成绩：

计算机科学与技术学院

《操作系统》课程大作业

学院： 计算机科学与技术学院

班级： 04012202

任课教师： 吴桂军

学号： 2022211813

姓名： 陈海彬

2024年 12月 15日

**操作系统大作业文档提纲**

1. **设计目标**

模拟网络请求处理流程：通过多线程协作，模拟客户端发送请求、服务器处理请求并分配给后端服务、后端服务处理请求并返回响应的整个网络请求处理流程，帮助理解实际网络应用的工作原理。

提高请求处理效率：利用多线程并行处理请求和任务，充分发挥多核CPU的计算能力，提高系统的请求吞吐量和响应速度，满足高并发场景下的性能需求。

保证线程安全和数据一致性：在多线程环境下，合理使用互斥锁和条件变量等同步机制，确保共享数据（如请求队列、任务队列和响应队列）的访问安全，避免出现数据竞争和不一致的问题，保障系统的稳定性和可靠性。

增强系统的可扩展性：通过模块化的设计，将生产者、请求处理和消费者等不同功能的线程分离，使得系统在面对不同规模和类型的请求时，可以灵活地调整线程数量和任务分配策略，方便后续的扩展和优化。

1. **功能描述**

生产者功能：模拟客户端，负责生成网络请求。每个生产者线程在运行期间会持续生成包含请求ID和请求数据的请求，并将请求放入请求队列中，供请求处理线程处理.

请求处理功能：模拟服务器，负责从请求队列中取出请求，并根据请求类型将请求分配给相应的消费者线程（后端服务）。请求处理线程会管理多个任务队列，将请求放入相应的任务队列中，以便消费者线程进行处理.

消费者功能：模拟后端服务，负责从任务队列中取出请求，处理请求并生成响应数据。每个消费者线程会处理特定类型的任务队列中的请求，处理完成后将响应数据放入响应队列中，供后续处理或展示.

同步和协调功能：通过互斥锁和条件变量等同步机制，协调生产者、请求处理线程和消费者线程之间的协作。确保在请求队列、任务队列和响应队列满或空时，线程能够正确地等待和通知对方，避免出现死锁或资源浪费的情况.

1. **总体设计**

系统架构

生产者模块：包含多个生产者线程，负责生成请求。每个生产者线程独立运行，生成的请求通过请求队列传递给请求处理模块.

请求处理模块：包含一个请求处理线程，负责从请求队列中取出请求，并根据请求类型将请求分配给相应的消费者线程。请求处理线程管理多个任务队列，每个任务队列对应一个消费者线程.

消费者模块：包含多个消费者线程，负责处理任务队列中的请求。每个消费者线程独立运行，处理完成后将响应数据放入响应队列中.

同步机制模块：提供互斥锁和条件变量等同步机制，用于协调生产者、请求处理线程和消费者线程之间的协作，确保共享数据的访问安全和线程间的正确同步.

数据流

请求生成：生产者线程生成请求，包含请求ID和请求数据.

请求传递：生成的请求被放入请求队列中，等待请求处理线程处理.

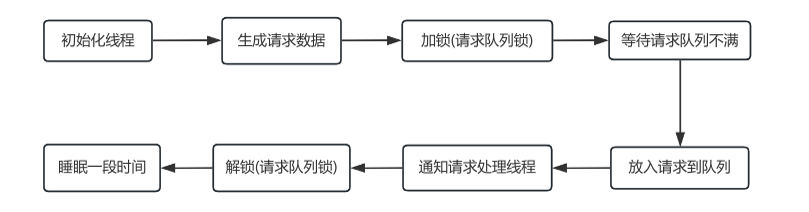
请求处理和分配：请求处理线程从请求队列中取出请求，根据请求类型将请求分配给相应的消费者线程，请求被放入任务队列中.

任务处理：消费者线程从任务队列中取出请求，处理请求并生成响应数据，响应数据被放入响应队列中.

响应处理：后续可以对响应队列中的数据进行进一步处理或展示.

1. **详细设计（流程图或算法步骤）**
   1. 生产者线程设计

流程图：



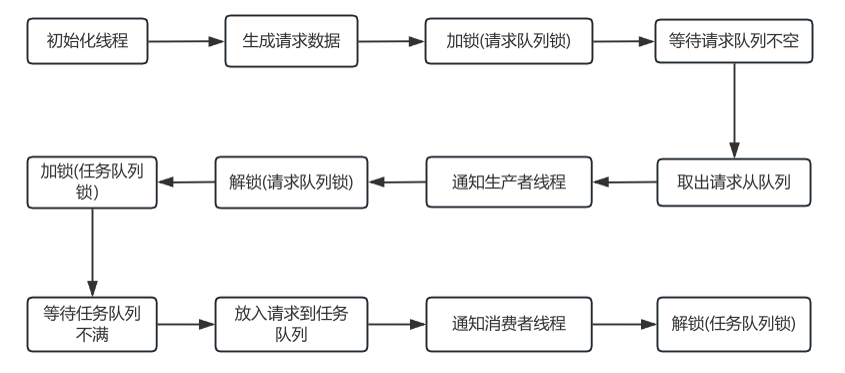
算法步骤：

1. 初始化生产者线程，设置线程参数.
2. 生成请求数据，包括请求ID和请求内容.
3. 加锁请求队列的互斥锁，确保对请求队列的独占访问.
4. 等待请求队列不满，如果请求队列已满，则调用条件变量等待.
5. 将生成的请求放入请求队列中，更新队列状态.
6. 通知请求处理线程请求队列不为空，唤醒等待的请求处理线程.
7. 解锁请求队列的互斥锁，释放对请求队列的访问.
8. 睡眠一段时间，模拟请求生成的间隔.
9. 重复步骤2-8，持续生成请求.



* 1. 请求处理线程设计

流程图：



算法步骤：

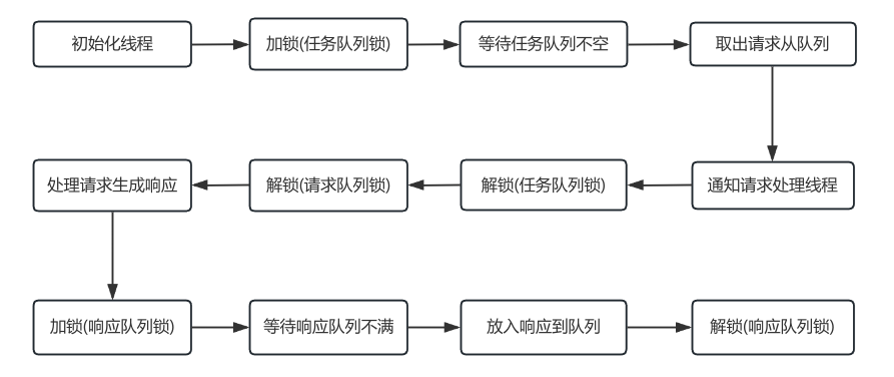
1. 初始化请求处理线程，设置线程参数.
2. 加锁请求队列的互斥锁，确保对请求队列的独占访问.
3. 等待请求队列不为空，如果请求队列为空，则调用条件变量等待.
4. 从请求队列中取出一个请求，更新队列状态.
5. 通知生产者线程请求队列不满，唤醒等待的生产者线程.
6. 解锁请求队列的互斥锁，释放对请求队列的访问.
7. 根据请求类型确定对应的消费者线程和任务队列.
8. 加锁任务队列的互斥锁，确保对任务队列的独占访问.
9. 等待任务队列不满，如果任务队列已满，则调用条件变量等待.
10. 将请求放入任务队列中，更新队列状态.
11. 通知消费者线程任务队列不为空，唤醒等待的消费者线程.
12. 解锁任务队列的互斥锁，释放对任务队列的访问.
13. 重复步骤2-12，持续处理请求.





* 1. 消费者线程设计

流程图：



算法步骤：

1. 初始化消费者线程，设置线程参数.

2. 加锁任务队列的互斥锁，确保对任务队列的独占访问.

3. 等待任务队列不为空，如果任务队列为空，则调用条件变量等待.

4. 从任务队列中取出一个请求，更新队列状态.

5. 通知请求处理线程任务队列不满，唤醒等待的请求处理线程.

6. 解锁任务队列的互斥锁，释放对任务队列的访问.

7. 处理请求，生成响应数据.

8. 加锁响应队列的互斥锁，确保对响应队列的独占访问.

9. 等待响应队列不满，如果响应队列已满，则调用条件变量等待.

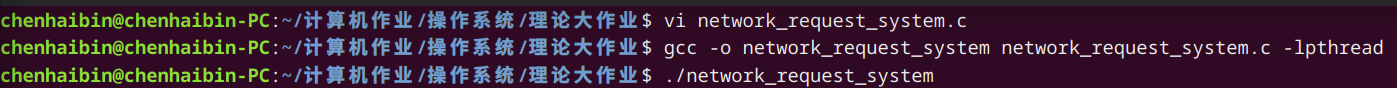
10. 将响应数据放入响应队列中，更新队列状态.

11. 解锁响应队列的互斥锁，释放对响应队列的访问.

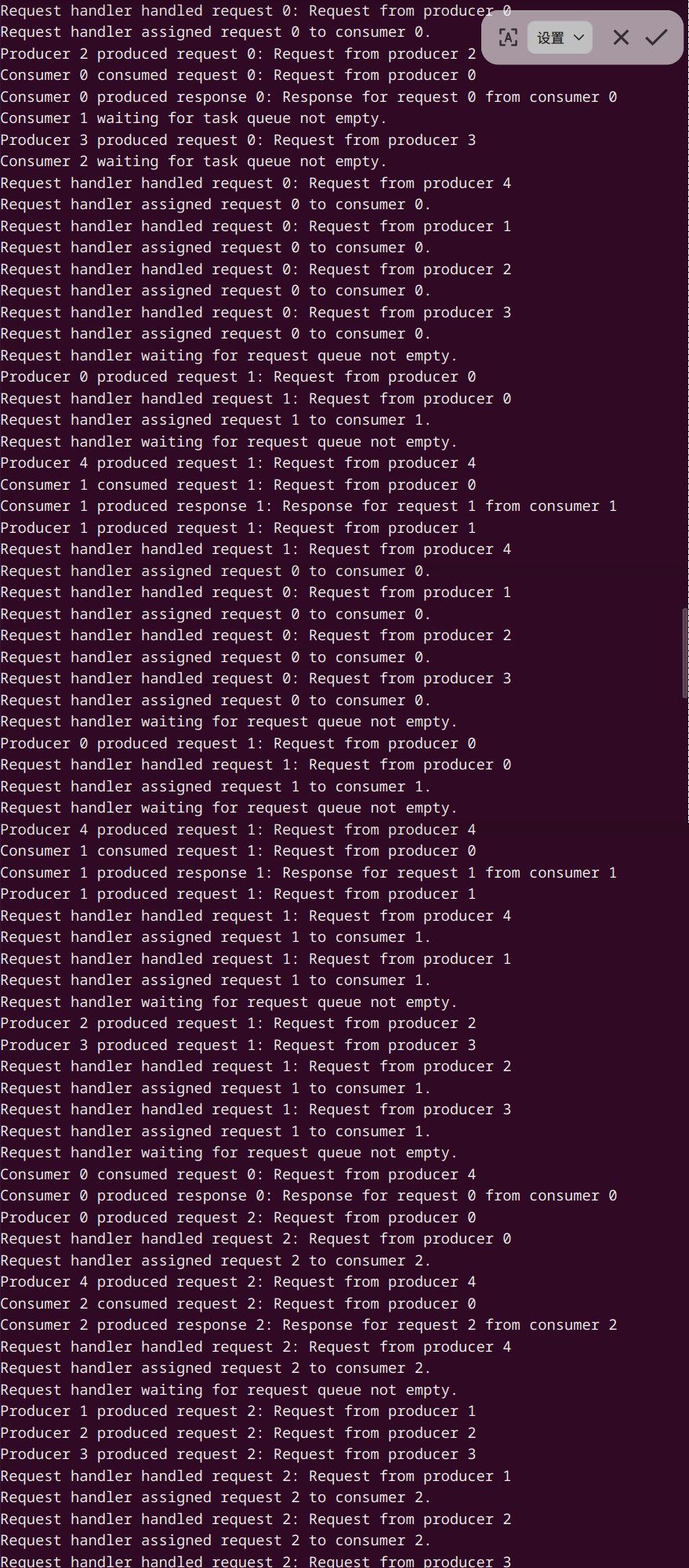
12. 重复步骤2-11，持续处理任务.

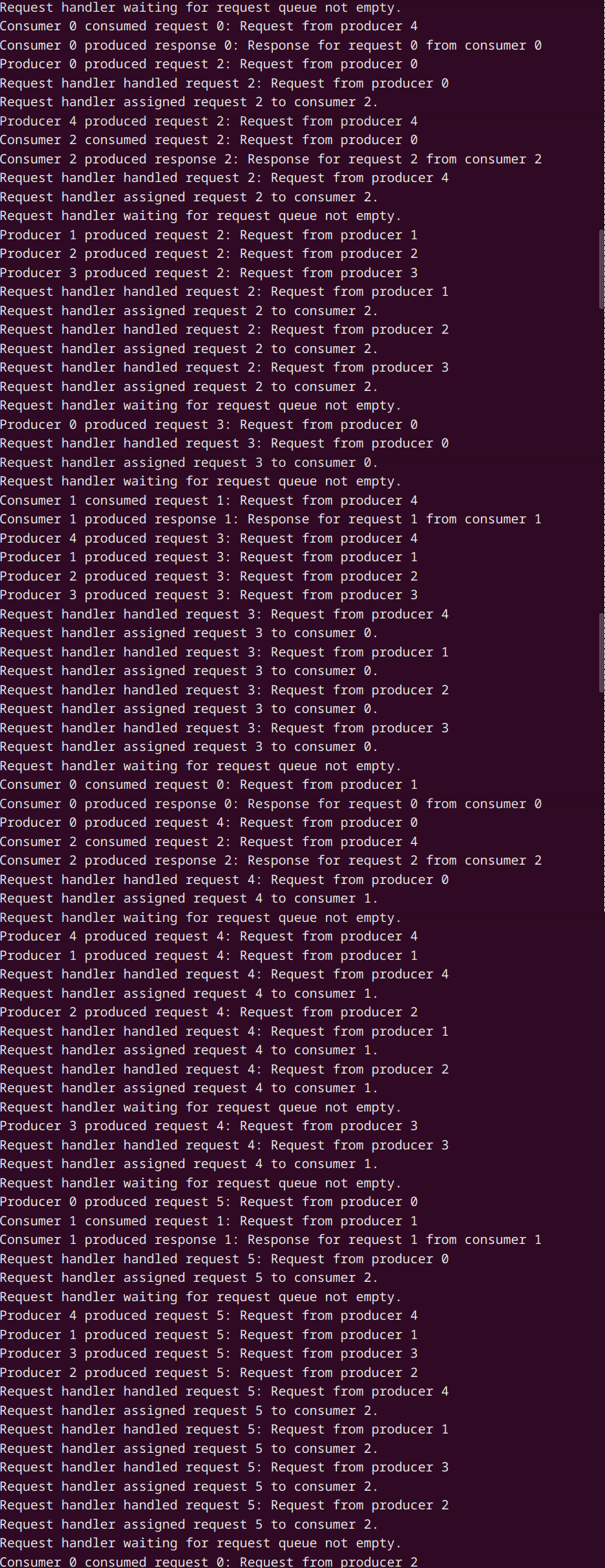


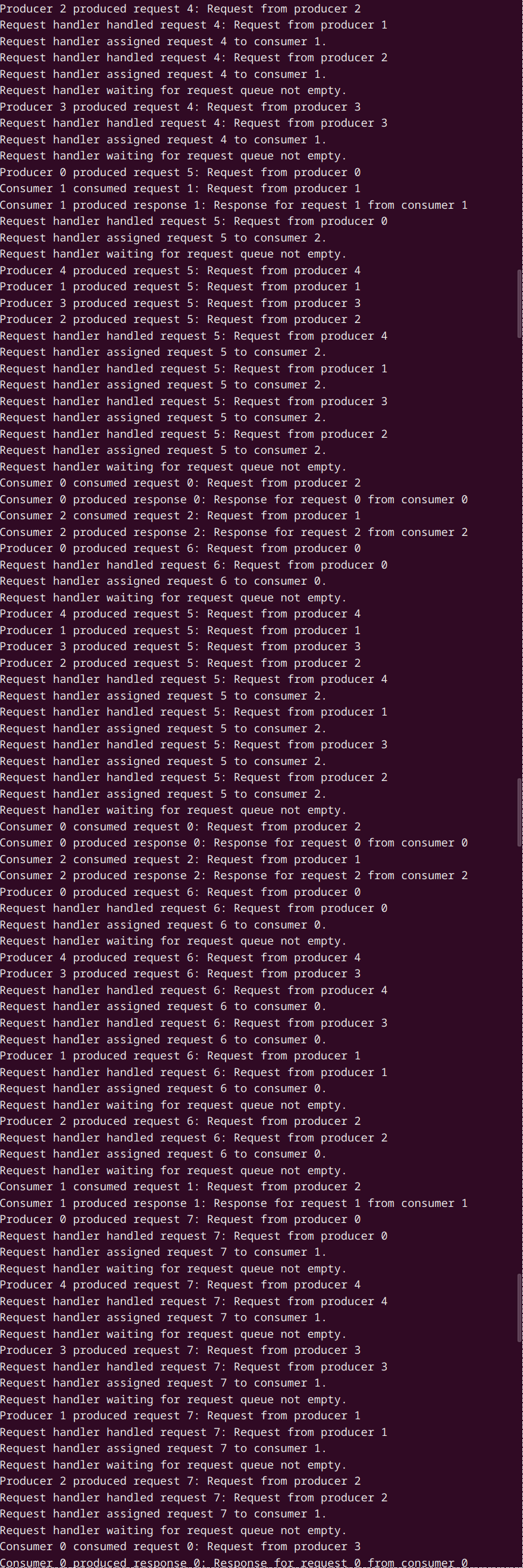
1. **实现（给出编译截图、运行截图）**

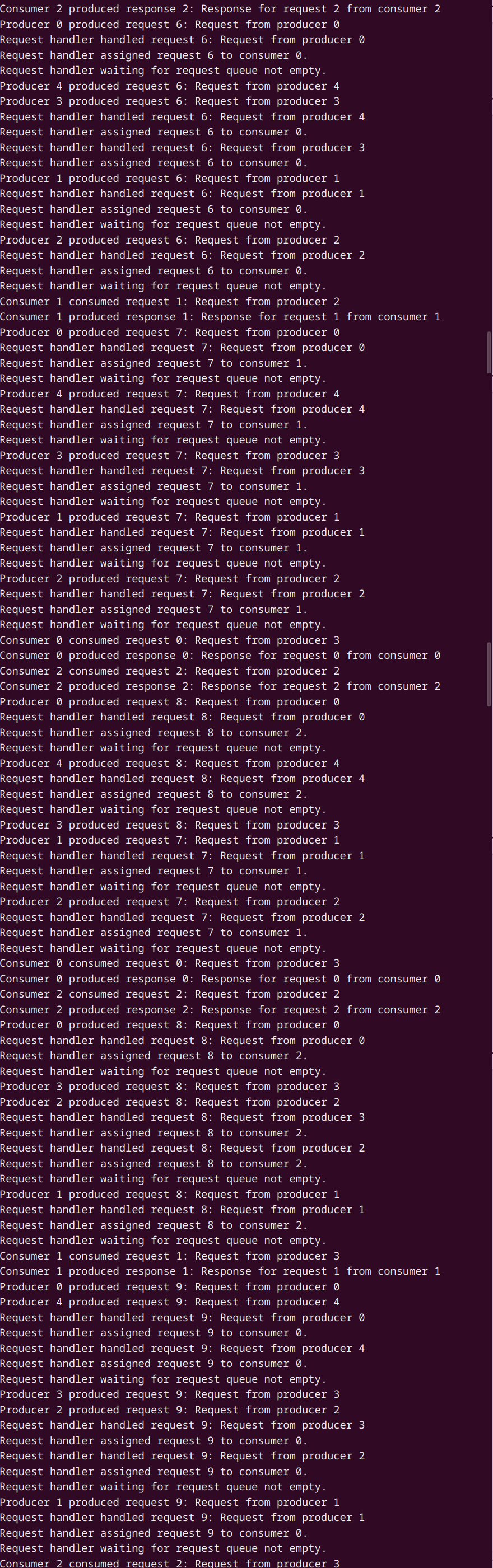


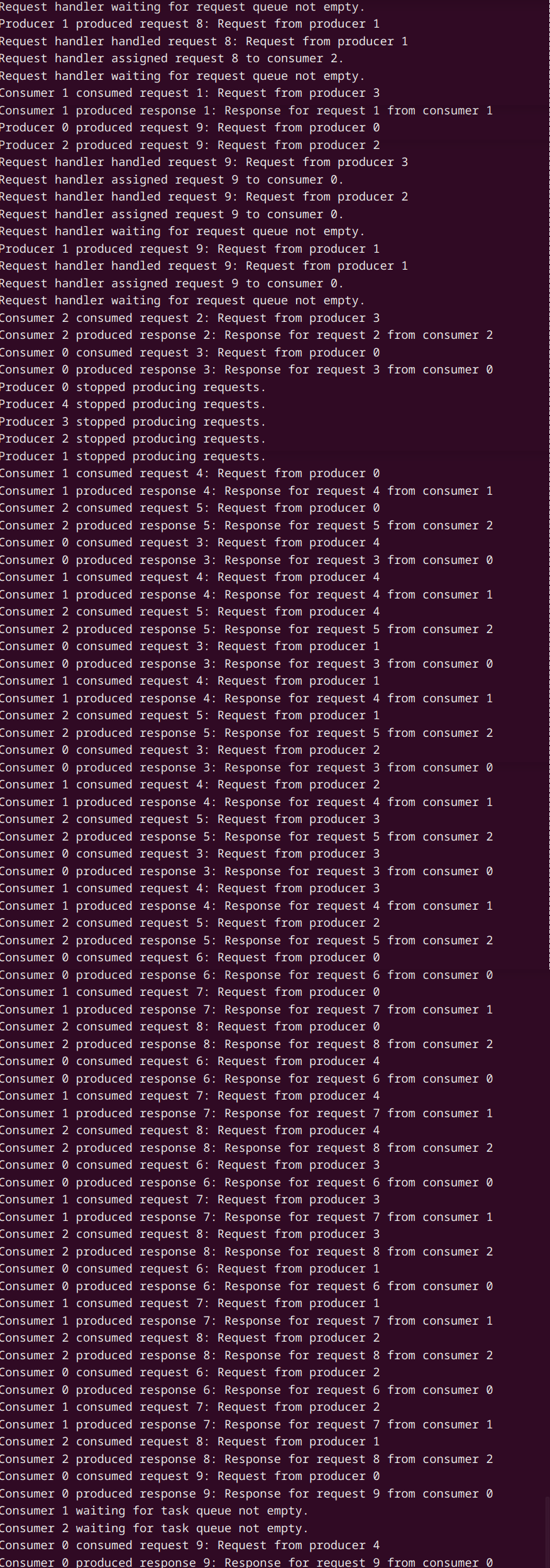


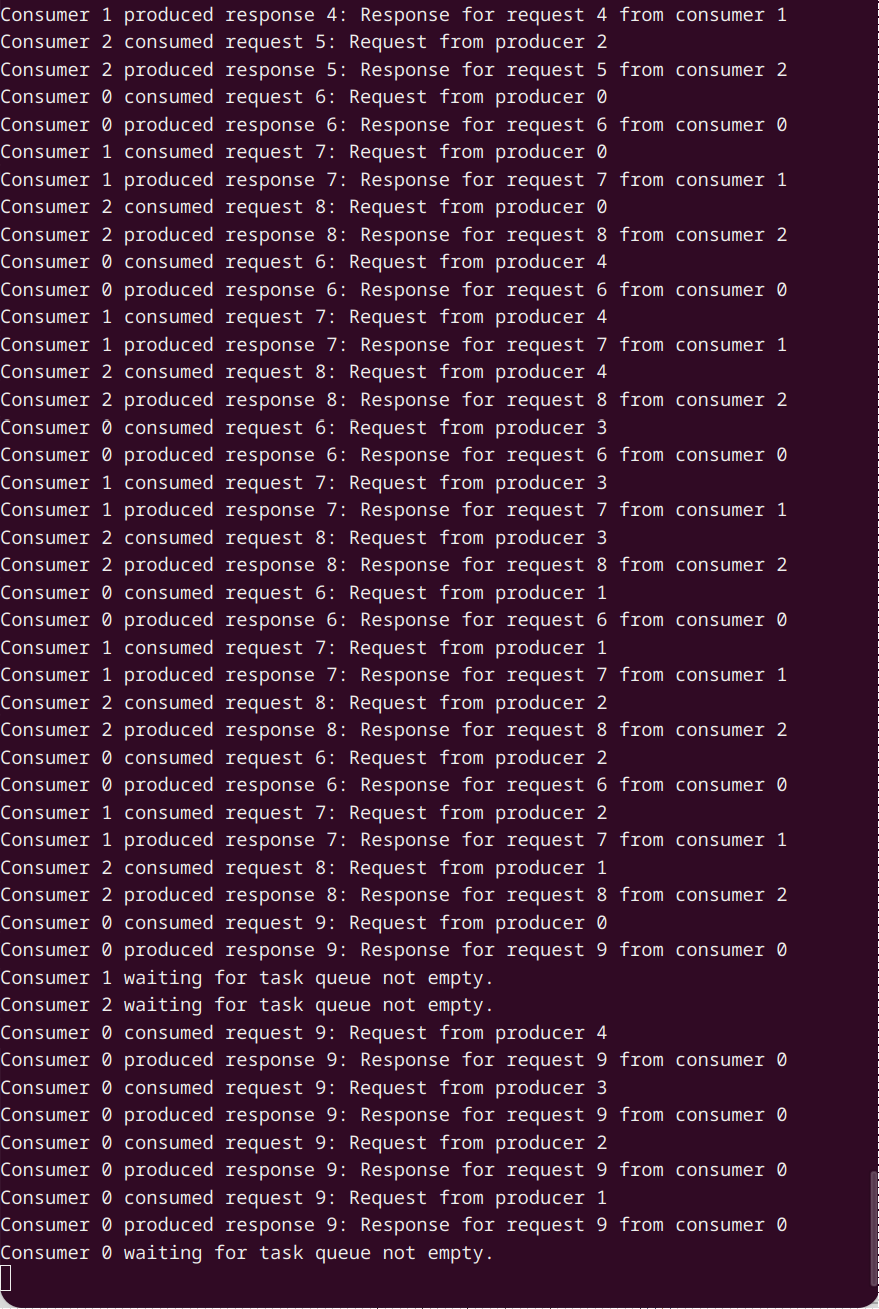












1. **参考资料（3-5个，文献或链接）**

《Linux多线程编程》：介绍了Linux环境下多线程编程的基本概念、互斥锁、条件变量等同步机制的使用方法，为本系统的设计提供了理论基础和编程指导.

《操作系统原理》：详细阐述了操作系统中进程和线程的概念、调度算法、同步和通信机制等，帮助理解多线程协作的原理和实现方式.

《C语言程序设计》：提供了C语言的基本语法和编程技巧，是编写多线程程序的基础.

附件：源代码

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdbool.h>

#define REQUEST\_QUEUE\_SIZE 100

#define TASK\_QUEUE\_SIZE 50

#define RESPONSE\_QUEUE\_SIZE 100

#define NUM\_PRODUCERS 5

#define NUM\_CONSUMERS 3

#define PRODUCER\_LIFETIME 10 // 生产者线程运行时间（秒）

// 请求结构体

typedef struct {

int id;

char data[100];

} Request;

// 响应结构体

typedef struct {

int id;

char data[100];

} Response;

// 请求队列

Request request\_queue[REQUEST\_QUEUE\_SIZE];

int request\_count = 0;

int request\_in = 0;

int request\_out = 0;

bool request\_produced = false; // 标记是否所有生产者线程都停止生成请求

// 任务队列

Request task\_queues[NUM\_CONSUMERS][TASK\_QUEUE\_SIZE];

int task\_counts[NUM\_CONSUMERS] = {0};

int task\_ins[NUM\_CONSUMERS] = {0};

int task\_outs[NUM\_CONSUMERS] = {0};

// 响应队列

Response response\_queue[RESPONSE\_QUEUE\_SIZE];

int response\_count = 0;

int response\_in = 0;

int response\_out = 0;

// 互斥锁和条件变量

pthread\_mutex\_t request\_mutex;

pthread\_cond\_t request\_not\_empty;

pthread\_cond\_t request\_not\_full;

pthread\_mutex\_t task\_mutexes[NUM\_CONSUMERS];

pthread\_cond\_t task\_not\_empty[NUM\_CONSUMERS];

pthread\_cond\_t task\_not\_full[NUM\_CONSUMERS];

pthread\_mutex\_t response\_mutex;

pthread\_cond\_t response\_not\_empty;

pthread\_cond\_t response\_not\_full;

// 生产者线程函数

void\* producer(void\* arg) {

int producer\_id = \*((int\*)arg);

int request\_id = 0;

time\_t start\_time = time(NULL);

while (true) {

if (time(NULL) - start\_time >= PRODUCER\_LIFETIME) {

printf("Producer %d stopped producing requests.\n", producer\_id);

break; // 生产者线程运行时间达到上限，停止生成请求

}

Request request;

request.id = request\_id++;

snprintf(request.data, sizeof(request.data), "Request from producer %d", producer\_id);

// 加锁

pthread\_mutex\_lock(&request\_mutex);

// 等待请求队列不满

while (request\_count == REQUEST\_QUEUE\_SIZE) {

printf("Producer %d waiting for request queue not full.\n", producer\_id);

pthread\_cond\_wait(&request\_not\_full, &request\_mutex);

}

// 将请求放入请求队列

request\_queue[request\_in] = request;

request\_in = (request\_in + 1) % REQUEST\_QUEUE\_SIZE;

request\_count++;

printf("Producer %d produced request %d: %s\n", producer\_id, request.id, request.data);

// 通知请求处理线程请求队列不为空

pthread\_cond\_signal(&request\_not\_empty);

// 解锁

pthread\_mutex\_unlock(&request\_mutex);

// 睡眠一定时间

sleep(1);

}

// 标记生产者线程停止生成请求

pthread\_mutex\_lock(&request\_mutex);

request\_produced = true;

pthread\_mutex\_unlock(&request\_mutex);

return NULL;

}

// 请求处理线程函数

void\* request\_handler(void\* arg) {

while (true) {

Request request;

// 加锁

pthread\_mutex\_lock(&request\_mutex);

// 等待请求队列不为空，或者所有生产者线程都停止生成请求且请求队列为空

while (request\_count == 0 && !request\_produced) {

printf("Request handler waiting for request queue not empty.\n");

pthread\_cond\_wait(&request\_not\_empty, &request\_mutex);

}

if (request\_count == 0 && request\_produced) {

// 所有生产者线程都停止生成请求且请求队列为空，退出请求处理线程

printf("Request handler stopped handling requests.\n");

pthread\_mutex\_unlock(&request\_mutex);

break;

}

// 从请求队列中取出请求

request = request\_queue[request\_out];

request\_out = (request\_out + 1) % REQUEST\_QUEUE\_SIZE;

request\_count--;

printf("Request handler handled request %d: %s\n", request.id, request.data);

// 通知生产者线程请求队列不满

pthread\_cond\_signal(&request\_not\_full);

// 解锁

pthread\_mutex\_unlock(&request\_mutex);

// 根据请求类型将请求分配给相应的消费者线程

int consumer\_id = request.id % NUM\_CONSUMERS;

// 加锁

pthread\_mutex\_lock(&task\_mutexes[consumer\_id]);

// 等待任务队列不满

while (task\_counts[consumer\_id] == TASK\_QUEUE\_SIZE) {

printf("Request handler waiting for task queue %d not full.\n", consumer\_id);

pthread\_cond\_wait(&task\_not\_full[consumer\_id], &task\_mutexes[consumer\_id]);

}

// 将请求放入任务队列

task\_queues[consumer\_id][task\_ins[consumer\_id]] = request;

task\_ins[consumer\_id] = (task\_ins[consumer\_id] + 1) % TASK\_QUEUE\_SIZE;

task\_counts[consumer\_id]++;

printf("Request handler assigned request %d to consumer %d.\n", request.id, consumer\_id);

// 通知消费者线程任务队列不为空

pthread\_cond\_signal(&task\_not\_empty[consumer\_id]);

// 解锁

pthread\_mutex\_unlock(&task\_mutexes[consumer\_id]);

}

return NULL;

}

// 消费者线程函数

void\* consumer(void\* arg) {

int consumer\_id = \*((int\*)arg);

while (true) {

Request request;

// 加锁

pthread\_mutex\_lock(&task\_mutexes[consumer\_id]);

// 等待任务队列不为空

while (task\_counts[consumer\_id] == 0) {

printf("Consumer %d waiting for task queue not empty.\n", consumer\_id);

pthread\_cond\_wait(&task\_not\_empty[consumer\_id], &task\_mutexes[consumer\_id]);

}

// 从任务队列中取出请求

request = task\_queues[consumer\_id][task\_outs[consumer\_id]];

task\_outs[consumer\_id] = (task\_outs[consumer\_id] + 1) % TASK\_QUEUE\_SIZE;

task\_counts[consumer\_id]--;

printf("Consumer %d consumed request %d: %s\n", consumer\_id, request.id, request.data);

// 通知请求处理线程任务队列不满

pthread\_cond\_signal(&task\_not\_full[consumer\_id]);

// 解锁

pthread\_mutex\_unlock(&task\_mutexes[consumer\_id]);

// 处理请求并生成响应数据

Response response;

response.id = request.id;

snprintf(response.data, sizeof(response.data), "Response for request %d from consumer %d", request.id, consumer\_id);

// 加锁

pthread\_mutex\_lock(&response\_mutex);

// 等待响应队列不满

while (response\_count == RESPONSE\_QUEUE\_SIZE) {

printf("Consumer %d waiting for response queue not full.\n", consumer\_id);

pthread\_cond\_wait(&response\_not\_full, &response\_mutex);

}

// 将响应放入响应队列

response\_queue[response\_in] = response;

response\_in = (response\_in + 1) % RESPONSE\_QUEUE\_SIZE;

response\_count++;

printf("Consumer %d produced response %d: %s\n", consumer\_id, response.id, response.data);

// 通知主线程响应队列不为空

pthread\_cond\_signal(&response\_not\_empty);

// 解锁

pthread\_mutex\_unlock(&response\_mutex);

// 睡眠一定时间

sleep(2);

}

return NULL;

}

// 主函数

int main() {

pthread\_t producer\_threads[NUM\_PRODUCERS];

pthread\_t request\_handler\_thread;

pthread\_t consumer\_threads[NUM\_CONSUMERS];

int producer\_ids[NUM\_PRODUCERS];

int consumer\_ids[NUM\_CONSUMERS];

// 初始化互斥锁和条件变量

pthread\_mutex\_init(&request\_mutex, NULL);

pthread\_cond\_init(&request\_not\_empty, NULL);

pthread\_cond\_init(&request\_not\_full, NULL);

for (int i = 0; i < NUM\_CONSUMERS; i++) {

pthread\_mutex\_init(&task\_mutexes[i], NULL);

pthread\_cond\_init(&task\_not\_empty[i], NULL);

pthread\_cond\_init(&task\_not\_full[i], NULL);

}

pthread\_mutex\_init(&response\_mutex, NULL);

pthread\_cond\_init(&response\_not\_empty, NULL);

pthread\_cond\_init(&response\_not\_full, NULL);

// 创建生产者线程

for (int i = 0; i < NUM\_PRODUCERS; i++) {

producer\_ids[i] = i;

pthread\_create(&producer\_threads[i], NULL, producer, &producer\_ids[i]);

}

// 创建请求处理线程

pthread\_create(&request\_handler\_thread, NULL, request\_handler, NULL);

// 创建消费者线程

for (int i = 0; i < NUM\_CONSUMERS; i++) {

consumer\_ids[i] = i;

pthread\_create(&consumer\_threads[i], NULL, consumer, &consumer\_ids[i]);

}

// 等待生产者线程结束

for (int i = 0; i < NUM\_PRODUCERS; i++) {

pthread\_join(producer\_threads[i], NULL);

}

// 等待请求处理线程结束

pthread\_join(request\_handler\_thread, NULL);

// 等待消费者线程结束

for (int i = 0; i < NUM\_CONSUMERS; i++) {

pthread\_join(consumer\_threads[i], NULL);

}

// 销毁互斥锁和条件变量

pthread\_mutex\_destroy(&request\_mutex);

pthread\_cond\_destroy(&request\_not\_empty);

pthread\_cond\_destroy(&request\_not\_full);

for (int i = 0; i < NUM\_CONSUMERS; i++) {

pthread\_mutex\_destroy(&task\_mutexes[i]);

pthread\_cond\_destroy(&task\_not\_empty[i]);

pthread\_cond\_destroy(&task\_not\_full[i]);

}

pthread\_mutex\_destroy(&response\_mutex);

pthread\_cond\_destroy(&response\_not\_empty);

pthread\_cond\_destroy(&response\_not\_full);

return 0;

}