INT2214 Bài tập #2: Tiến trình (Processes)

Nguyễn Thành Đô - 19020250

Giảng viên: Bùi Duy Hiếu

Hạn nộp: Thứ hai, 07/03/2022 trước **23h:59'**

1 Bài toán Nhà sản xuất – Người dùng

Hai tiến trình chia sẻ một vùng đệm có kích thước cố định, trao đổi với nhau qua phương thức trao đổi thông điệp. Nhà sản xuất muốn đưa thông tin lên vùng đệm, còn người dùng muốn lấy thông tin từ vùng đệm. Giả định tình huống vùng đệm đầy mà nhà sản xuất vẫn muốn đưa thông tin lên. Vậy nhà sản xuất phải chuyển sang trạng thái ngủ, được đánh thức bởi người dùng khi họ lấy thông tin từ vùng đệm. Tương tự, khi vùng đệm trống và người dùng muốn lấy thông tin từ vùng đệm, thì người dùng sẽ chuyển sang trạng thái ngủ và được đánh thức bởi nhà sản xuất khi họ đưa thông tin lên vùng đệm.

Yêu cầu: Viết chương trình minh hoạ cho tình huống trên.

1.1 Giới thiệu vấn đề

Truyền thông liên tiến trình sử dụng bộ nhớ dùng chung (interprocess communication) yêu cầu các tiến trình phải thiết lập một vùng bộ nhớ dùng chung. Thông thường, bộ nhớ chung nằm ở không gian địa chỉ của tiến trình khởi tạo ra bộ nhớ đó. Các tiến trình khác muốn truyền thông sử dụng bộ nhớ chung này phải liên kết nó vào không gian địa chỉ của các tiến trình này. Bình thường hệ điều hành sẽ cố gắng ngăn chặn một tiến trình truy cập vào bộ nhớ của một tiến trình khác, do đó bộ nhớ chung yêu cầu các tiến trình phải tháo bỏ ràng buộc này. Sau đó chúng có thể trao đổi thông tin bằng cách đọc và ghi dữ liệu lên bộ nhớ chung.

Bài toán Nhà sản xuất - Người dùng minh họa cho khái niệm truyền thông liên tiến trình, ở đó nhà sản xuất và người dùng phải được **đồng bộ hóa**, để tránh trường hợp người dùng cố gắng lấy thông tin mà chưa được đưa lên vùng đệm. Tuy nhiên, cách tiếp cận bài toán này sẽ tạm thời bỏ qua vấn đề liên quan đến tình huống trong đó cả nhà sản xuất và người dùng đều cố gắng truy cập đồng thời bộ đệm dùng chung.

1.2 Thiết kế và cài đặt

Code đầy đủ được cung cấp tại: `https://github.com/dont-penciler/INT2214/tree/main/a2/code`

1.2.1 Hướng tiếp cận 1

Bộ đệm chung được cài đặt như một array xoay vòng với hai con trỏ *in* và *out*. Biến *in* trỏ tới vị trí còn trống tiếp theo trong bộ đệm; biến *out* trỏ tới vị trí đầy (có chứa item được đưa lên bởi nhà sản xuất) đầu tiên trong bộ đệm. Nhà sản xuất và người dùng được cài đặt như hai luồng chạy song song, nhận biết trạng thái của bộ đệm thông qua hai biến *in* và *out* để thực hiện các hành vi phù hợp. Cụ thể, bộ đệm sẽ được hiểu là rỗng khi $in == out$; và bộ nhớ được hiểu là đầy khi $((in + 1) \% \text{BUFFER\_SIZE}) == out$.

Để mô phỏng nhà sản xuất và người dùng, cài đặt hai threads chạy song song với dữ liệu toàn cục *buffer* được sử dụng, cụ thể trong Hình 1 và Hình 2. Thread "nhà sản xuất" có một biến cục bộ *produced* chứa item mới chuẩn bị được đưa lên bộ đệm; thread "người dùng" có một biến cục bộ *consumed* chứa item chuẩn bị được người dùng lấy về.

```c
void *producer(void *args){
    while (1){
        int produced = rand() % 100;
        while ((in + 1) % BUFFER_SIZE == out){
            // buffer is full, do nothing
            printf("Buffer is full: %d/%d slots are occupied\n",
                   in - out >= 0 ? (in - out) : BUFFER_SIZE - (out - in),
                   BUFFER_SIZE);
            sleep(1);
        }
        buffer[in] = produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("produced: %d, %d/%d slots are occupied\n", produced,
        in - out >= 0 ? (in - out) : BUFFER_SIZE - (out - in), BUFFER_SIZE);
        sleep(1);
    }
    return 0;
}
```

Hình 1: Code cho nhà sản xuất #1

```c
void *consumer(void *args){
    while (1){
        while (in == out){
            // buffer is empty, do nothing
            printf("Buffer is empty: %d/%d slots are occupied\n",
                    in - out >= 0 ? (in - out) : BUFFER_SIZE - (out - in),
                        BUFFER_SIZE);
            sleep(1);
        }
        int consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("consumed: %d, %d/%d slots are occupied\n", consumed,
        in - out >= 0 ? (in - out) : BUFFER_SIZE - (out - in), BUFFER_SIZE);
        sleep(2);
    }
    return 0;
}
```

Hình 2: Code cho người dùng #1

Thử chạy chương trình với 2 testcases (Hình 3), tương ứng với hai trường hợp tốc độ đưa item lên lớn hơn tốc độ lấy item về (testcase 1), và ngược lại (testcase 2). Dễ nhận thấy, cách cài đặt này chỉ cho phép tối đa BUFFER_SIZE - 1 items cùng tồn tại ở trên bộ đệm. Điều này thể hiện rõ qua kết quả chạy thử testcase 1, với BUFFER_SIZE = 5 thì chỉ có tối đa 4/5 bộ đệm được chiếm giữ. Mặt khác vấn đề đồng bộ hóa cũng chưa được giải quyết bằng cách cài đặt này, như có thể thấy trong hai dòng đầu tiên của testcase 1: chương trình in ra sự kiện "consumed: 97" trước sự kiện "produced: 97".

```
consumed: 97, 0/5 slots are occupied
produced: 97, 1/5 slots are occupied
produced: 49, 1/5 slots are occupied
consumed: 49, 0/5 slots are occupied
produced: 19, 1/5 slots are occupied
produced: 54, 2/5 slots are occupied
consumed: 19, 1/5 slots are occupied
produced: 71, 2/5 slots are occupied
produced: 85, 3/5 slots are occupied
consumed: 54, 2/5 slots are occupied
produced: 97, 3/5 slots are occupied
produced: 54, 4/5 slots are occupied
consumed: 71, 3/5 slots are occupied
produced: 24, 4/5 slots are occupied
Buffer is full: 4/5 slots are occupied
consumed: 85, 3/5 slots are occupied
produced: 75, 4/5 slots are occupied
Buffer is full: 4/5 slots are occupied
consumed: 97, 3/5 slots are occupied
produced: 44, 4/5 slots are occupied
Buffer is full: 4/5 slots are occupied
consumed: 54, 3/5 slots are occupied
produced: 59, 4/5 slots are occupied
Buffer is full: 4/5 slots are occupied
```

(a) Testcase 1: $v_{produce} > v_{consume}$

```
produced: 71, 1/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
consumed: 71, 0/5 slots are occupied
produced: 86, 1/5 slots are occupied
consumed: 86, 0/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
produced: 38, 1/5 slots are occupied
consumed: 38, 0/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
produced: 76, 1/5 slots are occupied
consumed: 76, 0/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
produced: 80, 1/5 slots are occupied
consumed: 80, 0/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
produced: 11, 1/5 slots are occupied
consumed: 11, 0/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
produced: 84, 1/5 slots are occupied
consumed: 84, 0/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
produced: 38, 1/5 slots are occupied
consumed: 38, 0/5 slots are occupied
Buffer is empty: 0/5 slots are occupied
```

(b) Testcase 2: $v_{produce} < v_{consume}$

Hình 3: Kết quả thử nghiệm #1

1.2.2 Hướng tiếp cận 2

Hướng tiếp cận này sẽ cải thiện việc sử dụng không gian bộ nhớ chưa hiệu quả của hướng tiếp cận trước. Với mục đích đó, ta có thể thêm một biến đếm *count*, được khởi tạo với giá trị 0. *count* được tăng lên mỗi khi nhà sản xuất thêm dữ liệu vào bộ đệm và được giảm xuống mỗi khi người dùng lấy dữ liệu ra khỏi bộ đệm. Code cho nhà sản xuất và người dùng được điều chỉnh như sau dưới đây (Hình 4-5).

Với sự điều chỉnh này, nhà sản xuất và người dùng có thể đặt được hiệu quả tối ưu (sử dụng được tối đa BUFFER_SIZE kích thước bộ đệm) nếu hoạt động tách biệt nhau. Nhưng ngược lại, chúng có thể gây ra lỗi nếu như hoạt động đồng thời. Ví dụ, giả sử giá trị của biến *count* = 5, khi đó nhà sản xuất và người dùng đồng thời thực các đoạn lệnh *count++* và *count−−*. Tùy theo thứ tự thực hiện các đoạn lệnh này ở mức độ ngôn ngữ máy, mà giá trị của *count* có thể nhận được là 4, 5 hoặc 6. Tuy nhiên khi chạy thử cài đặt này, trong giới hạn thời gian chạy 10000000 vòng lặp, chúng ta vẫn chưa gặp phải vấn đề này.

```c
void *producer(void *args){
    while (1){
        int produced = rand() % 100;
        while (count == BUFFER_SIZE){
            printf("Buffer is full, count = %d\n", count);
            sleep(1);
        }

        buffer[in] = produced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        printf("produced: %d, count = %d\n", produced, count);
        sleep(1);
    }
    return 0;
}
```

Hình 4: Code cho nhà sản xuất #2

```c
void *consumer(void *args){
    while (1){
        while (count == 0){
            printf("Buffer is empty, count = %d\n", count);
            sleep(2);
        }
        // do nothing
        int consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        printf("consumed: %d, count = %d\n", consumed, count);
        sleep(2);
    }

    return 0;
}
```

Hình 5: Code cho người dùng #2

```
produced: 18, count = 1
consumed: 18, count = 0
produced: 78, count = 1
consumed: 78, count = 0
produced: 47, count = 1
produced: 42, count = 2
consumed: 47, count = 1
produced: 44, count = 2
produced: 22, count = 3
consumed: 42, count = 2
produced: 63, count = 3
produced: 58, count = 4
consumed: 44, count = 3
produced: 27, count = 4
produced: 67, count = 5
consumed: 22, count = 4
produced: 84, count = 5
Buffer is full, count = 5
consumed: 63, count = 4
produced: 60, count = 5
Buffer is full, count = 5
consumed: 58, count = 4
produced: 40, count = 5
Buffer is full, count = 5
```

(a) Testcase 3: $v_{produce} > v_{consume}$

```
Buffer is empty, count = 0
produced: 87, count = 1
consumed: 87, count = 0
produced: 30, count = 1
consumed: 30, count = 0
Buffer is empty, count = 0
produced: 93, count = 1
consumed: 93, count = 0
Buffer is empty, count = 0
produced: 84, count = 1
consumed: 84, count = 0
Buffer is empty, count = 0
produced: 70, count = 1
consumed: 70, count = 0
Buffer is empty, count = 0
produced: 69, count = 1
consumed: 69, count = 0
Buffer is empty, count = 0
produced: 5, count = 1
consumed: 5, count = 0
Buffer is empty, count = 0
produced: 32, count = 1
consumed: 32, count = 0
Buffer is empty, count = 0
```

(b) Testcase 4: $v_{produce} < v_{consume}$

Hình 6: Kết quả thử nghiệm #2

2 Bài toán Bữa ăn của các nhà triết học

Có 5 nhà triết học ngồi trong một chiếc bàn tròn. Mỗi nhà triết học có một đĩa spaghetti. Vì spaghetti rất trơn nên mỗi nhà triết học phải dùng 2 chiếc nĩa để ăn. Giữa hai đĩa spaghetti chỉ có một chiếc nĩa. Vòng đời của nhà triết học là một chuỗi liên tiếp ăn và nghĩ. Khi nhà triết học đói, họ sẽ lấy nĩa bên trái và phải của minh cùng lúc. Nếu thành công, họ sẽ ăn một lúc rồi để nĩa xuống, rồi tiếp tục nghĩ.

Yêu cầu: Viết chương trình mô phỏng hành động các nhà triết học trên sao cho không xảy ra tình huống bế tắc.

2.1 Giới thiệu vấn đề

Bài toán Bữa ăn của các nhà triết học (the dining philosophers problem) là một bài toán kinh điển về đồng bộ hóa. Bản thân bài toán không mang ý nghĩa thực tế quan trọng gì hay thể hiện sự mỉa mai của các nhà khoa học máy tính lên các nhà triết học, mà nó chỉ là một minh họa đơn giản cho nhu cầu cấp phát các tài nguyên cho các tiến trình khác nhau, sao cho không xảy ra hiện tượng **deadlock** và **starvation**.

Có một điều quan trọng trong ngữ cảnh bài toán này đó là, nhà triết học không thể một lúc nhấc hai chếc nĩa được mà phải làm lần lượt: nhấc nĩa trái trước rồi đến nĩa phải, hoặc ngược lại. Hiện tượng deadlock xảy ra khi, tất cả các nhà triết học cùng một lúc nhấc một chiếc nĩa, và tiếp theo không thể nhấc thêm cái thứ hai và phải chờ mãi mãi, do đó tất cả các nhà triết học sẽ chết đói (starvation). Để mô phỏng lại bài toán, chúng ta sẽ giả sử hành động của các nhà triết học là đồng bộ (tức là một lúc hai nhà triết học sẽ không cùng lấy một cái nĩa, gây ra xung đột) và tập trung vào giao thức giao tiếp giữa họ.

```
        while (true) {
            /* think for a while */
            think();
            /* calls pickup to pick up the forks */
            pickup(); // pick up the forks
            /* eat for a while */
            eat();
            /* the philosopher is done eating,
                and calls putdown to put down the forks */
            putdown();

        }
```

Hình 7: Minh họa cách hoạt động của nhà triết học

2.2 Thiết kế và cài đặt

Code đầy đủ được cung cấp tại: `https://github.com/dont-penciler/INT2214/tree/main/a2/code`

Phần cài đặt này tham khảo chủ yếu từ C560 Lecture notes – Dining Philosophers, có sử dụng các kỹ thuật **mutex, semaphore, monitor** để đảm bảo sự đồng bộ giữa các nhà triết học. Tuy nhiên báo cáo này sẽ chỉ tận dụng các kỹ thuật trên mà không giải thích, nguyên nhân là do giới hạn kiến thức của người viết trong thời gian thực hiện báo cáo. Thay vào đó, báo cáo sẽ tập trung vào thiết kế logic cách nhà triết học hoạt động, cụ thể là cách cài đặt các hàm *think()*, *pickup()*, *eat()* và *putdown()*.

Mặt khác, không ai có thể quyết định một nhà triết học sẽ ăn và nghĩ lúc nào, trong bao lâu (thực ra vẫn cần giới hạn thời gian ăn tối đa của nhà triết học, giống như việc không thể để một process chiếm tài nguyên CPU mãi được.) Hàm *think()* và *eat()* được cài đặt dựa trên ý tưởng nhà triết học sẽ ăn và nghĩ trong một khoảng thời gian ngẫu nhiên, trong một giới hạn MAX_TIME cho trước. Các lời giải khác nhau sẽ chỉ khác nhau ở cách cài đặt hai hàm *pickup()* và *putdown()*.

```
        void think() {
            duration = (random() % MAX_TIME) + 1;
            sleep(duration);
        }
        void eat() {
            duration = (random() % MAX_TIME) + 1;
            sleep(duration);
        }
```

Hình 8: Cài đặt hàm *think()* và *eat()*

Đồng thời để đánh giá hiệu quả từng cài đặt, thông tin về thời gian chờ của từng nhà triết học (tổng thời gian những khoảng chờ từ lúc muốn ăn cho đến lúc có đủ nĩa để ăn) sẽ được ghi lại.

2.2.1 Hướng tiếp cận đơn giản và hiện tượng deadlock

Trong hướng tiếp cận này, đơn giản mỗi nhà triết học sẽ thực hiện việc ăn theo quy trình sau: đầu tiên kiểm tra xem nĩa bên trái có ai dùng không, nếu có thì đợi, nếu không thì cầm nĩa lên, sau đó làm tương tự với nĩa bên phải; sau khi ăn xong thì đặt nĩa phải xuống trước, rồi sau đó đến nĩa trái (Hình 9). Cách tiếp cận này không ngăn chặn được hiện tượng deadlock, tuy nhiên khi chạy thử sẽ không gặp phải tình huống đó bởi vì sự ngẫu nhiên trong thời gian ăn và nghĩ của của từng nhà triết học. Deadlock chỉ xảy ra khi tất cả nhà triết học bị chen ngang (preempted) giữa lúc nhấc hai chiếc nĩa, dẫn đến mỗi người bọn họ chỉ cầm được một chiếc nĩa và sẽ chờ để có chiếc nĩa thứ hai cho đến khi chết đói. Do vậy để mô phỏng tình huống này ta chỉ cần thêm một khoảng chờ đủ lâu giữa hai lần nhấc nĩa (Hình x).

```
typedef struct forks {
    pthread_mutex_t **lock; // for synchronization
    int phil_count; // number of philosophers, = 5 in this case
} Forks;
// a Phil_struct struct contains information of a philosopher
void pickup(Phil_struct *ps) {
    Forks *forks;
    int phil_count;

    forks = (Forks *)ps->v; // get info of forks of the philosphers
    phil_count = forks->phil_count;
    /* pick up left fork and make sure others won't try to pick it,
     in other words, lock up left fork */
    pthread_mutex_lock(forks->lock[ps->id]);
    /* pick up and lock up right fork */
    pthread_mutex_lock(forks->lock[(ps->id + 1) % phil_count]);
}

void putdown(Phil_struct *ps) {
    Forks *forks;
    int i;
    int phil_count;

    forks = (Forks *)ps->v;
    phil_count = forks->phil_count;
    /* release the right fork after done eating,
        in other words, unlock right fork */
    pthread_mutex_unlock(forks->lock[(ps->id + 1) % phil_count]);
    /* unlock left fork */
    pthread_mutex_unlock(forks->lock[ps->id]);
}
```

Hình 9: Cài đặt đơn giản hàm *pickup()* và *putdown()*

Kết quả chạy thử với 5 nhà triết học, thời gian ăn/nghĩ tối đa 10 giây, trong tổng cộng 2100 giây không cho thấy deadlock. Kết quả đầy đủ của lần chạy mẫu này được cung cấp trong file dp2.txt, với tổng thời gian chờ được ghi lại tại dòng cuối như sau:

```
2100 Total blocktime:  5165 :  1026  1041  1047  1042  1009
```

Tuy nhiên, như đã nói, để mô phỏng tình huống deadlock chúng ta có thể để thời gian giữa hai lần nhấc nĩa là 5 giây. Cụ thể ta có thể điều chỉnh một phần code của hàm *pickup()* như sau:

```
pthread_mutex_lock(forks->lock[ps->id]);
sleep(5);
pthread_mutex_lock(forks->lock[(ps->id + 1) % phil_count]);
```

Hình 10: Cài đặt hàm *pickup()* mô phỏng tình huống deadlock

Khi chạy thử ta có thể thấy hiện tượng deadlock xảy ra ngay trong 10 giây đầu tiên:

```
0 Philosopher 0 thinking for 4 seconds
0 Philosopher 1 thinking for 3 seconds
0 Philosopher 2 thinking for 5 seconds
0 Philosopher 3 thinking for 6 seconds
0 Philosopher 4 thinking for 2 seconds
0 Total blocktime:     0 :     0     0     0     0     0
2 Philosopher 4 no longer thinking -- calling pickup()
3 Philosopher 1 no longer thinking -- calling pickup()
4 Philosopher 0 no longer thinking -- calling pickup()
5 Philosopher 2 no longer thinking -- calling pickup()
6 Philosopher 3 no longer thinking -- calling pickup()
10 Total blocktime:    30 :     6     7     5     4     8
20 Total blocktime:    80 :    16    17    15    14    18
30 Total blocktime:   130 :    26    27    25    24    28
```

Hình 11: Kết quả chạy thử mô phỏng tình huống deadlock

2.2.2 Lời giải không đối xứng

Để giải quyết vấn đề deadlock chỉ cần điều chỉnh nhỏ so với cách cài đặt trước: chỉ các nhà triết học được đánh số lẻ bắt đầu nhấc nĩa trái trước, trong khi đó các nhà triết học được đánh số chẵn sẽ bắt đầu với nĩa phải trước; tương tự ai số lẻ sẽ đặt nĩa phải xuống trước, ai số chẵn sẽ đặt nĩa trái xuống trước (Hình 12). Sự không đồng đều này giải thích cho cái tên "lời giải không đối xứng." Với thuật toán này, deadlock sẽ không xảy ra kể cả khi ta có thêm một khoảng delay giữa hai lần nhấc nĩa. Lí giải cho sự hiệu quả của thuật toán này khá đơn giản, với quy luật được mô tả như trên thì nếu tất cả các nhà triết học cùng muốn ăn một lúc, thì sẽ luôn có người phải chờ người khác ăn xong rồi mới có thể nhấc chiếc dĩa đầu tiên lên được, cho nên không thể xảy ra trường hợp mỗi nhà triết học đều cầm một chiếc nĩa một lúc.

```
void pickup(Phil_struct *ps) {
  Forks *pp;
  int phil_count;
  pp = (Forks *) ps->v;
  phil_count = pp->phil_count;

  if (ps->id % 2 == 1) { // odd labeled philosopher
    /* lock up left fork */
    pthread_mutex_lock(pp->lock[ps->id]);
    /* lock right fork */
    pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]);
  } else { // even labeled philosopher
    /* lock right fork */
    pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]);
    /* lock up left fork */
    pthread_mutex_lock(pp->lock[ps->id]);
  }
}

void putdown(Phil_struct *ps) {
  Forks *pp;
  int i;
  int phil_count;
  pp = (Forks *) ps->v;
  phil_count = pp->phil_count;

  if (ps->id % 2 == 1) { // odd labeled philosopher
    /* unlock right fork */
    pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]);
    /* unlock left fork */
    pthread_mutex_unlock(pp->lock[ps->id]);
  } else { // even labeled philosopher
    /* unlock left fork */
    pthread_mutex_unlock(pp->lock[ps->id]);
    /* unlock right fork */
    pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]);
  }
}
```

Hình 12: Cài đặt lời giải không đối xứng cho hàm *pickup()* và *putdown()*

Kết quả chạy thử với 5 nhà triết học, thời gian ăn/nghĩ tối đa 10 giây, trong tổng cộng 2100 giây không cho thấy deadlock. Kết quả đầy đủ của lần chạy mẫu này được cung cấp trong file dp4.txt, với tổng thời gian chờ được ghi lại tại dòng cuối như sau:

```
        2100 Total blocktime:  3184 :   753   688   642   580   521
```

2.3 Đánh giá và kết luận

Với hướng tiếp cận đơn giản, sự ngẫu nhiên trong thời gian ăn/nghĩ của các nhà triết học khiến nó vẫn hiệu quả trong một khoảng thời gian giới hạn. Tuy nhiên, cũng chính vì thế mà xác suất xảy ra deadlock vẫn tồn tại, đồng thời thông tin về tổng thời gian chờ của các nhà triết học trong 2100 giây cho thấy sự thiếu hiệu quả về năng suất khi mất tới 5165 giây.

Với cách tiếp cận qua lời giải không đối xứng, hiện tượng deadlock không thể xảy ra, thỏa mãn hoàn toàn được yêu cầu bài toán. Đồng thời thuật toán này cũng cho thấy sự hiệu quả về năng suất hơn so với cách tiếp cận đơn giản, khi tổng thời gian chờ chỉ mất 3184 giây với cùng các tham số khởi chạy. Tuy nhiên, cách tiếp cận này vẫn có một số vấn đề. Thứ nhất, nó vẫn có xác suất khiến nhà triết học chết đói (starvation). Ví dụ, nhà triết học A đang đợi một chiếc nĩa mà nhà triết học B đang dùng để ăn, sau cùng nhà triết học B ăn xong và bỏ nĩa xuống, nhưng không có gì đảm bảo là A sẽ lấy được chiếc nĩa đó nếu như B muốn ăn tiếp ngay sau đó, trước khi A kịp nhấc nĩa lên. Trong cài đặt ở Hình 12, hệ thống của chúng ta sẽ rất khó rơi vào tình huống đó vì thời gian ăn/nghĩ được cài đặt ngẫu nhiên, tuy nhiên đây có thể là vấn đề trong một hệ thống khác. Thứ hai, các nhà triết học không được đối xử công bằng với thuật toán này. Có thể thấy rằng nhà triết học #4 có thời gian chờ ít hơn hẳn so với các nhà triết học khác. Lí do cho sự bất công này là vì với những khi tất cả các nhà triết học đều cùng muốn ăn, thì #0 và #1 sẽ phải giành nhau chiếc nĩa đầu tiên, tương tự với #2 và #3. Trong khi đó #4 sẽ luôn có được chiếc nĩa đầu tiên, khiến cho #4 có được lợi thế hơn so với các nhà triết học còn lại, do đó #4 sẽ được ăn nhiều hơn. Cho nên nếu như muốn thiết kế một hệ thống phân chia tài nguyên công bằng, thì ta sẽ không thể sử dụng cách tiếp cận này.

Thực tế, tồn tại các lời giải giải quyết được cả vấn đề starvation và phân chia tài nguyên công bằng. Ví dụ để không nhà triết học nào có lợi thế hơn phần còn lại, về mặt ý tưởng, nhà triết học sẽ chỉ bắt đầu ăn sau khi kiểm tra thấy cả hai nĩa cạnh mình đều đang không được dùng. Tuy nhiên do giới hạn thời gian tìm hiểu vấn đề, báo cáo này sẽ không thể trình bày thêm về các lời giải tốt hơn.

Tài liệu

[1] A. Silberschatz, P.B. Galvin, G. Gagne (2018) *Operating System Concepts*, John Wiley & Sons, Inc, 10th ed.

[2] *C560 Lecture notes – Dining Philosophers*, `https://web.archive.org/web/20120722014159/http://web.eecs.utk.edu/~plank/plank/classes/cs560/560/notes/Dphil/lecture.html`, 06/03/2022.