1.

(1)

Work = (1, 0, 1, 2)

Need = Max − Alloc

| | A | B | C | D | |
|---|---|---|---|---|---|
| $P_1$ | 2 | 1 | 0 | 0 | T |
| $P_2$ | 0 | 0 | 2 | 1 | T |
| $P_3$ | 1 | 0 | 0 | 1 | T |
| $P_4$ | 0 | 1 | 1 | 1 | T |

Finish[3] = false, $Need_3 \leq Work$

∴ Work += $Alloc_3$ → (1, 0, 2, 2)

Finish[3] = true

Finish[2] = false, $Need_2 \leq Work$

∴ Work += $Alloc_2$ → (1, 1, 2, 3)

Finish[2] = true

Finish[1] = false, $Need_1 \leq Work$

∴ Work += $Alloc_1$ → (1, 3, 3, 3)

Finish[1] = true

Finish[4] = false, $Need_4 \leq Work$

∴ Work += $Alloc_4$ → (2, 4, 3, 3)

Finish[4] = true

∴ sequence <$P_3$, $P_2$, $P_1$, $P_4$> satisfies safety requirement

(2) $Request_4 = (0, 0, 1, 1) \leq Need_4 (0, 1, 1, 1)$

↓

doesn't satisfy $Request_4 = (0, 0, 1, 1) \leq Available (1, 0, 1, 2)$

↓

P4 waits, insufficient resource

(3)

| Alloc | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 0 | 2 | 1 | 0 |
| $P_2$ | 0 | 1 | 0 | 1 |
| $P_3$ | 0 | 0 | 1 | 0 |
| $P_4$ | 1 | 1 | 1 | 1 |

| Request | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 2 | 1 | 0 | 0 |
| $P_2$ | 0 | 0 | 1 | 0 |
| $P_3$ | 1 | 0 | 0 | 0 |
| $P_4$ | 0 | 1 | 0 | 0 |

Use deadlock detection algorithm

Work = (1, 0, 0, 1)

i = 3, Finish[3] = false and $Request_4 \leq Work$

Work += $Alloc_3$ → (1, 0, 1, 1)

Finish[3] = true

i = 2, Finish[2] = false and $Request_2 \leq Work$

Work += $Alloc_2$ → (1, 1, 1, 2)

Finish[2] = true

i = 4, Finish[4] = false and $Request_4 \leq Work$

Work += $Alloc_4$ → (2, 2, 2, 3)

Finish[4] = true

---

i = 1, Finish[1] = false and $Request_1 \leq Work$

Work += $Alloc_4$ → (2, 4, 3, 3)

Finish[1] = true

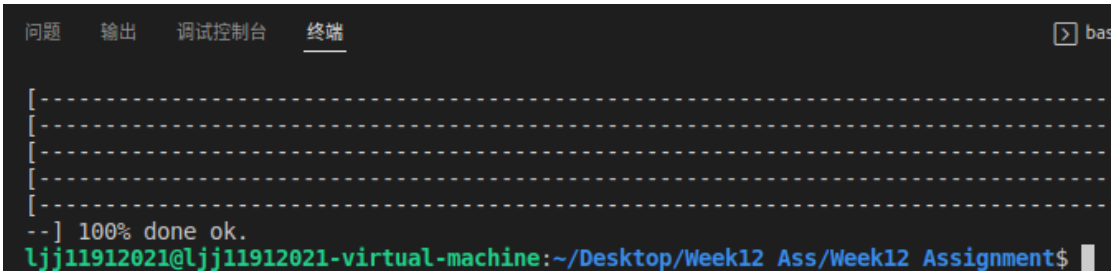
∴ For all i, Finish[i] = true

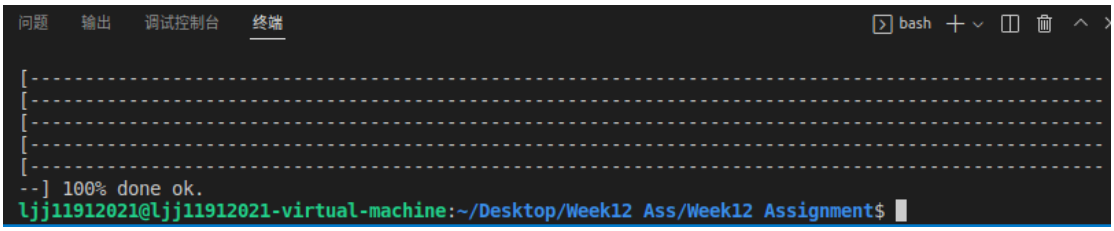
∴ No deadlock

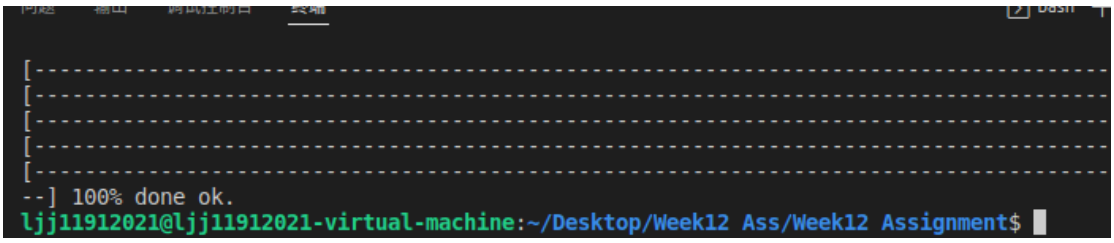
2. Design idea one: Use mutex lock. Only one person eats at a time.

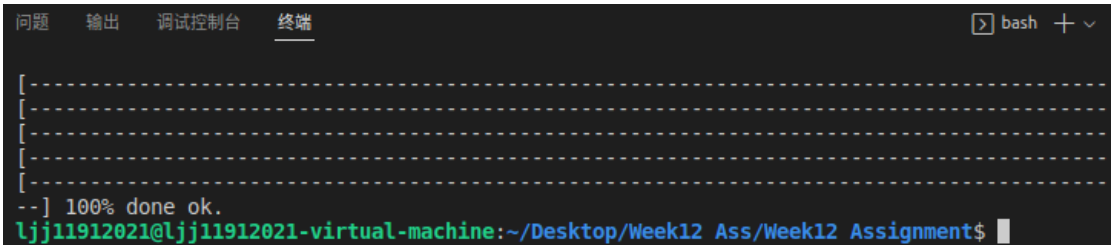
```
82  pthread_mutex_t mutex;
83  void init() {
84      // write code if you desire.
85
86  }
87
88  void wants_to_eat(int p_no) {
89      // fixme
90      pthread_mutex_lock(&mutex);
91      pick_right_fork(p_no);
92      pick_left_fork(p_no);
93
94      eat(p_no);
95
96
97      put_left_fork(p_no);
98      put_right_fork(p_no);
99      pthread_mutex_unlock(&mutex);
100
101 }
```

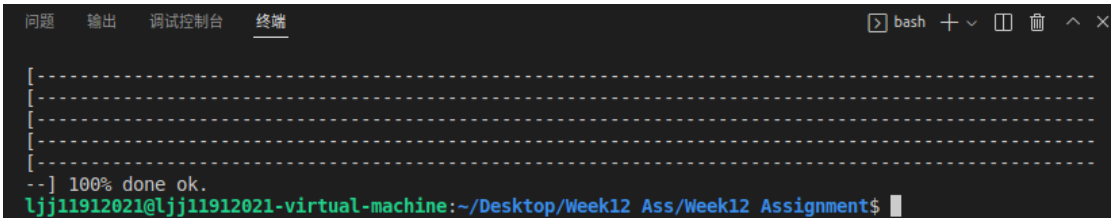
Result (5 times):



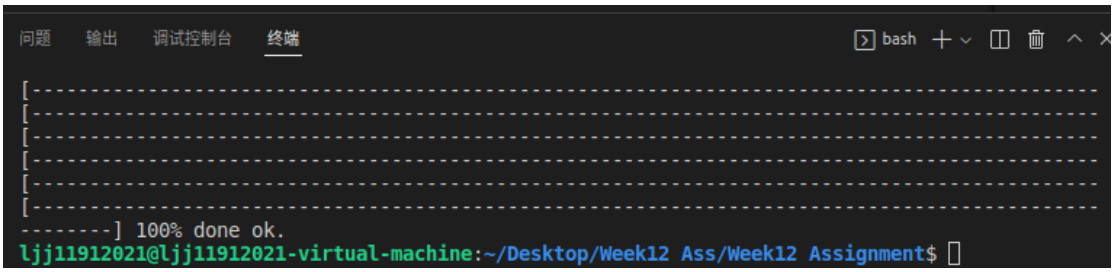



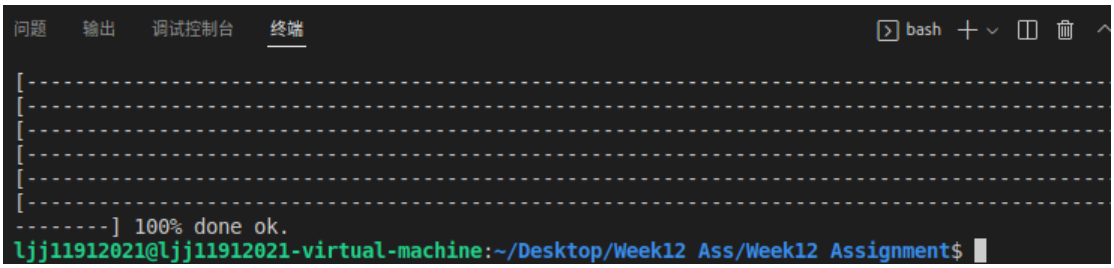


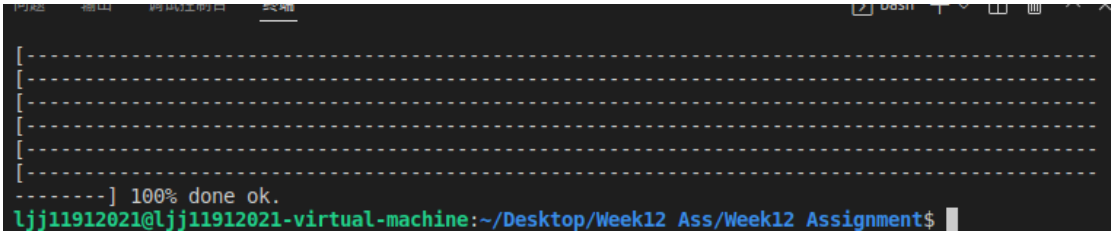

Design idea two: Use each person's number. The ones with even number pick their left forks first and then pick the right forks. The ones with odd number pick their right forks first and then pick the left forks.

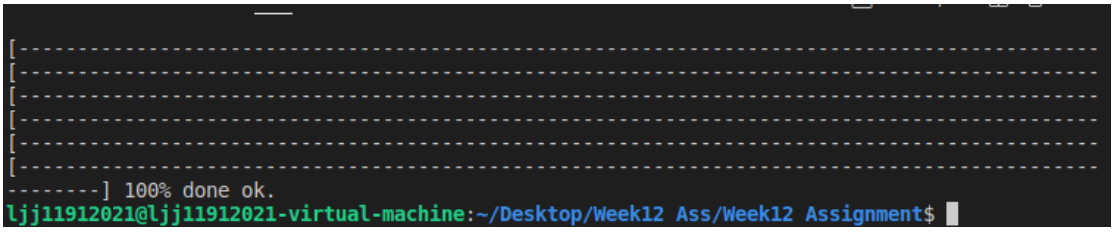
```
void init() {
    // write code if you desire.

}

void wants_to_eat(int p_no) {
    // fixme

    if (p_no % 2 == 0) {
        pick_left_fork(p_no);
        pick_right_fork(p_no);
    }
    else {
        pick_right_fork(p_no);
        pick_left_fork(p_no);
    }

    eat(p_no);

    put_left_fork(p_no);
    put_right_fork(p_no);
}
```

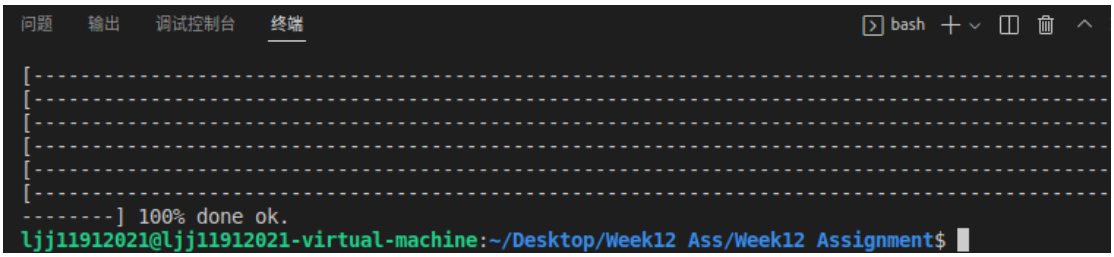
Result (5 times):






3. Design idea: Use two semaphores: sem_drink and sem_buy. Sem_drink is initialized as 2 and sem_buy is initialized as 0. At the beginning of every producer (mom, dad, grandfather), execute wait(sem_drink), and at the end of every producer, execute post(sem_buy). At the beginning of consumer (son), execute wait(sem_buy), and at the end of consumer, execute post(sem_drink). Therefore, before the fridge is empty, there would always be someone to buy milk, and before the fridge is full, there would always be the consumer to drink milk. Therefore the fridge will not be empty or full.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>


pthread_mutex_t fri_lock;
sem_t * sem_buy;
sem_t * sem_drink;

void *mom(int *num){
    for(int i=0;i<10;i++){
        sem_wait(sem_drink);
        pthread_mutex_lock(&fri_lock);
        printf("Mom comes home.\n");
        sleep(rand()%2+1);
        printf("Mom goes to buy milk.\n");
        *num += 1;
        if (*num > 2){
            printf("What a waste of food! The fridge can not hold
            while(1)printf("TAT~");
        }
        printf("Mom puts milk in fridge and leaves.\n");
        pthread_mutex_unlock(&fri_lock);
        sem_post(sem_buy);
    }
}
```

```c
void *dad(int *num){
    for(int i=0;i<10;i++){
        sem_wait(sem_drink);
        pthread_mutex_lock(&fri_lock);
        printf("Dad comes home.\n");
        sleep(rand()%2+1);
        printf("Dad goes to buy milk.\n");
        *num += 1;
        if (*num > 2){
            printf("What a waste of food! The fridge can not hold
            while(1)printf("TAT~");
        }
        printf("Dad puts milk in fridge and leaves.\n");
        pthread_mutex_unlock(&fri_lock);
        sem_post(sem_buy);
    }
}
```

```c
void *grandfather(int *num){
    for(int i=0;i<10;i++){
        sem_wait(sem_drink);
        pthread_mutex_lock(&fri_lock);
        printf("Grandfather comes home.\n");
        sleep(rand()%2+1);
        printf("Grandfather goes to buy milk.\n");
        *num += 1;
        if (*num > 2){
            printf("What a waste of food! The fridge can not hold
            while(1){
                printf("TAT~");
            }
        }
        printf("Grandfather puts milk in fridge and leaves.\n");
        pthread_mutex_unlock(&fri_lock);
        sem_post(sem_buy);
    }
}
```

```c
void *son(int *num){
    for(int i = 0; i < 30 ; i++){
        sem_wait(sem_buy);
        pthread_mutex_lock(&fri_lock);
        printf("Son comes home.\n");
        if(*num == 0){
            printf("The fridge is empty!\n");
            while(1){
                printf("TAT~");
            }
        }
        printf("Son fetches a milk\n");
        *num -= 1;
        printf("Son leaves\n");
        pthread_mutex_unlock(&fri_lock);
        sem_post(sem_drink);
    }
}
```

```c
int main(int argc, char * argv[]) {
    srand(time(0));

    int num_milk = 0;
    pthread_t p1, p2, p3, p4;
    pthread_mutex_init(&fri_lock,NULL);

    sem_drink = sem_open("empty", O_CREAT, 0666, 2);
    sem_buy = sem_open("full", O_CREAT, 0666, 0);

    // Create two threads (both run func)
    pthread_create(&p1, NULL, mom, &num_milk);
    pthread_create(&p2, NULL, dad, &num_milk);
    pthread_create(&p3, NULL, grandfather, &num_milk);
    pthread_create(&p4, NULL, son, &num_milk);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);
    pthread_join(p4, NULL);

    sem_close(sem_buy);
    sem_close(sem_drink);

    printf("success!\n");
}
```

Result: