Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Курсовой проект по курсу

«Операционные системы»

Группа: М8О-209БВ-24

Студент: Забродин Р.У.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 18.12.25

Москва, 2025

Постановка задачи

Вариант 33**.**

§ **На языке C\C++ написать программу, которая:**

По конфигурационному файлу в формате yaml, json или ini принимает спроектированный DAG джобов и проверяет на корректность: отсутствие циклов, наличие только одной компоненты связанности, наличие стартовых и завершающих джоб. Структура описания джоб и их связей произвольная.

При завершении джобы с ошибкой, необходимо прервать выполнение всего DAG'а и всех запущенных джоб.

Общий метод и алгоритм решения

Использованные системные вызовы:

- fork() - создание нового процесса
- exec*() семейство - замена образа процесса
- waitpid() - ожидание завершения дочернего процесса
- exit() - завершение процесса
- pthread_*() функции - хотя формально это библиотечные вызовы, они в конечном итоге обращаются к системным вызовам:

Сканирую файл построчно, игнорирую комментарии и пустые строки

Определяю секции по ключевым словам jobs: и barriers:

Для каждой задачи:

Имя задачи распознаю по строке с двоеточием (но не command:, deps:, barrier:)

Читаю команду выполнения (command: ...)

Разбираю зависимости (deps: [a, b, c]) — вырезаю скобки, разделяю по запятым

Запоминаю барьер, если указан (barrier: имя_барьера)

Для каждого барьера:

Имя барьера — строка с двоеточием в секции barriers

Список задач барьера — jobs: [x, y, z], считаю количество

Проверка графа — "Три теста перед стартом"

А) Проверка циклов (алгоритм Кана):

Считаю сколько задач зависят от каждой (in[i])
Нахожу стартовые задачи — у которых in[i] = 0
"Удаляю" стартовые задачи, уменьшаю счётчики зависимостей у их последователей
Если в итоге удалил не все задачи — есть цикл

Б) Проверка связности (обход графа):

Начинаю с первой задачи (индекс 0)
Рекурсивно ищу все связанные задачи через зависимости
Если осталась непосещённая задача — граф разбит на несвязанные части

В) Проверка старта и конца:

Старт: задачи без зависимостей (deps_count == 0)
Конец: задачи, от которых никто не зависит (никто не ссылается в своих deps)
Если нет ни одной стартовой или ни одной конечной — ошибка

По сути: Три отдельных проверки, каждая прерывает выполнение при неудаче.

Для каждой задачи создаю поток, который:

А) Ожидает зависимости

Б) Запускаю команду в отдельном процессе:

В) После выполнения:

Если команда упала — ставлю глобальный флаг failed = 1 и будю всех
Если есть барьер — иду на барьер (barrier_wait)
Оповещаю последователей:
У каждой задачи, которая зависит от меня, увеличиваю done_deps++
Делаю pthread_cond_broadcast(&cond) — будю всех ожидающих
Помечаю задачу как выполненную (job->done = 1)

Реализация барьера:

```
реализация барьера
void barrier_wait(Barrier* b) {
    pthread_mutex_lock(&b->mutex);
    b->done++;  // Я завершился
    if (b->done < b->total) {
        // Не все завершили — сплю
        pthread_cond_wait(&b->cond, &b->mutex);
    } else {
        // Я последний — будю всех
        pthread_cond_broadcast(&b->cond);
    }
    pthread_mutex_unlock(&b->mutex);
}
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/wait.h>

#define MAX_JOBS 50
#define MAX_DEPS 10
#define MAX_BARRIERS 10
```

```c
typedef struct {
    char name[32];
    char command[256];
    char deps[MAX_DEPS][32];
    int deps_count;
    int done_deps;
    char barrier[32];
    int done;
    pthread_t thread;
    pid_t pid;
} Job;


typedef struct {
    char name[32];
    int total;
    int done;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} Barrier;


Job jobs[MAX_JOBS];
Barrier barriers[MAX_BARRIERS];
int job_cnt = 0;
int barrier_cnt = 0;
int failed = 0;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;


int find_job(char* name) {
    for (int i = 0; i < job_cnt; i++)
        if (strcmp(jobs[i].name, name) == 0) return i;
    return -1;
}


Barrier* find_barrier(char* name) {
    for (int i = 0; i < barrier_cnt; i++)
        if (strcmp(barriers[i].name, name) == 0) return &barriers[i];
    return NULL;
}


int check_cycles() {
    int in[MAX_JOBS] = {0};

    for (int i = 0; i < job_cnt; i++) {
        for (int d = 0; d < jobs[i].deps_count; d++) {
            int dep = find_job(jobs[i].deps[d]);
            if (dep == -1) {
                printf("Error: job %s depends on unknown %s\n", jobs[i].name, jobs[i].deps[d]);
                return 0;
            }
            in[i]++;
        }
    }
```

```
    int q[MAX_JOBS], front = 0, rear = 0;
    for (int i = 0; i < job_cnt; i++)
        if (in[i] == 0) q[rear++] = i;

    int processed = 0;
    while (front < rear) {
        int u = q[front++];
        processed++;

        for (int i = 0; i < job_cnt; i++) {
            for (int d = 0; d < jobs[i].deps_count; d++) {
                if (strcmp(jobs[i].deps[d], jobs[u].name) == 0) {
                    if (--in[i] == 0) q[rear++] = i;
                }
            }
        }
    }

    if (processed != job_cnt) {
        printf("Error: cycle detected\n");
        return 0;
    }
    return 1;
}



int check_connected() {
    if (job_cnt == 0) return 1;

    int visited[MAX_JOBS] = {0};
    int stack[MAX_JOBS], top = 0;
    stack[top++] = 0;
    visited[0] = 1;

    while (top > 0) {
        int v = stack[--top];

        for (int i = 0; i < job_cnt; i++) {
            for (int d = 0; d < jobs[i].deps_count; d++) {
                if (strcmp(jobs[i].deps[d], jobs[v].name) == 0 && !visited[i]) {
                    visited[i] = 1;
                    stack[top++] = i;
                }
            }
            for (int d = 0; d < jobs[v].deps_count; d++) {
                if (strcmp(jobs[v].deps[d], jobs[i].name) == 0 && !visited[i]) {
                    visited[i] = 1;
                    stack[top++] = i;
                }
            }
        }
    }

    for (int i = 0; i < job_cnt; i++) {
        if (!visited[i]) {
            printf("Error: multiple connected components\n");
            return 0;
        }
    }
    return 1;
}
```

```c
int check_start_end() {
    int start = 0, end = 0;

    for (int i = 0; i < job_cnt; i++)
        if (jobs[i].deps_count == 0) start++;

    for (int i = 0; i < job_cnt; i++) {
        int is_end = 1;
        for (int j = 0; j < job_cnt; j++) {
            for (int d = 0; d < jobs[j].deps_count; d++) {
                if (strcmp(jobs[j].deps[d], jobs[i].name) == 0) {
                    is_end = 0;
                    break;
                }
            }
            if (!is_end) break;
        }
        if (is_end) end++;
    }

    if (start == 0) {
        printf("Error: no start jobs\n");
        return 0;
    }
    if (end == 0) {
        printf("Error: no end jobs\n");
        return 0;
    }
    return 1;
}



void barrier_wait(Barrier* b) {
    pthread_mutex_lock(&b->mutex);
    b->done++;
    if (b->done < b->total) {
        pthread_cond_wait(&b->cond, &b->mutex);
    } else {
        pthread_cond_broadcast(&b->cond);
    }
    pthread_mutex_unlock(&b->mutex);
}


void* run_job(void* arg) {
    Job* job = (Job*)arg;

    pthread_mutex_lock(&lock);
    while (job->done_deps < job->deps_count && !failed)
        pthread_cond_wait(&cond, &lock);
    pthread_mutex_unlock(&lock);

    if (failed) return NULL;

    printf("Start: %s\n", job->name);
```

```
    job->pid = fork();
    if (job->pid == 0) {
        execl("/bin/sh", "sh", "-c", job->command, NULL);
        exit(1);
    } else {
        int status;
        waitpid(job->pid, &status, 0);
        if (WEXITSTATUS(status) != 0) {
            printf("Error: %s\n", job->name);
            failed = 1;
            pthread_cond_broadcast(&cond);
            return NULL;
        }
    }


void parse_simple_yaml(const char* filename) {
    printf("DEBUG: Parsing file: %s\n", filename);

    FILE* f = fopen(filename, "r");
    if (!f) {
        perror("Error opening file");
        exit(1);
    }

    char line[1024];
    int line_num = 0;
    int in_jobs = 0, in_job = 0, in_barriers = 0;
    char current_job[32] = "";
    int job_idx = -1;


    memset(jobs, 0, sizeof(jobs));
    memset(barriers, 0, sizeof(barriers));
    job_cnt = 0;
    barrier_cnt = 0;

    while (fgets(line, sizeof(line), f)) {
        line_num++;

        // Убираем перевод строки
        line[strcspn(line, "\n")] = 0;

        // Пропускаем комментарии и пустые строки
        if (line[0] == '#' || strlen(line) == 0) {
            continue;
        }

        // Убираем ВСЕ ведущие пробелы для анализа
        char* l = line;
        while (*l == ' ' || *l == '\t') l++;

                // Проверяем разделы
                if (strcmp(l, "jobs:") == 0) {
                    in_jobs = 1;
                    in_barriers = 0;
                    in_job = 0;
                    continue;

                } else if (strcmp(l, "barriers:") == 0) {
                            in_jobs = 0;
                            in_barriers = 1;
                            in_job = 0;
                            continue;
                        }
```

```
        if (in_jobs) {
            // Проверяем начало новой задачи (имя заканчивается двоеточием, НЕ "command:", "deps:",
"barrier:")
            if (strchr(l, ':') &&
                strstr(l, "command:") == NULL &&
                strstr(l, "deps:") == NULL &&
                strstr(l, "barrier:") == NULL) {

                // Это новая задача
                char* colon = strchr(l, ':');
                if (colon) {
                    *colon = '\0';

                    // Убираем пробелы из имени
                    char* name = l;
                    while (*name == ' ') name++;

                    printf("DEBUG: Found job: '%s'\n", name);

                    strcpy(current_job, name);
                    job_idx = job_cnt;
                    strcpy(jobs[job_idx].name, name);
                    jobs[job_idx].deps_count = 0;
                    jobs[job_idx].done_deps = 0;
                    jobs[job_idx].done = 0;
                    jobs[job_idx].barrier[0] = '\0';
                    strcpy(jobs[job_idx].command, "echo 'default'");
                    job_cnt++;
                    in_job = 1;
                }
            }


            // Параметры задачи (содержат "command:", "deps:", "barrier:")
            else if (in_job && job_idx >= 0) {
                if (strstr(l, "command:") == l) {
                    char* value = l + 8; // после "command:"
                    while (*value == ' ') value++;
                    strcpy(jobs[job_idx].command, value);
                    printf("DEBUG: Job '%s' command: '%s'\n", current_job, value);
                }

                else if (strstr(l, "barrier:") == l) {
                    char* value = l + 8; // после "barrier:"
                    while (*value == ' ') value++;
                    strcpy(jobs[job_idx].barrier, value);
                    printf("DEBUG: Job '%s' barrier: '%s'\n", current_job, value);
                }

                else if (strstr(l, "deps:") == l) {
                    char* value = l + 5; // после "deps:"
                    while (*value == ' ') value++;

                    // Обработка зависимостей
                    if (strcmp(value, "[]") == 0) {
                        // Пустой список
                        jobs[job_idx].deps_count = 0;
                    } else {
                        // Убираем скобки
                        if (value[0] == '[') value++;
                        char* end = value + strlen(value) - 1;
```

```
    if (*end == ']') *end = '\0';

                    // Разделяем по запятым
                    char* token = strtok(value, ",");
                    while (token && jobs[job_idx].deps_count < MAX_DEPS) {
                        // Убираем пробелы и кавычки
                        while (*token == ' ' || *token == '"' || *token == '\'') token++;
                        char* t = token + strlen(token) - 1;
                        while (t > token && (*t == ' ' || *t == '"' || *t == '\')) *t-- = '\0';

                        if (strlen(token) > 0) {
                            strcpy(jobs[job_idx].deps[jobs[job_idx].deps_count], token);
                            jobs[job_idx].deps_count++;
                            printf("DEBUG: Job '%s' depends on: '%s'\n", current_job, token);
                        }
                        token = strtok(NULL, ",");
                    }
                }
            }
        }

        else if (in_barriers) {
            // Барьер (имя с двоеточием, но не "jobs:")
            if (strchr(l, ':') && strstr(l, "jobs:") == NULL) {
                char* colon = strchr(l, ':');
                if (colon) {
                    *colon = '\0';

                    char* name = l;
                    while (*name == ' ') name++;

                    printf("DEBUG: Found barrier: '%s'\n", name);

                    Barrier* barrier = &barriers[barrier_cnt];
                    strcpy(barrier->name, name);
                    barrier->total = 0;
                    barrier->done = 0;
                    pthread_mutex_init(&barrier->mutex, NULL);
                    pthread_cond_init(&barrier->cond, NULL);
                    barrier_cnt++;
                }
            }
            // Задачи барьера
            else if (barrier_cnt > 0 && strstr(l, "jobs:") == l) {
                char* jobs_list = l + 5; // после "jobs:"
                while (*jobs_list == ' ') jobs_list++;

                printf("DEBUG: Barrier jobs list: '%s'\n", jobs_list);

                // Убираем скобки
                if (jobs_list[0] == '[') jobs_list++;
                char* end = jobs_list + strlen(jobs_list) - 1;
                if (*end == ']') *end = '\0';

                // Разделяем задачи
                char* token = strtok(jobs_list, ",");
                while (token) {
                    while (*token == ' ' || *token == '"' || *token == '\'') token++;
                    char* t = token + strlen(token) - 1;
                    while (t > token && (*t == ' ' || *t == '"' || *t == '\')) *t-- = '\0';

                    if (strlen(token) > 0) {
                        barriers[barrier_cnt-1].total++;
                        printf("DEBUG: Barrier job: '%s'\n", token);
                    }
                    token = strtok(NULL, ",");
                }
            }
        }
```

```
        else if (in_barriers) {
            // Барьер (имя с двоеточием, но не "jobs:")
            if (strchr(l, ':') && strstr(l, "jobs:") == NULL) {
                char* colon = strchr(l, ':');
                if (colon) {
                    *colon = '\0';

                    char* name = l;
                    while (*name == ' ') name++;

                    printf("DEBUG: Found barrier: '%s'\n", name);

                    Barrier* barrier = &barriers[barrier_cnt];
                    strcpy(barrier->name, name);
                    barrier->total = 0;
                    barrier->done = 0;
                    pthread_mutex_init(&barrier->mutex, NULL);
                    pthread_cond_init(&barrier->cond, NULL);
                    barrier_cnt++;
                }
            }
            // Задачи барьера
            else if (barrier_cnt > 0 && strstr(l, "jobs:") == l) {
                char* jobs_list = l + 5; // после "jobs:"
                while (*jobs_list == ' ') jobs_list++;

                printf("DEBUG: Barrier jobs list: '%s'\n", jobs_list);

                // Убираем скобки
                if (jobs_list[0] == '[') jobs_list++;
                char* end = jobs_list + strlen(jobs_list) - 1;
                if (*end == ']') *end = '\0';

                // Разделяем задачи
                char* token = strtok(jobs_list, ",");
                while (token) {
                    while (*token == ' ' || *token == '"' || *token == '\'') token++;
                    char* t = token + strlen(token) - 1;
                    while (t > token && (*t == ' ' || *t == '"' || *t == '\')) *t-- = '\0';

                    if (strlen(token) > 0) {
```

```
                barriers[barrier_cnt-1].total++;
                    printf("DEBUG: Barrier job: '%s'\n", token);
                }
                token = strtok(NULL, ",");
            }
        }
    }
    fclose(f);
    printf("DEBUG: Finished parsing. Found %d jobs, %d barriers\n", job_cnt,
barrier_cnt);
}
// ==================== ГЛАВНАЯ ФУНКЦИЯ
====================

int main(int argc, char** argv) {
    if (argc < 2) {
        printf("Usage: %s <config.yaml>\n", argv[0]);
        return 1;
    }

    // Читаем YAML конфиг
    parse_simple_yaml(argv[1]);

    // Проверяем DAG
    if (!check_cycles()) return 1;
    if (!check_connected()) return 1;
    if (!check_start_end()) return 1;

    printf("\nDAG is valid. Starting execution...\n\n");

    // Запускаем задачи
    for (int i = 0; i < job_cnt; i++)
        pthread_create(&jobs[i].thread, NULL, run_job, &jobs[i]);

    // Ждём завершения
    for (int i = 0; i < job_cnt; i++)
        pthread_join(jobs[i].thread, NULL);

    if (failed) {
        printf("\nDAG failed\n");
        return 1;
    }

    printf("\nAll jobs done successfully\n");
    return 0;
}
```

Протокол работы программы

```
./dag_scheduler config.yaml
DEBUG: Parsing file: config.yaml
DEBUG: Found job: 'A'
DEBUG: Job 'A' command: 'echo 'Job A' && sleep 1'
DEBUG: Job 'A' barrier: """"
DEBUG: Found job: 'B'
DEBUG: Job 'B' command: 'echo 'Job B' && sleep 1'
DEBUG: Job 'B' depends on: 'A'
DEBUG: Job 'B' barrier: '"barrier1"'
DEBUG: Finished parsing. Found 2 jobs, 0 barriers

DAG is valid. Starting execution...

Start: A
Job A
Done: A
Start: B
Job B
Done: B

All jobs done successfully
```

```
 strace -f ./dag_scheduler
execve("./dag_scheduler", ["./dag_scheduler"], 0x7ffd37b89358 /* 36 vars */) = 0
brk(NULL)               = 0x57f86d27a000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x76eff4c1a000
access("/etc/ld.so.preload", R_OK)   = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=19827, ...}) = 0
mmap(NULL, 19827, PROT_READ, MAP_PRIVATE, 3, 0) = 0x76eff4c15000
close(3)                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x76eff4a00000
mmap(0x76eff4a28000, 1605632, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x76eff4a28000
mmap(0x76eff4bb0000, 323584, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x76eff4bb0000
mmap(0x76eff4bff000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x76eff4bff000
mmap(0x76eff4c05000, 52624, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x76eff4c05000
close(3)                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x76eff4c12000
arch_prctl(ARCH_SET_FS, 0x76eff4c12740) = 0
set_tid_address(0x76eff4c12a10)     = 27234
set_robust_list(0x76eff4c12a20, 24)   = 0
rseq(0x76eff4c13060, 0x20, 0, 0x53053053) = 0
mprotect(0x76eff4bff000, 16384, PROT_READ) = 0
mprotect(0x57f8429a4000, 4096, PROT_READ) = 0
mprotect(0x76eff4c52000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x76eff4c15000, 19827)      = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x5), ...}) = 0
getrandom("\x55\xb7\xb2\xb3\x26\xbd\x3a\x07", 8, GRND_NONBLOCK) = 8
brk(NULL)               = 0x57f86d27a000
brk(0x57f86d29b000)           = 0x57f86d29b000
write(1, "Usage: ./dag_scheduler <config.y"..., 37Usage: ./dag_scheduler <config.yaml>
) = 37
exit_group(1)             = ?
+++ exited with 1 +++
```

ВЫВОД

В ходе курсовой работы была разработана система для управления параллельным выполнением задач с учётом зависимостей между ними. Актуальность работы обусловлена необходимостью эффективного использования вычислительных ресурсов в современных многопроцессорных системах, особенно в контексте CI/CD пайплайнов, систем сборки и обработки данных.