식사하는 철학자

식사하는 철학자 문제 설명

철학자 5명이서 원형 모양의 식탁에 둘러 앉아 생각에 빠지다가 배고플 때 밥을 먹는다.

그들의 양쪽엔 각각 젓가락 한 짝씩 놓여있고, 밥을 먹으려 할 땐 다음의 과정을 따른다.

1. 왼쪽 젓가락부터 집어든다. 만약 다른 철학자가 이미 왼쪽 젓가락을 쓰고 있다면 내려놓을 때까지 생각을 하며 대기를 한다.
2. 왼쪽 젓가락을 들었다면 오른쪽 젓가락을 든다. 만약 들 수 없다면 1번과 마찬가지로 들 수 있을 때까지 생각을 하며 대기한다.
3. 두 젓가락을 모두 들었다면 일정 시간동안 식사를 한다.
4. 식사를 마쳤으면 오른쪽 젓가락을 내려놓은 후 왼쪽 젓가락을 내려놓는다.
5. 다시 생각하다가 배고프면 1번으로 돌아간다.

발생할 수 있는 문제

위의 상황을 프로그램으로 만들어 실행하면 잘 돌아가다가 어느순간 멈춰버리는 것을 알 수 있다

그 이유는 식사하는 철학자 문제가 **데드락 발생의 4가지 필요조건**을 모두 만족하고 있기 때문이다

만약 모든 철학자가 동시에 배가 고파서 왼쪽 젓가락을 집어 든다면 오른쪽 젓가락은 이미 자신의 우측에 앉은 철학자가 집었을 것이므로, 모두가 영원히 오른쪽 젓가락을 들지 못하게 된다. 그렇게 과정 2번에서 더이상 진행하지 못하고 철학자들은 평생 생각을 하게 된다 이런 현상을 **교착상태**라고 한다 한 번 교착상태에 빠진 철학자들은 계속 생각만 하다가 **기아현상**으로 굶어 죽게 된다

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_PHILOSOPHER 5
#define HUNGRY 0;    // 배고픔 상태를 나타내는 값
#define THINKING 1;  // 생각 중인 상태를 나타내는 값

int state[NUM_PHILOSOPHER];
```

```
typedef struct _chopstick_t {
  int value;
  pthread_mutex_t lock;
  pthread_cond_t cond;
} chopstick_t;

chopstick_t chopstick[NUM_PHILOSOPHER];

// 젓가락 초기화 함수
void chopstick_init(chopstick_t *c, int value) {
  c->value = value;
  pthread_mutex_init(&c->lock, NULL);         // 뮤텍스 초기화
  pthread_cond_init(&c->cond, NULL);           // 조건 변수 초기화
}

// 젓가락을 기다리는 함수
void chopstick_wait(chopstick_t *c) {
  pthread_mutex_lock(&c->lock);                 // 뮤텍스 락(lock) 획득
  while(c->value <= 0) {                        // 젓가락의 사용 가능 여부를 확인
    pthread_cond_wait(&c->cond, &c->lock);     // 젓가락이 사용 불가능한 경우 조건 변수 대기
  }
  c->value--;                                   // 젓가락 사용 표시
  pthread_mutex_unlock(&c->lock);               // 뮤텍스 언락(unlock)
}

// 젓가락을 반환하는 함수
void chopstick_post(chopstick_t *c) {
  pthread_mutex_lock(&c->lock);                 // 뮤텍스 락(lock) 획득
  c->value++;                                   // 젓가락 사용 가능 표시
  // 대기 중인 철학자에게 신호를 보내 젓가락 사용 가능을 알림
  pthread_cond_signal(&c->cond);
  pthread_mutex_unlock(&c->lock);               // 뮤텍스 언락(unlock)
}

// 철학자가 젓가락을 가져오는 함수, philosopher는 철학자의 인덱스를 나타냄
void get_chopstick(int philosopher) {
  printf("philosopher %d waits for the left chopstick. \n", philosopher);
  // 철학자는 왼쪽 젓가락을 기다리고 chopstick_wait 함수를 호출하여 젓가락을 얻는다
  chopstick_wait(&chopstick[philosopher]);
  printf("philosopher %d waits for the right chopstick. \n", philosopher);
  // 이후 오른쪽 젓가락을 기다리고 다시 chopstick_wait 함수를 호출하여 젓가락을 얻는다
  chopstick_wait(&chopstick[(philosopher+1)%5]);
  printf("philosopher %d gets the chopsticks. \n", philosopher);
}

// 철학자가 젓가락을 놓는 함수, 마찬가지로 philosopher는 철학자의 인덱스를 나타냄
void put_chopstick(int philosopher) {
  // 철학자는 젓가락을 순서대로 반환하기 위해 chopstick_post 함수를 호출함
  chopstick_post(&chopstick[philosopher]);
  chopstick_post(&chopstick[(philosopher+1)%5]);
}

void *philosopher(int *number) { // 철학자의 동작을 정의하는 함수
  while(1) {
    // 철학자는 계속해서 생각을 하며 printf를 통해 메시지를 출력함
    printf("philosopher %d thinks \n", (int *) number);
    loop(1000000); // loop 함수를 호출하여 잠시 대기함
    get_chopstick((int *) number); // get_chopstick 함수를 호출하여 젓가락을 가져옴
```

```
    // 젓가락을 가져온 후에는 printf를 통해 식사하는 메시지를 출력함
    printf("philosopher %d eats \n", (int *) number);
    loop(1000000); // 다시 loop 함수를 호출하여 잠시 대기함
    put_chopstick((int *) number); // put_chopstick 함수를 호출하여 젓가락을 놓음
    loop(1000000); // loop 함수를 호출하여 잠시 대기함
  }
}

// 철학자가 식사 가능한지 확인하는 함수, i 인덱스의 철학자를 기준으로 주변 철학자들의 상태를 확인함
void test(int i) {
  // 주변 철학자들의 상태가 조건을 만족하면
  if(state[i+1]==0 && state[i]!=1 && state[i+2]!=1) {
    state[i+1]=1; // state[i+1]을 1로 설정하여 식사 가능한 상태로 변경함
  }
}

// loop 함수는 주어진 횟수만큼 루프를 실행하는 함수, count 변수만큼 반복하며 아무 작업도 수행하지 않음
void loop(int count) {
  for(int i=0; i<count; i++);
}

int main(int argc, char *argv[]) {
  for(int i=0; i<NUM_PHILOSOPHER; i++) {
    // chopstick_init 함수를 사용하여 젓가락을 초기화함
    chopstick_init(&chopstick[i], 1);
  }

  pthread_t p[NUM_PHILOSOPHER]; // pthread_t 배열 p를 선언하여 철학자 스레드를 관리함

  printf("[main begin] \n");
  for(int i=0; i<NUM_PHILOSOPHER; i++) {
    // pthread_create 함수를 호출하여 철학자 스레드를 생성함,
    //philosopher 함수를 실행하는 스레드를 생성함
    pthread_create(&p[i], NULL, philosopher, i);
  }

  for(int i=0; i<NUM_PHILOSOPHER; i++) {
  // pthread_join 함수를 사용하여 각 철학자 스레드의 실행이 종료될 때까지 기다림
    pthread_join(p[i], NULL);
  }
  printf("[main end] \n");

  return 0;
}
```

간단히 생각해, 만약 모든 철학자들이 동시에 자신의 왼쪽 포크를 잡는다면, 자기 오른쪽의 포크가 사용 가능해질 때까지 기다려야 한다 이 상태에서는 모든 철학자가 영원히 오른쪽 포크가 사용 가능해질 때까지 무한 대기를 하게 되므로 아무것도 진행할 수 없게 된다, 이를 **교착상태**라 칭한다

1. **상호 배제** : 한 번에 한 개의 프로세스만이 공유자원을 사용할 수 있음

   ex) 철학자들이 젓가락을 공유할 수 있게 된다면 한 사람 당 2개의 젓가락을 가질 수 있게 되어 음식을 먹을 수 있게 되므로 공유할 수 없도록 함

2. **비선점** : 프로세스가 작업을 마친 후 자원을 자발적으로 반환할 때까지 기다림

   ex) 철학자가 다른 철학자의 젓가락을 반환한다면 한 사람 당 2개의 젓가락을 가질 수 있게 됨 따라서 음식을 먹을 수 있게 되므로 젓가락을 빼앗지 않도록 함

3. **점유와 대기** : 프로세스가 할당된 자원을 가진 상태에서 다른 자원을 기다림

   ex) 철학자가 왼쪽 젓가락을 잡은 채로 오른쪽 젓가락을 기다리게 되면 식탁에 남아있는 젓가락이 없어지게 됨

   이 때문에 각 철학자는 모두 1개의 젓가락만을 가질 수 밖에 없으므로 교착 상태가 발생하게 됨

   따라서 왼쪽 젓가락을 잡은 채로 오른쪽 젓가락을 기다려야 함

4. **원형 대기** : 프로세스의 자원 점유 및 점유된 자원의 요구 관계가 원형을 이루면서 대기하는 조건

   ex) 자원 할당 그래프가 원형이면 서로 양보를 한다고 해도 제자리 걸음이 될 것임

   만약 그것을 반복한다면 무한루프처럼 빠져나올 수 없게 됨

데드락 발생의 4가지 조건

1. 상호 배제
2. 비선점
3. 점유와 대기
4. 원형 대기

첫번째 해결방법

왼쪽 젓가락 들기 → 오른쪽 젓가락 들기 → 식사하기 → 왼쪽 젓가락 내려놓기 → 오른쪽 젓가락 내려놓기

이렇게 되면 모두가 동시에 왼쪽 젓가락을 든 상태에서 오른쪽 젓가락을 들고자 하게 된다.

하지만 남은 젓가락이 없으므로 들 수 없고 결국엔 프로세스가 아무 일도 할 수 없게 되면서 데드락이 발생한다.

두번째 해결방법

왼쪽 젓가락 들기 → 오른쪽 젓가락을 들 수 있나 보고 안되면 두 젓가락 모두 내려놓기

이런식으로 하면 모든 프로세스가 동시에 왼쪽 젓가락을 들었을 때 남은 젓가락이 없어서 왼쪽 젓가락을 다시 내려놓고, 또 다시 왼쪽 젓가락을 드는 행위가 반복되게 되어 결국 기아 현상이 발생한다.

세번째 해결방법

왼쪽 젓가락 잡기 → 오른쪽 젓가락이 있나 봤는데 없으면 젓가락을 모두 내려놓기 → 랜덤한 시간동안 기다리기

모두가 동시에 왼쪽 젓가락을 집는다고 하더라도 서로 기다리는 시간이 다르므로 결국엔 어떤 한 사람은 식사를 할 수 있지만, 낮은 확률로 기아 현상이 발생할 수 있다.

따라서 중요한, 완벽해야 하는 프로그램에는 사용할 수 없다.

네번째 해결방법

이진 세마포어로만 포크를 드는 행위(임계구역)를 하는곳을 감싼다

젓가락을 들면 이진 세마포어의 제어를 받게 되므로 어느 순간엔 한 프로세스만이 젓가락을 드는 행위를 할 수 있다.

하지만 이진 세마포어로만 감싸면 임계구역에 들어간 프로세스가 하나 있을 때 다른 프로세스들은 임계구역 밖에서 기다려야 한다.

즉, 한 명이 식사중일 때 반대편의 사람도 식사가 가능하지만 임계구역에 프로세스가 들어가 있기 때문에 식사를 할 수 없게 된다.

따라서 프로세스 별로 세마포어를 따로 두어, 최대한 많은 사람들이 식사를 할 수 있게 해야 한다.

완벽한 해결책 : 점유와 대기, 원형 대기를 제거한 것.

동시에 왼쪽 오른쪽 젓가락을 들게 함 (→ 각 **프로세스**마다 **세마포어**를 **하나씩** 두게 함)

- 오른쪽 젓가락을 든 상태에서 왼쪽 프로세스의 젓가락을 얻을때 까지 기다리는 **점유와 대기**가 **사라짐**
- 각 프로세스는 자기가 젓가락을 들었을 때 또 젓가락을 달라고 요구하지 않으므로 덩달아 **원형 대기**도 **제거됨**

만약 이진 세마포어로 젓가락을 들고 내려놓는 행위를 한번에 한 프로세스만 할 수 있다고 가정했을 때,

1번 프로세스가 젓가락을 들면 **젓가락을 집는 함수**이 **호출**되고, 아무일도 없이 끝나게 됨

그렇게 되면 식사를 하게 되고 **젓가락을 놓는 함수**를 **호출**해야 하는데,

1번 프로세스가 먹고있는 와중에 0번과 2번은 젓가락을 집을 수 없으므로 1번이 식사를 마칠때까지 블락되어야 함

→ 각 프로세스별로 세마포어를 따로 두어서 **1번이 식사중일때 양옆은 1번이 식사를 끝낼때까지 기다리게** 한 것

```
enum {thinking, hungry, eating} state[5]; // 5명의 철학자들의 상태를 나타내는 열거형 선언
semaphore self[5] = 0; // 5명의 철학자들의 자기 자원을 나타내는 세마포어 배열 초기화
semaphore mutex = 1; // 상호 배제를 위한 세마포어 초기화


philosopher i // i번째 철학자의 동작을 수행하는 코드

do {
pickup(i); // i번째 철학자가 식사를 위해 젓가락을 집어들이는 동작
eat(); // 식사를 하는 동작
putdown(i); // i번째 철학자가 젓가락을 내려놓는 동작
think(); // 사고를 하는 동작
} while(1);

void pickup(int i) {
    P(mutex); // 상호 배제를 위해 mutex 세마포어 획득
    state[i] = hungry; // i번째 철학자의 상태를 hungry로 설정
    test(i); // i번째 철학자가 식사 가능한지 확인하는 함수 호출
    V(mutex); // mutex 세마포어 반환
    P(self[i]); // i번째 철학자의 자원(self[i])을 획득하기 위해 대기
}

void test(int i) {
    // 왼쪽, 오른쪽 철학자가 식사 중이 아니고 i가 배고픈 상태일 때
    if (state[(i+4) % 5] != eating && state[i] == hungry)
        && state[(i+1) % 5 != eating] {
            state[i] = eating; // i번째 철학자의 상태를 eating(식사 중)로 설정
            V(self[i]); // i번째 철학자의 자원(self[i])을 획득 가능하게 함
        }
}

void putdown(int i) {
    P(mutex); // 상호 배제를 위해 mutex 세마포어 획득
    state[i] = thinking; // i번째 철학자의 상태를 thinking(사고 중)으로 설정
    test((i+4) % 5); // i번째 철학자의 왼쪽 철학자에게 식사 가능 여부를 확인하는 함수 호출
    test((i+1) % 5); // i번째 철학자의 오른쪽 철학자에게 식사 가능 여부를 확인하는 함수 호출
    V(mutex); // mutex 세마포어 반환
}
```