מבוא

במטלה זו תחוו תכנות מערכות, קישור דינמי, רב-תהליכיות, סינכרון ותקשורת בין-תהליכים על ידי מימוש פייפליין מודולרי לעיבוד מחרוזות ב-C/Ubuntu. הפרויקט מדמה פייפליין עיבוד נתונים מהעולם האמיתי שבו כל רכיב (כלומר plugin) מבצע טרנספורמציה או פעולה ספציפית על מחרוזות טקסט.

תעצבו ותבנו מערכת רב-תהליכית מבוססת תוספים (plugins) שמעבדת שורות קלט מ-STDIN. המערכת מתוכננת להיות גמישה ודינמית: תוספים נטענים בזמן ריצה כאובייקטים משותפים (קבצי .so), וכל תוסף פועל בתהליך נפרד משלו.

תקשורת בין תוספים מתבצעת דרך תורים חסומים ובטוחים לתהליכים, בעקבות מודל יצרן-צרכן.

כל תוסף בשרשרת מבצע פעולת מחרוזת נפרדת, כגון המרת טקסט לאותיות גדולות, היפוך המחרוזת, או הדפסתה לאט תו-אחר-תו. ארכיטקטורת הפייפליין מבטיחה שתוספים פועלים במקביל ובאופן עצמאי תוך שמירה על סינכרון דרך מנגנוני תורים.

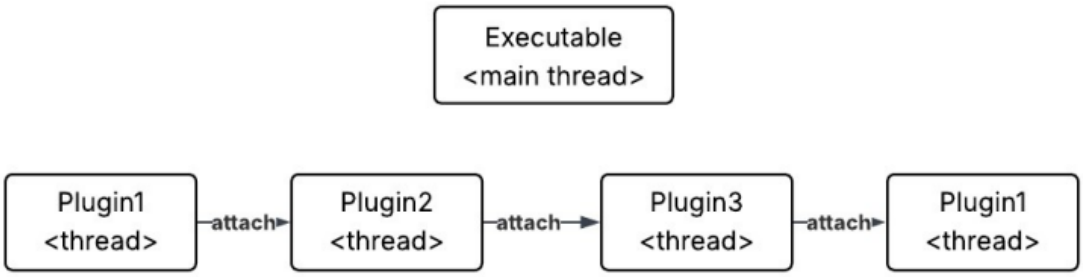
משתמש יכול לציין את סדר העיבוד וגודל התור על ידי העברת ארגומנטים לשורת הפקודה של התוכנית. כאשר המחרוזת <END> מתקבלת כקלט, המערכת נכבית בחן: תורים מרוקנים, וכל תהליכי התוספים מסתיימים נקיים.

פרויקט זה משלב מספר מושגי ליבה של מערכות הפעלה:

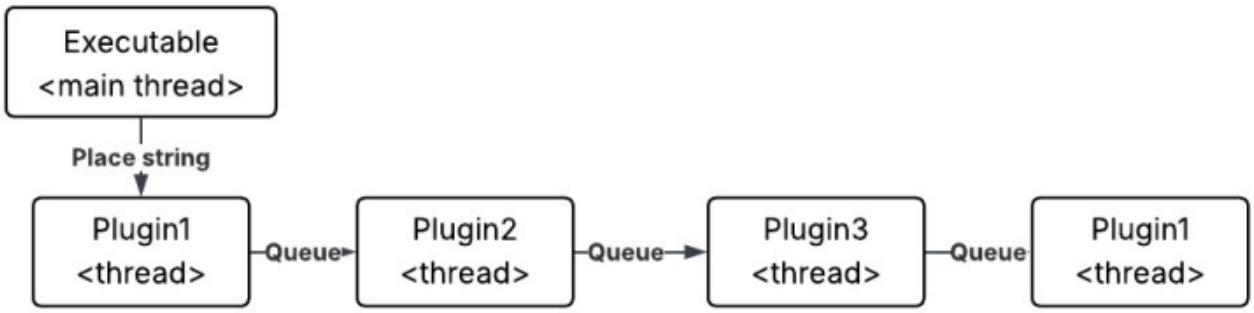
- רב-תהליכיות, סינכרון ותיאום
- טעינה דינמית (dlopen, dlsym)
- ניהול אובייקטים משותפים
- מודל תקשורת יצרן-צרכן
- יישום מודולרי דינמי

דיאגרמה

קובץ הריצה הראשי טוען את התוספים ומחבר אותם לשרשרת. לכל תוסף יש תהליך משלו, שמקשיב לתור הנכנס.



התהליך הראשי קורא מ-STDIN, מכניס "עבודה" לתוסף הראשון בשרשרת, שמבצע את פעולתו ועובר לבא אחריו, עד לתוסף האחרון. תוסף יכול להיות בשימוש יותר מפעם אחת בשרשרת.



הסעיפים הבאים מתארים את שלבי הפיתוח של הפרויקט:

1. סקירת המערכת
2. עיצוב ממשק התוספים ו-SDK
3. לוגיקת היישום הראשי
4. מימוש תשתית משותפת :
   - מנגנוני סינכרון
   - תשתית תוספים
5. פיתוח תוספים
6. בדיקות
7. הנחיות הגשה

סקירת המערכת | מה זה פייפליין?

פייפליין הוא תבנית עיצוב תוכנה שבה רצף של אלמנטי עיבוד (שלבים) מקושרים זה לזה. כל שלב לוקח קלט מהקודם, מבצע טרנספורמציה/פעולה, ומעביר את התוצאה לשלב הבא. פייפליינים נמצאים בשימוש נרחב בעיבוד נתונים, קומפיילרים, מערכות הפעלה (כגון UNIX shell), ומסגרות מולטימדיה.

בפרויקט זה, הפייפליין פועל על מחרוזות. כל תוסף בפייפליין הוא שלב עיבוד שמבצע טרנספורמציה או פעולה אחת על המחרוזת שהוא מקבל. המחרוזת הנוצרת מועברת אז לתוסף הבא דרך תור חסום יצרן-צרכן (מוכר גם כערוץ במודלי מקביליות אחרים).

התנהגות המערכת

היישום הראשי טוען דינמית את התוספים שצוינו בארגומנטים של שורת הפקודה וקובע את גודל התורים שלהם לגודל המבוקש.
היישום הראשי קורא קלט שורה אחת בכל פעם מקלט סטנדרטי (STDIN), כאשר שורה מוגדרת כרצף תווים המסתיים בתו שורה חדשה (\n).

כל תוסף רץ בתהליך משלו ומתקשר עם השלב הקודם והבא דרך תור חסום יצרן-צרכן בטוח לתהליכים.

כאשר שורת קלט מתקבלת, היא נכנסת לפייפליין ומועברת דרך כל תוסף, כאשר התוסף האחרון בשרשרת לא מעביר לאף אחד.

כאשר הקלט <END> מתקבל, הוא עובר דרך הפייפליין, כובה בחן כל תוסף, ולבסוף התהליך הראשי כובה את המערכת בחן.

אם שלב בפייפליין (כלומר, תוסף) מנסה להעביר עבודה לשלב הבא בעוד התור החסום יצרן-צרכן מלא, תהליך התוסף חייב להמתין עד שמקום יתפנה לפני הכנסת המחרוזת לתור. המתנה עסוקה אסורה לחלוטין, השתמשו במנגנוני סינכרון מתאימים לחסימת והמשכת התהליך.

אם שלב צרכן בפייפליין מוצא את תור הקלט שלו ריק, הוא חייב להמתין עד שנתונים חדשים יהיו זמינים. המתנה עסוקה אסורה לחלוטין, יש להשתמש במנגנוני סינכרון מתאימים לחסימת והמשכת התהליך ביעילות.

המערכת כוללת מספר תוספים, כל אחד מבצע טרנספורמציה או פעולה ייחודית על המחרוזת כשהיא עוברת דרך הפייפליין:

- **logger**: רושם את כל המחרוזות שעוברות דרכו לפלט הסטנדרטי.
- **typewriter**: מדמה אפקט מכונת כתיבה על ידי הדפסת כל תו עם עיכוב של 100 ms) ניתן להשתמש בפונקציית usleep). שימו לב, זה עלול לגרום ל"פקק תנועה."
- **uppercaser**: ממיר את כל התווים האלפביתיים במחרוזת לאותיות גדולות.
- **rotator**: מזיז כל תו במחרוזת מיקום אחד ימינה. התו האחרון עובר להתחלה.
- **flipper**: הופך את סדר התווים במחרוזת.
- **expander**: מכניס רווח יחיד בין כל תו במחרוזת.

דוגמת שימוש

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

פקודה זו קובעת את קיבולת כל התורים ל-20 ובונה פייפליין בסדר הבא:

1. uppercaser - ממיר קלט לאותיות גדולות
2. rotator – מסובב את המחרוזת תו אחד ימינה
3. logger – מדפיס את המחרוזת לSTDOUT-
4. flipper – הופך את המחרוזת
5. typewriter – מדפיס את המחרוזת תו-אחר-תו עם עיכוב של ms100

דוגמת קלט/פלט

Input:

```
hello
<END>
```

Output (approximate):

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

הערה: אותו תוסף יכול להיות בשימוש מספר פעמים בשרשרת.

עיצוב ממשק התוספים ו-SDK

מה זה SDK?

ערכת פיתוח תוכנה (SDK) היא אוסף כלים, ספריות ומפרטים המאפשרים למפתחים לכתוב תוכנה עבור מערכת ספציפית. בפרויקט זה, ה-SDK מגדיר את הממשק בין היישום הראשי והתוספים שלו. הוא מספק את חתימות הפונקציות הנדרשות שכל תוסף חייב למימש וחושף פרוטוקול סטנדרטי לאינטגרציה.

מה זה ממשק תוספים?

בהקשר של מערכת זו, תוסף הוא אובייקט משותף (.so) שנטען דינמית, רץ בתהליך משלו, מבצע פעולת מחרוזת או טרנספורמציה ספציפית, ומעביר את התוצאה לתוסף הבא, עד לתוסף האחרון בשרשרת.

ממשק תוספים מגדיר את הסט של פונקציות שכל תוסף חייב לייצא. פונקציות אלה משמשות את היישום הראשי לאתחל את התוסף, לשלוח לו נתונים, לחבר אותו לתוסף הבא, ולנהל את מחזור החיים שלו.

כל התוספים חייבים לממש ולייצא את הממשק הבא (מקם ב-plugins/plugin_sdk.h בתוך הפרויקט שלך):

```
/**
 * 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
 */
```

על ידי אכיפת ממשק זה, המערכת מבטיחה שכל התוספים יכולים להיות מנוהלים באופן אחיד, תומכת בטעינה דינמית ובהרצת פייפליין חזקה.

לוגיקת היישום הראשי

היישום הראשי מיושם ב-main.c ומשמש כנקודת כניסה לבנייה והרצה של מערכת הפייפליין. הוא אחראי על ניתוח קלט, טעינת תוספים, חיווט הפייפליין יחד, ניהול הרצה, וכיבוי הכל נקיים.

סיכום שלבים

1. ניתוח ארגומנטים של שורת הפקודה.
2. טעינת האובייקטים המשותפים של התוספים וחילוץ הממשקים שלהם.
3. אתחול כל תוסף.
4. בניית הפייפליין על ידי חיבור תוספים.
5. קריאת שורות קלט מ-stdin והזנתן לתוסף הראשון, עד שמתקבל <END>.
6. המתנה שכל התוספים יסיימו לעבד, ניקוי וסיום התהליכים שלהם.
7. ניקוי וביטול טעינה של כל התוספים.
8. יציאה.

שלב 1: ניתוח ארגומנטים של שורת הפקודה

- הארגומנט הראשון הוא גודל התור: חייב להיות מספר שלם חיובי.
- הארגומנטים הנותרים הם שמות התוספים (ללא סיומת .so).
- אם הארגומנטים חסרים או לא חוקיים:
  - הדפסת שגיאה ל-stderr.
  - הדפסת העזרה הבאה ל-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
```

יציאה עם EXIT 1

שלב 2: טעינת אובייקטים משותפים של תוספים

- עבור כל שם תוסף:
  - בניית שם הקובץ על ידי הוספת .so.
  - טעינה באמצעות dlopen עם דגלים RTLD_NOW | RTLD_LOCAL.
  - שימוש ב-dlsym לפתרון הפונקציות המצופות המיוצאות לפי ממשק התוספים.
  - במקרה של שגיאה, ניתן לקבל הודעת שגיאה באמצעות פונקציית dlerror.
- שמירת הנתונים ב:

```
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

שלב 3: אתחול תוספים

- קריאה לפונקציית init(queue_size) של כל תוסף.
- אם תוסף נכשל באתחול, עצירת ביצוע, ניקוי, הדפסת הודעת שגיאה ל-stderr ויציאה עם קוד שגיאה 2.

שלב 4: חיבור תוספים יחד

- עבור תוסף i, קריאה ל-attach ל-plugin[i+1].place_work.
- לא לחבר את התוסף האחרון לכלום. זכרו, בתוסף, אם attach לא נקרא, התוסף צריך לדעת שהוא האחרון.

שלב 5: קריאת קלט מ-STDIN

- שימוש ב-fgets() לקריאת שורות עד 1024 תווים.
- ודאו שאין \n נגרר שמסמן את סוף השורה. תוספים מניחים שהם לא מקבלים אותו.
- שליחת המחרוזת לתוסף הראשון באמצעות הפונקציה place_work שלו.
- אם המחרוזת היא בדיוק <END>, שלחו אותה ושברו את לולאת הקלט.

הנחות:

- הקלט הוא ASCII טהור.
- אף שורת קלט לא חורגת מ-1024 תווים (לא כולל \n ו-null המסיים).

שלב 6: המתנה לתוספים לסיים

- קריאה ל-wait_finished() של כל תוסף בסדר עולה (מהראשון לאחרון).
- זה מבטיח שכל העבודה עובדה והתהליכים הסתיימו.

שלב 7: ניקוי

- קריאה ל-fini() של כל תוסף לניקוי כל זיכרון שהוקצה בבעלות ובניהול התוסף.
- שחרור כל זיכרון שהתהליך הראשי הקצה.
- ביטול טעינה של כל תוסף באמצעות dlclose().

שלב 8: סיום

- הדפס :

```
Pipeline shutdown complete
```

- צא עם EXIT 0

בניית היישום הראשי MAIN

צרו סקריפט בשם build.sh בתיקיית השורש שמקמפל את היישום הראשי לתיקיית הפלט.

שימו לב ש-dlopen ו-dlsym מסופקים על ידי ספריית הקישור הדינמי libdl.so. מאחר שהפונקציות האלה לא יכולות להיטען דינמית בעצמן, עליכם לקשר מפורשות נגד ספריית הקישור הדינמי ולהשתמש ב-ELF Loader. כדי לעשות זאת, השתמשו בדגל -ldl כשמקמפלים עם gcc.

ודאו שאם כל פקודת gcc נכשלת, הסקריפט יוצא עם קוד יציאה לא-אפס. אתם יכולים לאכוף זאת אוטומטית על ידי הוספת set -e בראש סקריפט הbash שלכם.

תרחיבו סקריפט זה מאוחר יותר, כדי לכלול גם את התוספים.
הנה כמה פונקציות bash שמאפשרות לכם להדפיס עם צבעים, כדי להקל על קריאת הסקריפט שלכם:

```
# 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"
```

מימוש תשתית משותפת

במערכת מודולרית שבה מספר רכיבים (תוספים) חולקים התנהגות משותפת, זהו תרגול מומלץ לרכז לוגיקה משותפת בשכבת תשתית לשימוש חוזר. בפרויקט זה, כל התוספים בנויים על גבי SDK משותף שמיישם לוגיקת תורים סטנדרטיזית, מנגנוני סינכרון ותמיכה בתהליכיות.

גישה זו מבטיחה את הדברים הבאים:

- שימוש חוזר בקוד: מונע שכפול לוגיקת ניהול תורים ותהליכים בכל תוסף.
- ניתנות לתחזוקה: באגים שמתוקנים בתשתית המשותפת משפרים אוטומטית את כל התוספים.
- עקביות: מבטיח שכל התוספים עוקבים אחר אותו מודל מקביליות והתנהגות כיבוי.
- פישוט: מפתחים יכולים להתמקד רק בלוגיקת טרנספורמציית המחרוזות הספציפית לתוסף שלהם.

כל תוסף משתמש בתשתית זו כדי:

- לנהל את תור הקלט שלו עם קיבולת חסומה.
- להפעיל ולתאם את תהליך העבודה שלו.
- לקבל נתונים דרך plugin_place_work ולשלוח נתונים דרך התוסף המחובר.
- להיכבות בחן כאשר <END> מתקבל.

התת-סעיפים הבאים מתארים את רכיבי התשתית המפתח הכלולים ב-SDK של התוספים.

מנגנוני סינכרון

כדי למימש תורי יצרן-צרכן הנדרשים על ידי כל תוסף, אנחנו נסמכים על מנגנוני סינכרון סטנדרטיים הזמינים ב-<pthread.h>. כלים אלה, כגון mutex-ים, משתני תנאי ומבני סינכרון מותאמים אישית, מאפשרים לנו לתאם גישה בין מספר תהליכים בבטחה.

לדוגמה, כדי להגן על סעיפים קריטיים כך שרק תהליך אחד יוכל לגשת למשאב משותף בכל פעם, נעשה שימוש ב- mutual exclusion lock) mutex) (נעילה הדדית):

```
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

כפי שנאמר קודם, תור יצרן-צרכן שלנו צריך לתמוך בפעולות wait_for_not_empty ו-wait_for_not_full. פעולות אלה מיושמות בדרך כלל באמצעות משתני תנאי. עם זאת, משתני תנאי לא שומרים מצב פנימי, מה שיכול להוביל לאיבוד איתותים.

הgrace condition: אם תהליך אחד מאותת למשתנה תנאי לפני שתהליך אחר מתחיל להמתין עליו, האיתות נאבד, והתהליך הממתין יכול להיחסם לצמיתות. לדוגמה:

Thread 2:

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

Thread 1:

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

אם Thread 1 מאותת לפני ש-Thread 2 ממתין, האיתות נאבד.

מצב מירוץ זה יכול להיפתר באמצעות הproducer-consumer, אבל קל יותר ליצור רכיב "stateful condition variable" ולהשתמש בו, במקום למימש בתוך תור יצרן-צרכן. רכיב זה נקרא Monitor.

monitor הוא פרימיטיב סינכרון מותאם אישית שעוטף mutex ומשתנה תנאי וכולל דגל מצב מאותת. monitor יכול "לזכור" איתות עד שהוא נצרך על ידי thread שמחכה.

<pthread.h> לא מיישם monitor, לכן תצטרכו למימש אחד בעצמכם.

מימשו monitor.h ו-monitor.c ב-plugins/sync/.

אתם לא נדרשים לממש פונקציונליות auto-reset, מאחר שרק manual reset נדרש. Auto-reset, כפי שהשם מרמז, מאפס אוטומטית את ה-monitor ברגע שתהליך יחיד צורך את האיתות.

לאחר שתממשו את ה-monitor, כתבו יישום בדיקות יחידה ב-**monitor_test.c** לאמת את נכונותו. בדיקת יחידה היא תוכנית קטנה שבודקת מודול ספציפי (במקרה זה, ה-monitor) בבידוד כדי להבטיח שהוא מתנהג כצפוי. בדקו ביסודיות, מאחר שזה יחסוך לכם הרבה כאבי ראש ותסכולים מאוחר יותר במקרה של באג.

Consumer-Producer Queue

ממשו את התור החסום והבטוח לתהליכים צרכן-יצרן ב plugins/sync/consumer_producer.h-ו-consumer_producer.c. התור משתמש בבאפר מעגלי, mutex לבקרת גישה ו condition variables(עטופים בMONITOR) לחסימה כאשר התור מלא או ריק.

כאשר יצרן קורא ל-put כדי להכניס פריט לתור והתור מלא, הפונקציה חייבת להיחסם עד שמקום יתפנה.

באופן דומה, כאשר צרכן קורא ל-get והתור ריק, הפונקציה חייבת להיחסם עד שפריט יהיה זמין.

WAIT BUSY אסורה לחלוטין. השתמשו במנגנוני סינכרון מתאימים לחסימה והמשכה של תהליכים.

תשתית זו מפשטת את כל החששות המקבילות ומאפשרת للוגיקת התוספים לפעול בבטחה וביעילות בסביבה רב-תהליכית.

אתם יכולים להוסיף פונקציות נוספות לפי הצורך למימוש שלכם.

לאחר שתממשו את תור יצרן-צרכן, כתבו יישום בדיקת יחידה ב-consumer_producer_test.c לאמת את נכונותו. בדיוק כמו עם בדיקות ה-monitor, בדקו ביסודיות! זיהוי באגים מוקדם יחסוך לכם זמן ותסכול משמעותיים מאוחר יותר.

תשתית תוספים

כדי לפשט פיתוח תוספים ולאכוף התנהגות עקבית על פני כל המודולים, פרויקט זה מספק שכבת מימוש משותפת ב-plugins/plugin_common.h ו-plugin_common.c. תשתית זו מטפלת במחזור החיים של התוסף, תהליכיות, אינטראקציית תור והעברת תוצאות. על ידי שימוש בשכבה זו, כל תוסף צריך רק למימש את לוגיקת הטרנספורמציה שלו.

הרעיון הוא למימש את החלקים המשותפים של ממשק התוספים פעם אחת, ברכיב משותף, כדי לבטל שכפול קוד ולהפחית boilerplate. זה מבטיח שכל התוספים עוקבים אחר מבנה עקבי תוך מתן אפשרות למפתחים להתמקד רק בלוגיקה הספציפית לכל תוסף.

מבנים וחובות מסופקים

ליבת התשתית המשותפת הוא מבנה plugin_context_t, שמכיל:

- שם התוסף (לצורכי אבחון).
- מצביע לתור הקלט שלו (consumer_producer_t*).
- handle תהליך עבור תהליך העבודה.
- מצביע לפונקציית place_work() של התוסף הבא.
- מצביע לפונקציית טרנספורמציית מחרוזות הספציפית לתוסף.
- דגלים פנימיים לאתחול וכיבוי.

לאחר המימוש של plugin_common.h נוכל לעבור למימוש התוספים.

איך להשתמש

כל תוסף צריך:

1. לכלול את plugin_common.h.
2. לממש את לוגיקת הטרנספורמציה שלו כ:

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

3. לקרוא ל-common_plugin_init(plugin_transform, "plugin_name", queue_size); מה-plugin_init() שלו.
4. להסתמך על המימושים הברירת מחדל של place_work, attach, wait_finished, ו-fini.

יתרונות

שימוש בתשתית המשותפת:

- מפחית לוגיקה כפולה בכל תוסף.
- מעודד הפרדה נקיה בין לוגיקה ומכניקה.
- מקל על כתיבה, בדיקה ודיבאג של תוספים.
- מספק מקביליות חזקה ובטוחה לייצור מבלי לדרוש מומחיות עמוקה בסינכרון מכותבי התוספים.

שכבה משותפת זו היא חלק מכריע מה-SDK וצריכה להיות בשימוש על ידי כל תוסף שתמימשו.

פיתוח תוספים

כל התוספים צריכים להיות מיושמים בתיקיית plugins ולהיות מקומפלים כאובייקטים משותפים (קבצי so.). כל תוסף פועל כמודול שמקבל מחרוזת, מטרנספורמציה אותה ומעביר אותה לשלב הבא בפייפליין.

מבנה קובץ .c של תוסף

כל תוסף חייב:

1. לכלול את כותרת SDK התוסף המשותף: `#include "plugin_common.h"`
2. לממש את פונקציית הטרנספורמציה:

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

3. לקרוא ללוגיקת האתחול המשותפת:

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

4. לייצא את ממשק התוספים הנדרש כמתואר בסעיף plugin_sdk.h.

אתם לא צריכים לממש מחדש לוגיקת תהליכיות או תורים — כל זה מטופל בתשתית המשותפת.

הערה: אם לוגיקת העיבוד של התוסף מקצה זיכרון, הוא **חייב** לשחרר את כל הזיכרון חוץ מהמחרוזת שמועברת לתוסף הבא. אם התוסף הוא האחרון בשרשרת, הוא חייב להבטיח שגם המחרוזת הסופית הזו משוחררת, או על ידי התוסף עצמו או דרך התשתית המשותפת המשותפת. **היו זהירים למנוע דליפות זיכרון!**

בניית תוספים

כדי לקמפל תוסף, אתם יכולים להשתמש בפקודה הבאה בסקריפט build.sh שלכם:

```
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
```

הסבר על כל דגל:

- -fPIC: יוצר קוד עצמאי-מיקום. מפחית relocations.
- -shared: אומר לקומפיילר ליצור קובץ אובייקט משותף (.so).
- -o output/${plugin_name}.so: מציין את נתיב הפלט ושם הקובץ עבור התוסף המקומפל.
- plugins/${plugin_name}.c: קובץ המקור של התוסף.
- plugins/plugin_common.c: תשתית תוסף משותפת.
- plugins/sync/monitor.c: מימוש Monitor המשמש לסינכרון.
- plugins/sync/consumer_producer.c: תור בטוח לתהליכים המשמש את כל התוספים.
- -ldl: קושר את ספריית הקישור הדינמי, נדרש ל-dlopen/dlsym.
- -lpthread: קושר את ספריית POSIX thread לתמיכה ברב-תהליכיות.

עליכם לבנות כל תוסף בנפרד באמצעות הפקודה לעיל עבור כל שם תוסף.

אתם יכולים להפוך זאת לאוטומטית באמצעות לולאה בסקריפט build.sh שלכם כדי לעבור על רשימת קבצי מקור תוספים.

דוגמה:

```
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
```

לאחר שהתוסף שלכם מקומפל, הוא יכול להיטען על ידי היישום הMAIN דרך ארגומנטים של שורת פקודה. ודאו ששם הקובץ תואם לשם התוסף שסופק ל) ()dlopen-ללא סיומת.(so.

בדיקות

כדי לאמת את נכונות מערכת הפייפליין שלכם, אתם נדרשים למימש סקריפט test.sh בתיקיית השורש. סקריפט זה צריך לבנות את כל הפרויקט שלכם (גם היישום הראשי וגם התוספים באמצעות build.sh), ואז להריץ סדרת בדיקות לאמת התנהגויות נכונות ושגויות של המימוש שלכם.

מה הסקריפט צריך לעשות

1. לקרוא ל-build.sh כדי לקמפל את התוכנית הראשית ואת כל התוספים.
2. להריץ כמה מקרי בדיקה על ידי מתן קלט (דרך echo או here-document) ובדיקת הפלט הצפוי.
3. לבדוק גם מקרים חיוביים (קלטים נכונים שצריכים להיות מעובדים) וגם מקרים שליליים (שימוש שגוי, תוספים חסרים, התנהגות שבורה).
4. להדפיס הודעות הצלחה/כישלון ברורות עבור כל בדיקה.
5. לצאת עם קוד לא-אפס אם איזושהי בדיקה נכשלת.

דוגמה: מקרה בדיקה יחיד

הנה דוגמה קונספטואלית לאיך לבדוק פייפליין טרנספורמציה בסיסי באמצעות bash (למשל, 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
```

אתם יכולים לבחור כל גישה להפוך את הבדיקות שלכם לאוטומטיות, אבל כל הבדיקות חייבות להתבצע אוטומטית על ידי הרצת test.sh, ללא התערבות ידנית.

זה יאפשר לכם לכתוב ולבצע, בקלות, מספר מקיף של בדיקות.

הערות חשובות:

- בדקו מקרי קצה: מחרוזות ריקות, מחרוזות ארוכות, שרשראות תוספים מרובות.
- כללו לפחות בדיקה אחת שבודקת איך התוכנית מטפלת בארגומנטים שגויים (למשל, גודל תור חסר או שם תוסף לא חוקי).
- השתמשו בפלט משמעותי בסקריפט שלכם כדי להקל על דיבאג.

למה זה קריטי

בדיקות הן לא רק דרישת ציון, הן עוזרות לכם:

- לוודא שהמערכת שלכם עובדת כמתוכנן תחת תרחישים שונים.
- לתפוס דליפות זיכרון, deadlocks ומצבי מירוץ מוקדם.
- לחסוך זמן במהלך דיבאג.
- לזכות בביטחון לפני הגשת העבודה.

אל תזלזלו בחשיבות הבדיקות. פרויקט שנבדק היטב הרבה יותר חזק ובר-תחזוקה, וכנראה יקבל ציון גבוה יותר!

הנחיות הגשה

פלט

בהגשה שלכם – הסירו את כל הלוגים הפנימיים! STDOUT חייב לכלול רק את הדפסות הפייפליין (אלא אם כן שגיאות, שנכתבות ל-STDERR).

סביבה

ודאו שאתם בודקים את היישום שלכם על Ubuntu 24.04 באמצעות gcc 13. אם זה עובד על Mac או Windows, אבל לא על Ubuntu 24.04, זה לא נחשב!

טיפול בשגיאות

עליכם לטפל בשגיאות ובקלט שגוי לפונקציות או יישומים שלכם!
היישום שלכם לא צריך להתרסק!
אל תשכחו אימות קלט וטיפול בשגיאות!

קובץ README

עליכם לכלול קובץ README מבוסס טקסט שמכיל את הבא בפורמט הספציפי הזה:
[שם פרטי], [שם משפחה], [תעודת זהות] (למשל "יוחנן, כהן, 1234567890")

מבנה קובץ Zip

הגישו את כל הפרויקט שלכם כקובץ zip. יחיד באמצעות הפורמט הבא:
[שם_פרטי]*[_שם_משפחה]*[ת"ז].zip (למשל "yochanan_cohen_1234567890.zip")

קובץ ה-zip שלכם חייב להיות בעל המבנה הבא:

```
[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]
```

אל תגישו את תיקיית **output**
או קבצים מקומפלים כלשהם.

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! o Resubmission with README is considered as granted minor code fix

• build.sh does not compile automatic 0, unless appeal is accepted o 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 o 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!

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