7.7.2025

Reichman University - Operating System
Concluding Assignment
Modular Pipeline System

Tsvi Cherny-Shahar, Daniel Karalnik, Liam Tal

Modular Multithreaded String Analyzer Pipeline in C
Assignment is due to 31.8.2025 at 23:59 – **There will be no late submissions!**

Assignment is **individual**.

You will write in C, build with `gcc13` on Ubuntu

Further submission guidelines are detailed in the Submission Guidelines section.

Do NOT use external libraries, besides the ones listed in this document: `dl` and `pthread`.

Introduction

In this assignment you will experience Systems programming, dynamic linking, multithreading, synchronization, and inter-thread communication by implementing a modular, string-processing pipeline in C/Ubuntu. The project simulates a real-world data processing pipeline where each component (i.e. plugin) performs a specific transformation or action on strings of text.

You will design and build a **multithreaded**, **plugin-based** system that processes input lines from STDIN. The system is designed to be flexible and dynamic: plugins are loaded at runtime as shared objects (`.so` files), and each plugin operates **in its own thread**. Communication between plugins is done through **bounded, thread-safe queues**, following the **producer-consumer** model.

Each plugin in the chain performs a distinct string operation, such as converting text to uppercase, reversing the string, or printing it slowly character-by-character. The pipeline architecture ensures that plugins operate concurrently and independently while maintaining synchronization via queue mechanisms.

A user can specify the processing order and the queue size by passing arguments to the program's command line. When the string `<END>` is received as input, the system shuts down gracefully: queues are drained, and all plugin threads terminate cleanly.

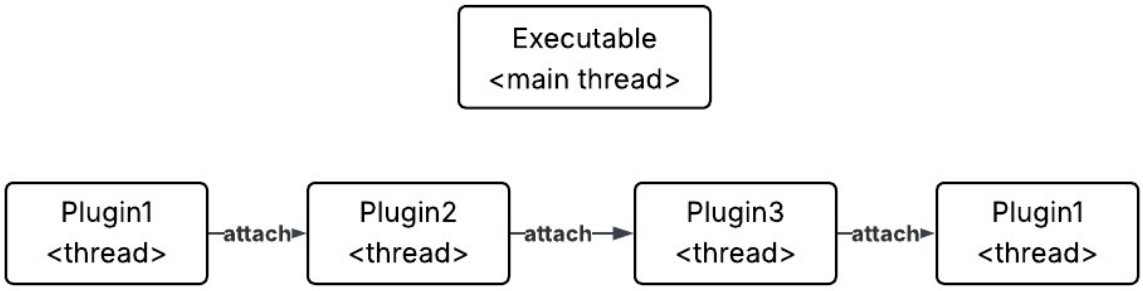
This project combines several core operating systems concepts:

- Multithreading, synchronization and coordiation
- Dynamic loading (`dlopen`, `dlsym`)

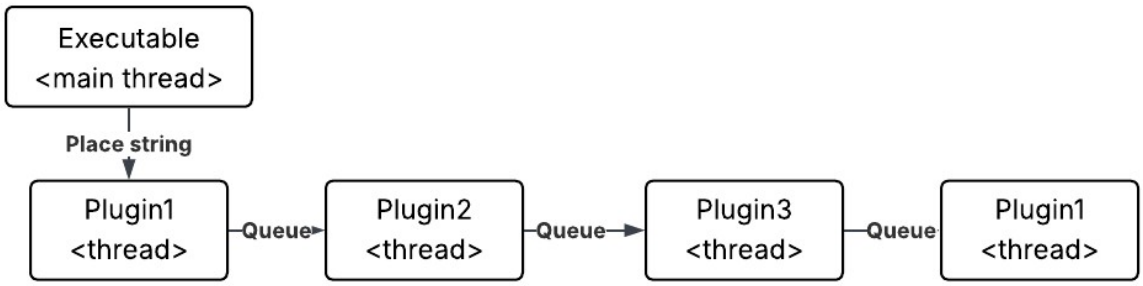
- Shared object Management
- Producer-consumer communication model
- Modular dynamic application

Diagram

Main executable loads the plugins and attaches them into a chain. Each plugin has its own thread, listening to the incoming queue.



Main thread reads from STDIN, places to “job” to first plugin in the chain, which performs its action and move on to the next, until the last plugin. A plugin can be used more than once in the chain.



The following sections outline the development steps for the project:

1. System Overview
2. Plugin Interface and SDK Design
3. Main Application Logic
4. Common Infrastructure Implementation
    1. Synchronization mechanisms
    2. Plugin infrastructure
5. Plugin Development
6. Testing
7. Submission Guidelines

7.7.2025

System Overview

What is a Pipeline?

A pipeline is a software design pattern in which a sequence of processing elements (stages) is chained together. Each stage takes input from the previous one, performs a transformation/action, and passes the result to the next stage. Pipelines are widely used in data processing, compilers, operating systems (e.g., UNIX shell), and multimedia frameworks.

In this project, the pipeline operates on **strings**. Each plugin in the pipeline is a processing stage that performs one transformation or action on the string it receives. The resulting string is then passed to the next plugin via a bounded producer-consumer queue (also referred to as a *channel* in other concurrency models).

System Behavior

- Main application dynamically loads the plugins specified in the command-line arguments and sets their queues to the requested size.
- The main application reads input one line at a time from standard input (STDIN), where a line is defined as a sequence of characters terminated by a newline character (`\n`).
- Each plugin runs in its own thread and communicates with the previous and next stage via a thread-safe bounded producer-consumer queue.
- When a line of input is received, it enters the pipeline and is passed through each plugin, where the last plugin in the chain does not pass to anyone.
- When the input `<END>` is received, it passes through the pipeline, gracefully shuts down every plugin, and finally the main thread shuts down the system gracefully.
- If a stage in the pipeline (i.e., a plugin) attempts to forward work to the next stage while the bounded producer-consumer queue is full, the plugin's thread must wait until space becomes available before enqueuing the string. **Busy waiting is strictly forbidden, use proper synchronization mechanisms to block and resume the thread.**
- If a consumer stage in the pipeline finds its input queue empty, it must wait until new data becomes available. **Busy waiting is strictly prohibited, proper synchronization mechanisms must be used to block and resume the thread efficiently.**

7.7.2025

Required Plugins

The system includes several plugins, each performing a unique transformation or action on the string as it passes through the pipeline:

- **logger**: Logs all strings that pass through to standard output.
- **typewriter**: Simulates a typewriter effect by printing each character with a 100ms delay (you can use the `usleep` function). Notice, this can cause a "traffic jam".
- **uppercaser**: Converts all alphabetic characters in the string to uppercase.
- **rotator**: Moves every character in the string one position to the right. The last character wraps around to the front.
- **flipper**: Reverses the order of characters in the string.
- **expander**: Inserts a single white space between each character in the string.

Example Usage

```
$ ./analyzer 20 uppercaser rotator logger flipper typewriter
```

This command sets all queues capacity to 20 and builds a pipeline in the following order:

1. `uppercaser` – converts input to uppercase
2. `rotator` – rotates the string one character to the right
3. `logger` – prints the string to STDOUT
4. `flipper` – reverses the string
5. `typewriter` – prints the string character-by-character with 100ms delay

Example Input/Output

Input:

```
hello
<END>
```

Output (approximate):

```
[logger] OHELL
[typewriter] LLEHO
```

Note: the same plugin can be used multiple times in the chain.

7.7.2025

Plugin Interface and SDK Design

What is an SDK?

A Software Development Kit (SDK) is a collection of tools, libraries, and specifications that allow developers to write software for a specific system. In this project, the SDK defines the interface between the main application and its plugins. It provides the required function signatures that each plugin must implement and exposes a standard protocol for integration.

What is a Plugin Interface?

In the context of this system, a plugin is a dynamically loaded shared object (.so) that runs in its own thread, performs a specific string action or transformation, and passes its result to the next plugin, until the last plugin in the chain.

A **plugin interface** defines the set of functions that every plugin must export. These functions are used by the main application to initialize the plugin, send it data, connect it to the next plugin, and manage its lifecycle.

All plugins must implement and export the following interface (place in `plugins/plugin_sdk.h` inside your project):

```
/**
 * Get the plugin's name
 * @return The plugin's name (should not be modified or freed)
 */
const char* plugin_get_name(void);

/**
 * Initialize the plugin with the specified queue size
 * @param queue_size Maximum number of items that can be queued
 * @return NULL on success, error message on failure
 */
const char* plugin_init(int queue_size);

/**
 * Finalize the plugin - terminate thread gracefully
 * @return NULL on success, error message on failure
 */
const char* plugin_fini(void);

/**
 * Place work (a string) into the plugin's queue
 * @param str The string to process (plugin takes ownership if it allocates
new memory)
 * @return NULL on success, error message on failure
 */
```

```
const char* plugin_place_work(const char* str);

/**
 * Attach this plugin to the next plugin in the chain
 * @param next_place_work Function pointer to the next plugin's place_work
function
 */
void plugin_attach(const char* (*next_place_work)(const char*));

/**
 * Wait until the plugin has finished processing all work and is ready to
shutdown
 * This is a blocking function used for graceful shutdown coordination
 * @return NULL on success, error message on failure
 */
const char* plugin_wait_finished(void);
```

By enforcing this interface, the system ensures that all plugins can be managed uniformly, supporting dynamic loading and robust pipeline execution.

7.7.2025

Main Application Logic

The main application is implemented in `main.c` and serves as the entry point for constructing and running the pipeline system. It is responsible for parsing input, loading plugins, wiring the pipeline together, managing execution, and shutting everything down cleanly.

Summary of Steps

1. Parse the command-line arguments.
2. Load the plugin shared objects and extract their interfaces.
3. Initialize each plugin.
4. Construct the pipeline by attaching plugins.
5. Read input lines from `stdin` and feed them into the first plugin, until `<END>` is received.
6. Wait for all plugins to finish processing, cleanup and terminate their threads.
7. Clean up and unload all plugins.
8. Exit.

Step 1: Parse Command-Line Arguments

- The first argument is the **queue size**: must be a positive integer.
- The remaining arguments are the names of plugins (without the `.so` extension).
- If the arguments are missing or invalid:
    - Print an error to `stderr`.
    - Print the following usage help to `stdout`:

```
Usage: ./analyzer <queue_size> <plugin1> <plugin2> ... <pluginN>

Arguments:
  queue_size    Maximum number of items in each plugin's queue
  plugin1..N    Names of plugins to load (without .so extension)

Available plugins:
  logger        - Logs all strings that pass through
  typewriter    - Simulates typewriter effect with delays
  uppercaser    - Converts strings to uppercase
  rotator       - Move every character to the right.  Last character moves to
the beginning.
  flipper       - Reverses the order of characters
  expander      - Expands each character with spaces

Example:
  ./analyzer 20 uppercaser rotator logger
```

```
echo 'hello' | ./analyzer 20 uppercaser rotator logger
echo '<END>' | ./analyzer 20 uppercaser rotator logger
```

- Exit with code 1.

Step 2: Load Plugin Shared Objects

- For each plugin name:
  - Construct the filename by appending `.so`.
  - Load it using `dlopen` with flags `RTLD_NOW | RTLD_LOCAL`.
  - Use `dlsym` to resolve the expected exported functions according to the plugin interface.
  - In case of error, you can get the error message using `dlerror` function.
- Store the data in:

```
typedef struct {
    plugin_init_func_t init;
    plugin_fini_func_t fini;
    plugin_place_work_func_t place_work;
    plugin_attach_func_t attach;
    plugin_wait_finished_func_t wait_finished;
    char* name;
    void* handle;
} plugin_handle_t;
```

- On any failure:
  - Print error to `stderr`
  - Print usage to `stdout`
  - Exit with code `1`

Step 3: Initialize Plugins

- Call each plugin's `init(queue_size)` function.
- If a plugin fails to initialize, stop execution, clean up, print error message to `stderr` and exit with error code 2.

Step 4: Attach Plugins Together

- For plugin `i`, call `attach` to `plugin[i+1].place_work`.
- Do not attach the last plugin to anything. Remember, in the plugin, if `attach` is not called, a plugin should know it is the last one.

Step 5: Read Input from STDIN

- Use `fgets()` to read lines up to 1024 characters.
- Make sure there is no trailing `\n` that marks the end of the line. Plugins assume they do not receive it.
- Send the string to the first plugin using its `place_work` function.
- If the string is exactly `<END>`, send it and break the input loop.

Assumptions:

- Input is pure ASCII.
- No input line exceeds 1024 characters (not counting `\n` and terminating null).

Step 6: Wait for Plugins to Finish

- Call each plugin's `wait_finished()` in ascending order (from first to last).
- This ensures all work has been processed and threads have terminated.

Step 7: Cleanup

- Call each plugin's `fini()` to cleanup any memory allocated owned and managed by the plugin.
- Free any memory the main thread allocated.
- Unload each plugin using `dlclose()`.

Step 8: Finalize

- Print:

```
Pipeline shutdown complete
```

- Exit with code `0`.

Build Main Application

Create a script named **`build.sh`** in the root directory that compiles the main application into the ***`output`*** directory.

Note that `dlopen` and `dlsym` are provided by the dynamic linking library `libdl.so`. Since these functions cannot be loaded dynamically themselves, you must explicitly link against the dynamic linking library and use the ELF Loader. To do so, use the `-ldl` flag when compiling with `gcc`.

Ensure that if any `gcc` command fails, the script exits with a non-zero exit code. You can enforce this automatically by adding `set -e` at the top of your Bash script.
You will expand this script later on, to include the plugins as well.

Here are some bash functions that allows you to print with colors, to make your script easier to read:

```
# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m' # No Color

# Function to print colored output
print_status()
{
    echo -e "${GREEN}[BUILD]${NC} $1"
}

print_warning()
{
    echo -e "${YELLOW}[WARNING]${NC} $1"
}

print_error()
{
    echo -e "${RED}[ERROR]${NC} $1"
}
```

Common Infrastructure Implementation

In a modular system where multiple components (plugins) share common behavior, it's a best practice to centralize shared logic in a reusable infrastructure layer. In this project, all plugins are built on top of a shared SDK that implements standardized queueing logic, synchronization mechanisms, and threading support.

This approach ensures the following:

- **Code reuse**: Prevents duplicating queue and thread management logic in every plugin.
- **Maintainability**: Bugs fixed in the shared infrastructure automatically improve all plugins.
- **Consistency**: Ensures that all plugins follow the same concurrency model and shutdown behavior.

- **Simplification**: Developers can focus only on the string transformation logic specific to their plugin.

Each plugin uses this infrastructure to:

- Manage its input queue with bounded capacity.
- Launch and coordinate its own worker thread.
- Receive data through `plugin_place_work` and send data through the attached plugin.
- Shut down gracefully when `<END>` is received.

The following subsections describe the key infrastructure components included in the plugin SDK.

Synchronization Mechanisms

To implement the producer-consumer queues required by each plugin, we rely on standard synchronization mechanisms available in `<pthread.h>`. These tools, such as mutexes, condition variables, and custom synchronization structures, allow us to safely coordinate access between multiple threads.

For example, to protect critical sections so that only one thread can access a shared resource at a time, a mutex (mutual exclusion lock) is used:

```
pthread_mutex_t mutex;
if (pthread_mutex_init(&mutex, NULL) != 0){
    return -1;
}
pthread_mutex_lock(&mutex);
// This is the critical section
pthread_mutex_unlock(&mutex);

// To free/destroy mutex
pthread_mutex_destroy(&mutex);
```

Monitor

As stated before, our producer-consumer queue needs to support `wait_for_not_empty` and `wait_for_not_full` operations. These operations are typically implemented using condition variables. However, condition variables do not maintain an internal state, which can lead to missed signals.

7.7.2025

The Race Condition:

If one thread signals a condition variable before another thread starts waiting on it, the signal is missed, and the waiting thread can block indefinitely. For example:

Thread 2:

```
wait_for_not_empty() {
    pthread_cond_wait(&cond, &mutex);
}
```

Thread 1:

```
pthread_cond_signal(&cond);
```

If Thread 1 signals before Thread 2 waits, the signal is missed.

This race condition can be resolved within the producer-consumer, but it is easier to create a "stateful condition variable" component and use it, rather than implement within the producer-consumer queue. This component is called **Monitor**.

`monitor` is a custom synchronization primitive that wraps a mutex and condition variable and includes a `signaled` state flag. A monitor can "remember" a signal until it is consumed by a waiting thread. `<pthread.h>` does not implement `monitor`, therefore you'll have to implement one yourself.

Implemented `monitor.h` and `monitor.c` in `plugins/sync/`.

Monitor Header (Recommended header, not mandatory):

```
/**
 * Monitor structure that can remember its state
 * This solves the race condition where signals sent before waiting are lost
 */
typedef struct
{
    pthread_mutex_t mutex;      /* Mutex for thread safety */
    pthread_cond_t condition;   /* Condition variable */
    int signaled;               /* Flag to remember if monitor was signaled */
} monitor_t;

/**
 * Initialize a monitor
 * @param monitor Pointer to monitor structure
 * @return 0 on success, -1 on failure
 */
int monitor_init(monitor_t* monitor);

/**
```

```c
 * Destroy a monitor and free its resources
 * @param monitor Pointer to monitor structure
 */
void monitor_destroy(monitor_t* monitor);

/**
 * Signal a monitor (sets the monitor state)
 * @param monitor Pointer to monitor structure
 */
void monitor_signal(monitor_t* monitor);

/**
 * Reset a monitor (clears the monitor state)
 * @param monitor Pointer to monitor structure
 */
void monitor_reset(monitor_t* monitor);

/**
 * Wait for a monitor to be signaled (infinite wait)
 * @param monitor Pointer to monitor structure
 * @return 0 on success, -1 on error
 */
int monitor_wait(monitor_t* monitor);
```

You are **not** required to implement auto-reset functionality, since only manual reset is required. Auto-reset, as the name implies, automatically resets the monitor once a single thread consumes the signal.

Once you implement the monitor, write unit tests application in `monitor_test.c` to verify its correctness. A unit test is a small program that tests a specific module (in this case, the monitor) in isolation to ensure it behaves as expected. Test thoroughly, as it will save you a lot of headaches and frustrations later on in case of a bug.

Consumer-Producer Queue

Implement the bounded thread-safe consumer-producer queue in `plugins/sync/consumer_producer.h` and `consumer_producer.c`. The queue uses a circular buffer, mutex for access control, and condition variables (wrapped in the monitor) for blocking when the queue is full or empty.

When a producer calls `put` to insert an item into the queue and the queue is full, the function must block until space becomes available.

Similarly, when a consumer calls `get` and the queue is empty, the function must block until an item becomes available.

Busy waiting is strictly prohibited. Use proper synchronization mechanisms to block and resume threads.

7.7.2025

Queue Header (Recommended header, not mandatory):

```c
/**
 * Consumer-Producer queue structure for thread-safe producer-consumer pattern
 * Now using monitors for simpler implementation
 */
typedef struct
{
    char** items;           /* Array of string pointers */
    int capacity;           /* Maximum number of items */
    int count;              /* Current number of items */
    int head;               /* Index of first item */
    int tail;               /* Index of next insertion point */
    monitor_t not_full_monitor;     /* Monitor for "not full" state */
    monitor_t not_empty_monitor;    /* Monitor for "not empty" state */
    monitor_t finished_monitor;     /* Monitor for finished signal */
} consumer_producer_t;

/**
 * Initialize a consumer-producer queue
 * @param queue Pointer to queue structure
 * @param capacity Maximum number of items
 * @return NULL on success, error message on failure
 */
const char* consumer_producer_init(consumer_producer_t* queue, int capacity);

/**
 * Destroy a consumer-producer queue and free its resources
 * @param queue Pointer to queue structure
 */
void consumer_producer_destroy(consumer_producer_t* queue);

/**
 * Add an item to the queue (producer).
 * Blocks if queue is full.
 * @param queue Pointer to queue structure
 * @param item String to add (queue takes ownership)
 * @return NULL on success, error message on failure
 */
const char* consumer_producer_put(consumer_producer_t* queue, const char*
item);

/**
 * Remove an item from the queue (consumer) and returns it.
 * Blocks if queue is empty.
 * @param queue Pointer to queue structure
 * @return String item or NULL if queue is empty
 */
char* consumer_producer_get(consumer_producer_t* queue);
```

```c
/**
 * Signal that processing is finished
 * @param queue Pointer to queue structure
 */
void consumer_producer_signal_finished(consumer_producer_t* queue);

/**
 * Wait for processing to be finished
 * @param queue Pointer to queue structure
 * @return 0 on success, -1 on timeout
 */
int consumer_producer_wait_finished(consumer_producer_t* queue);
```

This infrastructure abstracts away all concurrency concerns and enables the plugin logic to operate safely and efficiently in a multithreaded environment.

You may add additional functions as needed for your implementation.

Once you implement the producer-consumer queue, write a unit test application in `consumer_producer_test.c` to verify its correctness. Just like with the monitor tests, test thoroughly! **Catching bugs early will save you significant time and frustration later on**.

Plugin Infrastructure

To simplify plugin development and enforce consistent behavior across all modules, this project provides a shared implementation layer in `plugins/plugin_common.h` and `plugin_common.c`. This infrastructure handles the plugin's lifecycle, threading, queue interaction, and forwarding of results. By using this layer, each plugin only needs to implement its own transformation logic.

The idea is to implement the common portions of the plugin interface once, in a shared component, to eliminate code duplication and reduce boilerplate. This ensures that all plugins follow a consistent structure while allowing developers to focus only on the logic specific to each plugin.

Provided Structures and Responsibilities

The core of the common infrastructure is the `plugin_context_t` structure, which contains:

- The plugin's name (for diagnostics).
- A pointer to its input queue (`consumer_producer_t*`).
- A thread handle for the worker thread.
- A pointer to the next plugin's `place_work()` function.
- A pointer to the plugin-specific string transformation function.
- Internal flags for initialization and shutdown.

7.7.2025

Functions Provided in `plugin_common.h` (Recommended, not mandatory)

```
/**
 * Common SDK structures and functions for plugin implementation
 */

// Plugin context structure
typedef struct
{
    const char* name;                           // Plugin name (for diagnosis)
    consumer_producer_t* queue;                      // Input queue
    pthread_t consumer_thread;                       // Consumer thread
    const char* (*next_place_work)(const char*);   // Next plugin's place_work
function
    const char* (*process_function)(const char*);  // Plugin-specific
processing function
    int initialized;                                 // Initialization flag
    int finished;                                    // Finished processing flag
} plugin_context_t;

/**
 * Generic consumer thread function
 * This function runs in a separate thread and processes items from the queue
 * @param arg Pointer to plugin_context_t
 * @return NULL
 */
void* plugin_consumer_thread(void* arg);

/**
 * Print error message in the format [ERROR][Plugin Name] - message
 * @param context Plugin context
 * @param message Error message
 */
void log_error(plugin_context_t* context, const char* message);

/**
 * Print info message in the format [INFO][Plugin Name] - message
 * @param context Plugin context
 * @param message Info message
 */
void log_info(plugin_context_t* context, const char* message);

/**
 * Get the plugin's name
 * @return The plugin's name (should not be modified or freed)
 */
__attribute__((visibility("default")))
const char* plugin_get_name(void);
```

```
/**
 * Initialize the common plugin infrastructure with the specified queue size
 * @param process_function Plugin-specific processing function
 * @param name Plugin name
 * @param queue_size Maximum number of items that can be queued
 * @return NULL on success, error message on failure
 */
const char* common_plugin_init(const char* (*process_function)(const char*),
const char* name, int queue_size);

/**
 * Initialize the plugin with the specified queue size - calls
common_plugin_init
 * This function should be implemented by each plugin
 * @param queue_size Maximum number of items that can be queued
 * @return NULL on success, error message on failure
 */
__attribute__((visibility("default")))
const char* plugin_init(int queue_size);

/**
 * Finalize the plugin - drain queue and terminate thread gracefully (i.e.
pthread_join)
 * @return NULL on success, error message on failure
 */
__attribute__((visibility("default")))
const char* plugin_fini(void);

/**
 * Place work (a string) into the plugin's queue
 * @param str The string to process (plugin takes ownership if it allocates
new memory)
 * @return NULL on success, error message on failure
 */
__attribute__((visibility("default")))
const char* plugin_place_work(const char* str);

/**
 * Attach this plugin to the next plugin in the chain
 * @param next_place_work Function pointer to the next plugin's place_work
function
 */
__attribute__((visibility("default")))
void plugin_attach(const char* (*next_place_work)(const char*));

/**
 * Wait until the plugin has finished processing all work and is ready to
shutdown
 * This is a blocking function used for graceful shutdown coordination
```

```
 * @return NULL on success, error message on failure
 */
__attribute__((visibility("default")))
const char* plugin_wait_finished(void);
```

Once implemented we can move on to implement the plugins.

How to Use

Each plugin should:

1. Include `plugin_common.h`.
2. Implement its own transformation logic as:

   ```
   const char* plugin_transform(const char* input);
   ```

3. Call `common_plugin_init(plugin_transform, "plugin_name", queue_size);` from its `plugin_init()`.
4. Rely on the default implementations of `place_work`, `attach`, `wait_finished`, and `fini`.

Benefits

Using the common infrastructure:

- Reduces duplicated logic in every plugin.
- Encourages clean separation between logic and mechanics.
- Makes it easier to write, test, and debug plugins.
- Provides robust, production-safe concurrency without requiring deep synchronization expertise from plugin authors.

This shared layer is a crucial part of the SDK and should be used by every plugin you implement.

Plugin Development

All plugins should be implemented in the `plugins` directory and compiled as shared objects (`.so` files). Each plugin operates as a module that receives a string, transforms it, and passes it to the next stage in the pipeline.

Structure of a Plugin `.c` File

Every plugin must:

1. **Include the common plugin SDK header**:

```c
#include "plugin_common.h"
```

2. **Implement the transformation function**:

```c
const char* plugin_transform(const char* input) {
    // Your transformation logic here
}
```

3. **Call the shared initialization logic**:

```c
const char* plugin_init(int queue_size) {
    return common_plugin_init(plugin_transform, "<plugin_name>",
queue_size);
}
```

4. Export the required plugin interface as described in the `plugin_sdk.h` section.

You do NOT need to reimplement threading or queueing logic — all of that is handled in the common infrastructure.

Note: If the plugin's processing logic allocates memory, it **must** release all memory except the string passed to the next plugin. If the plugin is the last in the chain, it must ensure that this final string is also freed, either by the plugin itself or via the shared common infrastructure. **Be diligent to avoid memory leaks!**

Building Plugins

To compile a plugin, you can use the following command in your `build.sh` script:

```
gcc -fPIC -shared -o output/${plugin_name}.so \
    plugins/${plugin_name}.c \
    plugins/plugin_common.c \
    plugins/sync/monitor.c \
    plugins/sync/consumer_producer.c \
    -ldl -lpthread
```

Explanation of Each Flag:

- `-fPIC`: Generates position-independent code. Reduces relocations.
- `-shared`: Tells the compiler to generate a shared object file (`.so`).
- `-o output/${plugin_name}.so`: Specifies the output path and filename for the compiled plugin.
- `plugins/${plugin_name}.c`: The plugin source file.
- `plugins/plugin_common.c`: Shared plugin infrastructure.
- `plugins/sync/monitor.c`: Monitor implementation used for synchronization.
- `plugins/sync/consumer_producer.c`: Thread-safe queue used by all plugins.
- `-ldl`: Links the dynamic linking library, needed for `dlopen/dlsym`.
- `-lpthread`: Links the POSIX thread library for multithreading support.

You must build each plugin **individually** using the command above for every plugin name. You can automate this using a loop in your `build.sh` script to iterate over a list of plugin source files.

Example:

```
for plugin_name in logger uppercaser rotator flipper expander typewriter; do
    print_status "Building plugin: $plugin_name"
    gcc -fPIC -shared -o output/${plugin_name}.so \
        plugins/${plugin_name}.c \
        plugins/plugin_common.c \
        plugins/sync/monitor.c \
        plugins/sync/consumer_producer.c \
        -ldl -lpthread || {
        print_error "Failed to build $plugin_name"
        exit 1
    }
done
```

Once your plugin is compiled, it can be loaded by the main application via command-line arguments. Ensure the filename matches the plugin name provided to `dlopen()` (without the `.so` extension).

Testing

To verify the correctness of your pipeline system, you are required to implement a `test.sh` script in the root directory. This script should build your entire project (both the main application and plugins using `build.sh`), then run a series of tests to validate both correct and incorrect behaviors of your implementation.

What the Script Should Do

1. Call `build.sh` to compile the main program and all plugins.
2. Run several test cases by providing input (via `echo` or a here-document) and checking the expected output.
3. Check for both **positive cases** (correct inputs that should be processed) and **negative cases** (incorrect usage, missing plugins, broken behavior).
4. Print clear success/failure messages for each test.
5. Exit with a non-zero code if any test fails.

7.7.2025

Example: A Single Test Case

Here is a conceptual example of how to check a basic transformation pipeline using bash (e.g., uppercaser → logger):

```
EXPECTED="[logger] HELLO"
ACTUAL=$(echo "hello
<END>" | ./output/analyzer 10 uppercaser logger | grep "\[logger\]")

if [ "$ACTUAL" == "$EXPECTED" ]; then
    print_status "Test uppercaser + logger: PASS"
else
    print_error "Test uppercaser + logger: FAIL (Expected '$EXPECTED', got
'$ACTUAL')"
    exit 1
fi
```

You may choose any approach to automate your tests, but all tests must be executed automatically by running `test.sh`, with no manual intervention.

This will allow you to write and perform, easily, a comprehensive number of tests.

Important Notes:

- Check edge cases: empty strings, long strings, multiple plugin chains.
- Include at least one test that checks how the program handles incorrect arguments (e.g., missing queue size or invalid plugin name).
- Use meaningful output in your script to make debugging easier.

Why This Is Critical

Testing is not just a grading requirement, it helps you:

- Validate that your system works as intended under different scenarios.
- Catch memory leaks, deadlocks, and race conditions early.
- Save time during debugging.
- Gain confidence before submitting your work.

Do not underestimate the importance of testing. A well-tested project is far more robust and maintainable, and most likely get a higher grade!

7.7.2025

Submission Guidelines

Output

IN YOUR SUBMISSION – REMOVE ALL INTERNAL LOGS! STDOUT MUST CONTAIN ONLY THE PIPELINE PRINTOUTS (Unless errors, which are written to STDERR).

Environment

Make sure you test your application on Ubuntu 24.04 using gcc 13. If it works on Mac or Windows, but not on Ubuntu 24.04, it does not count!

Error Handling

You **must** handle errors and incorrect input to your functions or applications!

Your application **must not crash!**

DO NOT FORGET INPUT VALIDATION AND ERROR HANDLING!

README file

You must contain a text-based README file that contains the following in this specific format:

[First Name], [Last Name], [ID] (e.g. “John, Doe, 1234567890”)

Zip File Structure

Submit your entire project as a single `.zip` file using the following naming format:

`[first_name]_[last_name]_[id].zip` (e.g. “john_doe_1234567890.zip”)

Your zip file must have the following structure:

```
[zip root]
├── main.c
├── README
├── build.sh
├── test.sh
├── [any other required source files]
└── plugins/
    ├── plugin_common.c
    ├── plugin_common.h
    ├── plugin_sdk.h
    ├── [your plugin .c files]
    └── sync/
        ├── consumer_producer.c
        ├── consumer_producer.h
        ├── monitor.c
```

```
├── monitor.h
└── [any other sync utilities]
```

Do NOT submit the `output`** directory or any compiled binaries.****

Grading and Enforcement

- All submissions are automatically checked for **plagiarism — DON'T COPY CODE!**
- Usage of LLMs is forbidden – Notice, LLMs tend to generate very similar code, which might get detected as generated code or plagiarism with other students.
- The grader will run `build.sh` followed by automated tests. Make sure your project builds successfully.

Penalties

- **Incorrect zip structure**: -15 points
- **Missing README:** 0 points – the grader cannot tell who you are!
  - Resubmission with README is considered as granted minor code fix
- **`build.sh`** does not compile **automatic 0**, unless appeal is accepted
  - Make sure your code works in the requested environment! (gcc13/Ubuntu24.04)
- **Code change appeal penalty (if granted)**: -15 points for each fix deemed minor
  - Adding/fixing README and such are considered as code fix.

Avoid losing mistakes over submission – Check your submission well!

Check before and after you submit:

- Is my zip structure correct? (no root directory, as the diagram shows)
- Did I include the README file?
- Are all files in the zip? Including the test.sh and everything?
- Does build.sh builds successfully and is my app ready to go? **Let's check one more time, just to be safe!**
- **After I'll upload the zip to Moodle**, I'll download it from the website, unzip it, make sure all files are there, build with the script and run the test script. I want to be Sure!

Submission Platform

Submit your zip file via **Moodle** by the given deadline.

Double-check that your submission builds and runs correctly before and after uploading!

GOOD LUCK!