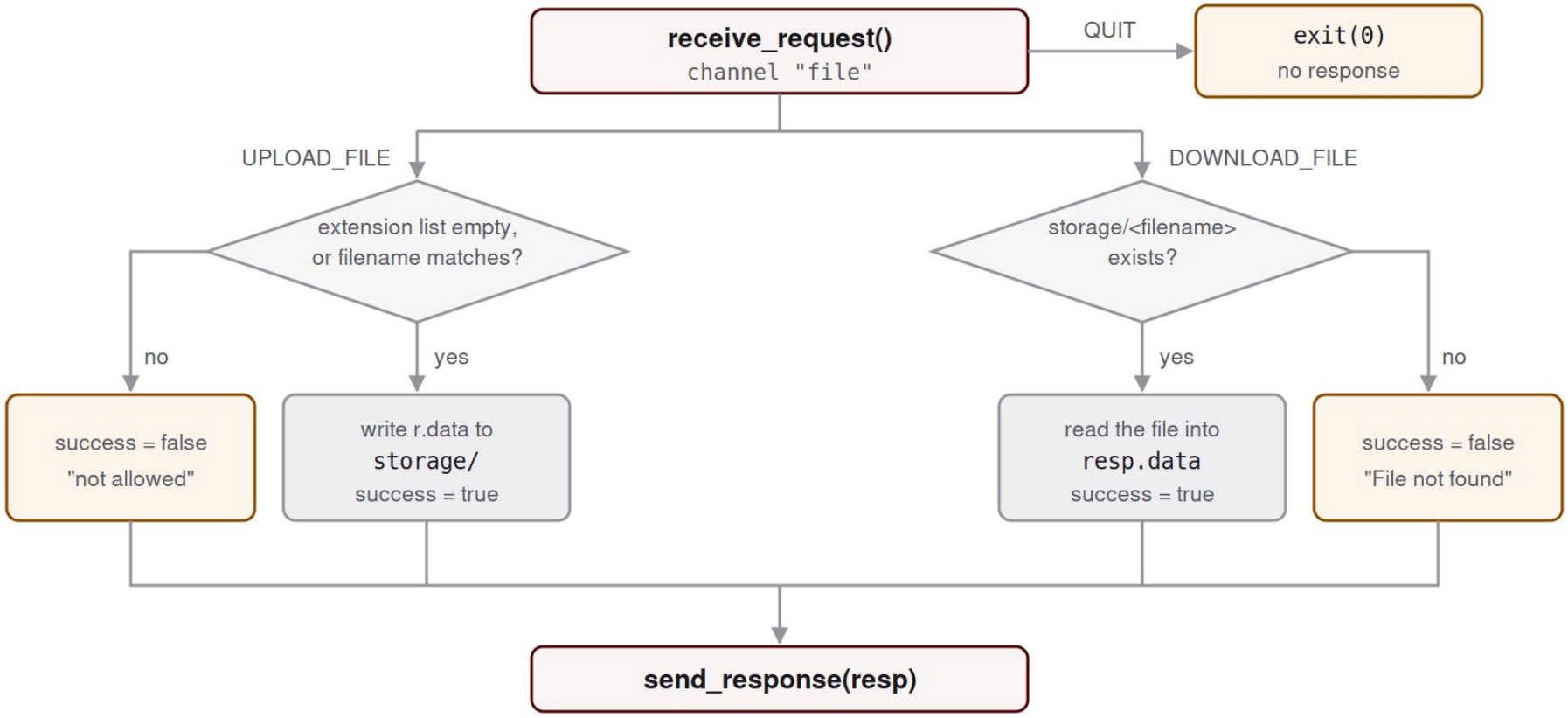
— DOWNLOAD_FILE — needs the filename; returns the bytes in resp.data

Started with NO extension arguments, it allows every upload.

The extension check is skipped when the list is empty — not the other way round.

On a download the server returns bytes. The client writes the file.

File server



The server never writes to your working directory. On a download it returns the bytes, and the client is what saves them to a file.

An upload is checked against the extension list only when that list is non-empty.

Logging server

```
./logging -f <log_filename>
```

Note the hyphen. An en dash will not parse, and the log silently goes to system.log.

Appends one line per request, and the file survives between runs.

Accepts every request type except QUIT, which it does not log —

— it exits before reaching the logging code.

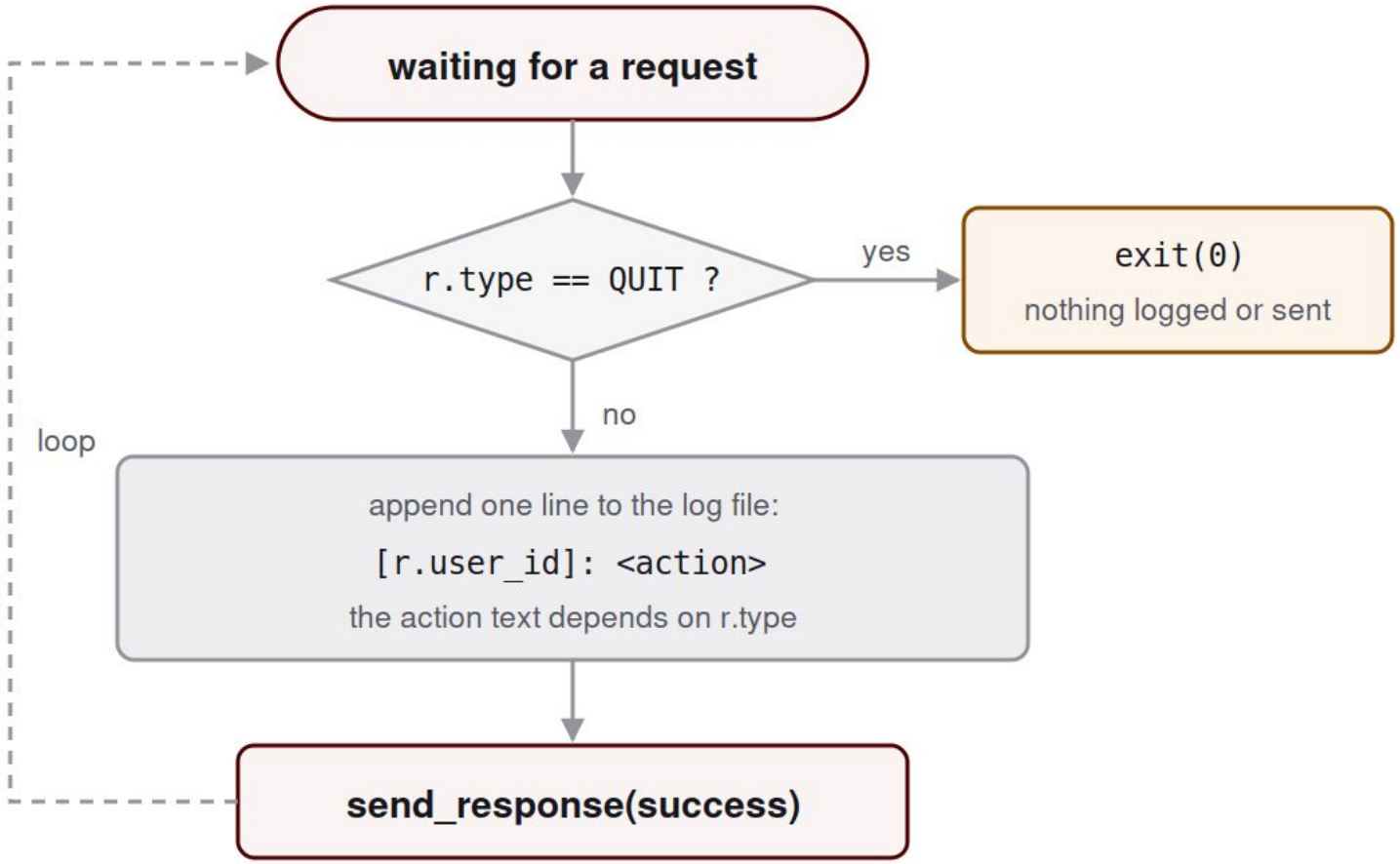
Your log must match storage/example.log exactly.

```
[7]: logged in        [7]: deposited 500      [7]: withdrew 200
[7]: viewed balance: 300      [7]: uploaded file: channel.h
```

For BALANCE the server prints r.amount, not a balance field.

A Request has no balance field. Copy the finance server's answer into amount.

Logging server



QUIT is the one request that is never logged, and never answered.

The RequestChannel

RequestChannel (1)

An inter-process channel. Implemented for you; you never change it.

— The interface is channel.h. The implementation is hidden in channel.ll.

A channel is identified by its NAME. Both sides must use the same string.

```
RequestChannel ch("finance", RequestChannel::CLIENT_SIDE);   // in your client
RequestChannel ch("finance", RequestChannel::SERVER_SIDE);   // in the server
```

The three names the servers use are "finance", "file" and "logging".

Each name must be opened exactly once from each side.

One CLIENT_SIDE and one SERVER_SIDE. Opening the same side twice is undefined behaviour.

RequestChannel (2)

You only ever call one function:

```
Response send_request(const Request& req);
```

It takes a Request and returns a Response.

receive_request() and send_response() are the servers' half of the conversation.

— Already written. You will not call them.

Read every response you ask for.

A reply you never read leaves the channel out of step, and the next reply you read will be the answer to the previous question.

The QUIT message

```
Request quit(QUIT);   finance.send_request(quit);
```

Send one down every channel you opened, before the client returns.

A server exits only when it receives QUIT.

The client's last act is:

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

— which returns only once every child has exited.

Miss one QUIT and the client does not crash. It hangs, forever, silently.

If your program stops responding at exit, this is why.

fork() and exec()

fork() (1)

```
pid_t fork();
```

Creates a new process by duplicating the calling one.

— The new process is the child; the caller is the parent

— The child gets a copy of the parent's memory, not a share of it

Called once, it returns twice — because afterwards there are two processes.

The return value is the only difference, and it is how each process knows which it is.

fork() (2) — one call, two returns

```
int main() {
    cout << "one process prints this" << endl;
    fork();
    cout << "both processes print this" << endl;
}
```

The first line prints once, the second twice.

Which of the two prints first is not defined. Do not rely on the order.

fork() (3) — telling them apart

```
pid_t pid = fork();
if (pid < 0)  { perror("fork"); exit(1); }   // failed
if (pid == 0) { /* the child  */ }
if (pid > 0)  { /* the parent */ }
```

Zero to the child. The child's PID to the parent. Negative on failure.

Guard the child's work with if (pid == 0) or the parent runs it too.

execvp()

```
int execvp(const char* file, char* const argv[]);
```

Replaces the current process image with a different program.

```
char* args[] = {(char*)"./finance", (char*)"-m", (char*)n.c_str(), nullptr};
execvp(args[0], args);
perror("execvp");   exit(1);   // ONLY reached if execvp failed
```

A successful execvp never returns — there is no code left to return to.

The array must end with nullptr, and args[0] is the program's own name.

Do not write to_string(n).c_str() inline: the temporary dies before execvp reads it.

Keep it in a named variable. It often appears to work, which is what makes it dangerous.

Getting started

Setting up

Accept the assignment on classroom50.org, then clone your repository.

```
classroom50.org/CSCE-313-FA26/csce-313-fa26/assignments/lab-2/accept
```

Install clang once — it assembles the channel implementation.

```
sudo apt update && sudo apt install clang
```

```
make
```

— builds four programs: client, finance, fileserver, logging.

Do NOT rename channel.ll. The Makefile picks x86 or ARM from uname -m.

There is nothing to rename and nothing to undo before submitting.

Testing your work

`make test`

— runs lab2-tests.sh — the same checks the autograder runs, out of 100.

Run it before you push. It is far faster than waiting for the grader.

While you are still building the client, you can run the servers by hand,

each in its own terminal:

`./finance -m 1000` `./logging -f system.log` `./fileserver .txt .h`

That is a debugging aid, not the deliverable.

The autograder fails every test if the client does not start the servers itself.

What you submit, and how it is marked

Commit and push client.cpp. That is the whole submission.

Do not commit executables, *_attributes.txt, your log file, or storage/ uploads — .gitignore already excludes them.

```
Task 1 — run the servers as child processes          45 points
Task 2 — print process details                       15 points
Task 3 — client-server communication                 30 points
Task 4 — close the channels                          10 points
```

Only client.cpp is graded. The servers, Makefile and tests are replaced with the reference copies, so editing them changes nothing.

Due Monday, September 21, 2026, 11:59 PM CT.

Questions?

Full handout: csce-313-fa26.github.io/lab-2/

Ask on Discord — bring the exact command and the exact output.