Bài 5
Các công cụ đồng bộ hóa (Synchronization Tools)

Nhu cầu đồng bộ hóa

- Các tiến trình cần có nhu cầu liên lạc với nhau để:
  - Chia sẻ thông tin.
  - Hợp tác hoàn thành tác vụ.
- Giải pháp liên lạc: (Interprocess Communication)
  - Shared memory
  - Message queue
  - Pipe
  - Socket

Nhu cầu đồng bộ hóa

- Nếu tiến trình có thể liên lạc với nhau, OS cần cung cấp cơ chế đồng bộ hóa để hoạt động của các tiến trình không tác động sai lệch đến nhau vì:
  - Yêu cầu độc quyền truy xuất (mutual exclusion)
    - ✓ Tài nguyên hệ thống gồm hai loại: có thể chia sẻ (nhiều P cùng truy xuất) và không thể chia sẻ (chỉ chấp nhận 1 hoặc một vài P truy xuất).
    - ✓ Mutual exclusion: OS phải kiểm soát sao cho tại một thời điểm chỉ có 1 (hoặc một số) P được truy xuất tài nguyên không thể chia sẻ.
  - Yêu cầu phối hợp (synchronization):
    - ✓ Tốc độ thực thi, thời gian ngắt, tần suất cấp phát CPU của các P khác nhau.
    - ✓ Synchronization: OS phải đảm bảo các P hoạt động nhịp nhàng với nhau. VD: P_B chỉ có thể xử lý nếu P_A đã hoàn tất một xử lý nào đó trước.

Race condition

- Giả sử có P_1 và P_2 cùng thực hiện công việc rút tiền, chúng sử dụng chung biến taikhoan, công việc của P_1 và P_2 như sau:

```
if (taikhoan - tienrut >=0)

    taikhoan = taikhoan - tienrut;
else
    error(« khong the rut tien ! »);
```

- Giả sử taikhoan=800, P_1 muốn rút 500 và P_2 muốn rút 400. Tình huống sau thường xảy ra:
  - P1 rút 500 → taikhoan – tienrut = 800 – 500 > 0, P1 được phép rút tiền.
  - P2 rút 400 → taikhoan – tienrut = 300 – 400 <0, P2 không được phép rút tiền.
  - Trường hợp tương tự nếu P2 rút trước P1.

Race condition

- Giả sử có P_1 và P_2 cùng thực hiện công việc rút tiền, chúng sử dụng chung biến taikhoan, công việc của P_1 và P_2 như sau:

```
if (taikhoan - tienrut >=0)

    taikhoan = taikhoan - tienrut;
else
    error(« khong the rut tien ! »);
```

- Tuy nhiên, tình huống sau có thể xảy ra
  - P1 kiểm tra điều kiện hợp lệ, chuẩn bị rút tiền thì bị OS ngắt.
  - P2 kiểm tra điều kiện hợp lệ, rút tiền thành công 400 → taikhoan = 800-400 = 400
  - P1 được tiếp tục running, thực thi đoạn code rút tiền taikhoan = 400 – 500 = -100
- Lỗi do đâu?

Race condition

- Race condition (tranh đoạt điều khiển): là trường hợp khi có hai hay nhiều tiến trình cùng thao tác trên một tài nguyên dùng chung không chia sẻ, kết quả phụ thuộc vào sự điều phối tiến trình của hệ thống.
- Giải quyết: áp đặt truy xuất độc quyền (mutual exclusion) – khi một P đang sử dụng tài nguyên thì các P khác không được phép truy xuất tài nguyên đó.
- Đoạn chương trình có chứa khả năng xảy ra mâu thuẫn truy xuất tài nguyên dùng chung gọi là critical section (miền găng)

```
if (taikhoan - tienrut >=0)

    taikhoan = taikhoan - tienrut;
```

Critical section

- Giải quyết critical section (CS – miền găng) phải thỏa 4 điều kiện:
  - Không có hai P cùng ở trong CS cùng lúc.
  - Không có giả thiết nào đặt ra cho sự liên hệ về tốc độ của các P cũng như về số lượng bộ xử lý trong hệ thống (ít đề cập đến).
  - Một P tạm dừng ngoài CS không được ngăn cản P khác vào CS
  - Không có P nào phải chờ vô hạn để được vào CS.

Các giải pháp đồng bộ hóa

- Giải pháp đồng bộ hóa: giải pháp đưa ra để truy xuất CS, gồm 2 cách tiếp cận:
  - Busy waiting: không cần sự hỗ trợ của OS
    - ✓ Phần mềm:
      - Đặt cờ hiệu
      - Biến luân phiên
      - Peterson
    - ✓ Phần cứng:
      - Cấm ngắt
      - TSL
  - Sleep & wakeup: cần sự hỗ trợ của OS
    - ✓ Semaphore
    - ✓ Monitor
    - ✓ Message

Các giải pháp đồng bộ hóa Busy waiting

- Sử dụng phần mềm:
  - Dùng các biến cờ hiệu: các P chia sẻ một biến chung đóng vai trò cờ hiệu (hay lock), khởi tạo = 0. Một P muốn vào CS phải kiểm tra biến lock.
    - ✓ Lock = 0: P đặt lock=1 và vào CS.
    - ✓ Lock = 1: P chờ ngoài CS cho đến khi lock=0.
    - ✓ Như vậy: lock=0 tương ứng không có P trong CS.

```
1 while (TRUE) {
2    while (lock == 1); // wait
3    lock = 1;
4    critical-section ();
5    lock = 0;
6    Noncritical-section ();
7 }
```

→ Giải pháp có khả năng vi phạm điều kiện 1: có 2 P ở trong CS cùng 1 thời điểm (xảy ra tương tự bài toán rút tiền).

- Sử dụng phần mềm:
  - Kiểm tra luân phiên: áp dụng với 2 P. Hai P sử dụng chung biến turn (cho biết tới phiên P nào vào CS), turn khởi tạo = 0.
    - ✓ Turn=0: P0 vào CS. Khi P0 rời CS, nó gán lại turn=1 để P1 có thể vào CS.
    - ✓ Turn=1: P0 chạy vòng lặp chờ cho đến khi turn=0.

```
while (TRUE) {

     while (turn != 0); // wait
     critical-section ();
     turn = 1;
     Noncritical-section ();

}
```

```
while (TRUE) {

     while (turn != 1); // wait
     critical-section ();
     turn = 0;
     Noncritical-section ();

}
```

→ Giải pháp ngăn chặn được 2 P cùng vào CS nhưng lại vi phạm điều kiện 3. Nếu P0 muốn vào CS 2 lần liên tiếp? Ngoài ra nếu P1 xử lý chậm hơn nhiều so với P0 thì giải pháp này vi phạm điều kiện 2.

Các giải pháp đồng bộ hóa Busy waiting

- Sử dụng phần mềm:
  - Peterson: kết hợp cả hai giải pháp trên, các P chia sẻ hai biến chung.

```
int turn; // đến phiên ai

int interesse[2]; // khởi động là FALSE
```

```
while (TRUE) {

    int j = 1-i; // j là tiến trình còn lại
    interesse[i]= TRUE;
    turn = j;
    while (turn == j && interesse[j]==TRUE);
    critical-section ();
    interesse[i] = FALSE;
    Noncritical-section ();

}
```

→ Giải pháp ngăn chặn được mâu thuẫn truy xuất: mỗi tiến trình Pi chỉ có thể vào CS khi interesse[j]=FALSE hoặc turn=i. Nếu cả hai P đều muốn vào CS thì interesse[i]=interesse[j]=TRUE nhưng turn chỉ mang giá trị 0 hoặc 1 nên chỉ có 1 P vào CS.

Các giải pháp đồng bộ hóa Busy waiting

- Giải pháp phần cứng:
  - Cấm ngắt:
    - ✓ Cho phép P cấm tất cả các ngắt (interrupt) trước khi vào CS và phục hồi ngắt khi ra khỏi CS → Hệ thống không thể tạm dừng hoạt động của một P để cấp CPU cho P khác.
    - ✓ Giải pháp này không phổ biến bởi:
      - Cho phép P can thiệp ngắt.
      - Nếu hệ thống có nhiều CPU → giải pháp không hiệu quả.
  - Chỉ thị TSL (test and set):
    - ✓ Nhờ sự hỗ trợ của phần cứng, cung cấp mọt chỉ thị đặc biệt cho phép kiểm tra và cập nhật nội dung một vùng nhớ trong một thao tác không thể phân chia, gọi là chỉ thị TSL.
    - ✓ Nếu có hai chỉ thị TSL xử lý đồng thời, chúng sẽ được xử lý tuần tự.
    - ✓ Giải pháp này khó cài đặt nhất là đối với máy có nhiều CPU.

Các giải pháp đồng bộ hóa Sleep & Wakeup

- P chưa đủ điều kiện vào CS chuyển sang trạng thái blocked, từ bỏ quyền dùng CPU.
- OS cung cấp các function để thực hiện hai thao tác mà P cần: sleep và wakeup
  - Sleep: lời gọi hệ thống tạm dừng hoạt động của P (blocked) và chờ cho đến khi được một P khác "đánh thức".
  - Wakeup: "đánh thức" một P đang sleep trong hàng đợi.
- Ý tưởng của sleep&wakeup:
  - Một P chưa đủ điều kiện vào CS sẽ tự gọi sleep để tự block cho đến khi có P khác gọi wakeup để giải phóng.
  - Một P gọi wakeup khi ra khỏi CS để đánh thức một P đang sleep, cho nó vào CS.

Các giải pháp đồng bộ hóa Sleep & Wakeup

- Cần lưu ý tình trạng mâu thuẫn khi sử dụng sleep&wakeup:
  - A vào CS, B được kích hoạt.
  - B thử vào CS nhưng thấy A đang trong CS, B tăng blocked và chuẩn bị gọi sleep() để tự khóa chính nó.
  - Ngay lúc đó, A rời khỏi CS, thấy có tiến trình đang blocked nên gọi wakeup() và giảm blocked.
  - Lúc đó, tín hiệu wakeup() bị thất lạc do B chưa "thật sự ngủ".
  - → B bị khóa vĩnh viễn

```
int busy;    // 1 nếu miền găng đang bị chiếm, nếu không là 0
int blocked;    // đếm số lượng tiến trình đang bị khóa

while (TRUE) {

    if (busy){
        blocked = blocked + 1;
        sleep();
    }
    else busy = 1;

    critical-section ();

    busy = 0;
    if(blocked){
        wakeup(process);
        blocked = blocked - 1;
    }

    Noncritical-section ();

}
```

Các giải pháp đồng bộ hóa Sleep & Wakeup

- Nguyên nhân lỗi: kiểm tra tư cách vào CS và gọi sleep&wakeup là những hành động tách biệt, có thể bị ngắt.
- Do đó, OS đưa ra các cơ chế đồng bộ hóa an toàn hơn, dựa trên ý tưởng sleep&wakeup, có kiểm tra điều kiện vào CS:
  - Semaphore
  - Monitor
  - Trao đổi thông điệp

Các giải pháp đồng bộ hóa Sleep & Wakeup - Semaphore

- Được Dijkstra đề xuất năm 1965, một semaphore s là một biến có:
  - Giá trị nguyên dương e(s)
  - Một hàng đợi f(s) lưu danh sách các P đang bị khóa/chờ trên semaphore s
  - Chỉ có 2 thao tác primitive được định nghĩa trên semaphore:
    - ✓ Down(s)/P(s)/Wait(s): giảm giá trị semaphore s 1 đơn vị nếu e(s)>0 và tiếp tục xử lý. Ngược lại, P phải chờ đến khi e(s)>0.
    - ✓ Up(s)/V(s)/Signal(s): tăng giá trị semaphore s 1 đơn vị. Nếu có một/nhiều P đang chờ trên semaphore s (bị khóa bởi thao tác Down) thì hệ thống chọn một trong các P để kết thúc thao tác Down và cho tiếp tục xử lý

Các giải pháp đồng bộ hóa Sleep & Wakeup - Semaphore

- Minh họa cài đặt semaphore

```
Down(s):
    e(s) = e(s) - 1;
    if e(s) < 0 {
        status(P)= blocked;
        enter(P,f(s));
        }
```

```
Up(s):
    e(s) = e(s) + 1;
    if s ≤ 0 {
        exit(Q,f(s)); //Q là tiến trình đang chờ trên s
        status (Q) = ready;
        enter(Q,ready-list);
        }
```

- Sử dụng semaphore giải quyết vấn đề truy xuất độc quyền: nhiều P cùng truy xuất đến CS mà không gây mâu thuẫn dữ liệu.
  - n tiến trình cùng sử dụng 1 semaphore s.
  - e(s) được khởi tạo = 1

```
1 while (TRUE) {
2     Down(s)
3     critical-section ();
4     Up(s)
5     Noncritical-section ();
6 }
```

Các giải pháp đồng bộ hóa Sleep & Wakeup - Semaphore

- Sử dụng semaphore giải quyết vấn đề đồng bộ hóa: một P phải đợi một P khác hoàn tất thao tác nào đó mới tiếp tục hoạt động.
  - Hai P chia sẻ một semaphore s.
  - e(s) được khởi tạo = 0

```
P1:

while (TRUE) {
   job1();
   Up(s); //đánh thức P2
}
```

```
P2:

while (TRUE) {
   Down(s); // chờ P1
   job2();
}
```

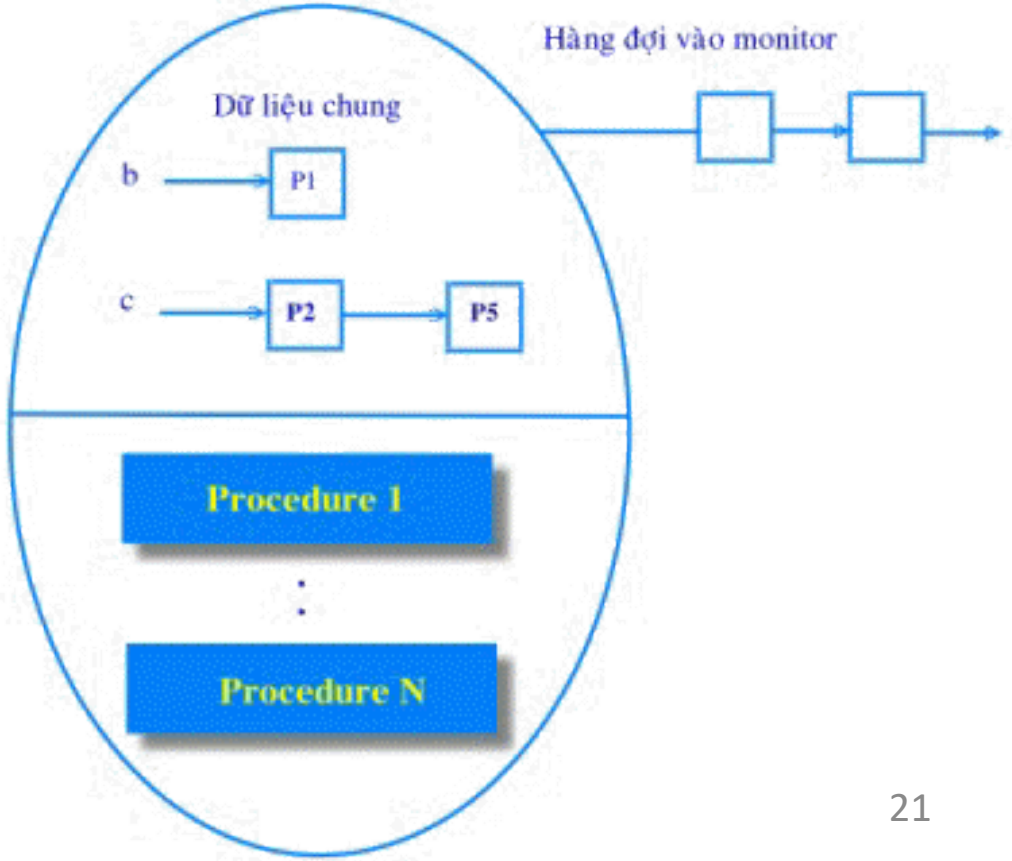
- Semaphore giải quyết vấn đề tín hiệu "đánh thức" bị thất lạc. Nhưng nếu đặt primitive Down & Up sai vị trí, thứ tự thì P sẽ bị khóa vĩnh viễn.

```
while (TRUE) {
   Down(s);
   critical-section ();
   Noncritical-section ();
   // Quên gọi Up(s);
}
```

Các giải pháp đồng bộ hóa Sleep & Wakeup - Monitor

- Monitor: Hoare(1974) và Brinch & Hansen (1975) đề nghị một cơ chế cao hơn được cung cấp bởi ngôn ngữ lập trình, để dễ viết đúng các chương trình đồng bộ hóa hơn.
- Monitor là một cấu trúc đặc biệt bao gồm các thủ tục, các biến và cấu trúc dữ liệu có các thuộc tính sau:
  - Các biến và cấu trúc dữ liệu bên trong monitor chỉ có thể được thao tác bởi các thủ tục định nghĩa bên trong monitor đó. (encapsulation).
  - Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor (mutual exclusive).
  - Trong một monitor, có thể định nghĩa các biến điều kiện và hai thao tác kèm theo là Wait và Signal

- Gọi c là biến điều kiện được định nghĩa trong monitor:
  - Wait(c): chuyển trạng thái tiến trình gọi sang blocked , và đặt tiến trình này vào hàng đợi trên biến điều kiện c.
  - Signal(c): nếu có một tiến trình đang bị khóa trong hàng đợi của c, tái kích hoạt tiến trình đó, và tiến trình gọi sẽ rời khỏi monitor.



Các giải pháp đồng bộ hóa Sleep & Wakeup - Monitor

- Trình biên dịch chịu trách nhiệm thực hiện việc truy xuất độc quyền đến dữ liệu trong monitor.
- Để thực hiện điều này, một semaphore nhị phân thường được sử dụng.
  - Mỗi monitor có một hàng đợi toàn cục lưu các tiến trình đang chờ được vào monitor
  - Mỗi biến điều kiện c cũng gắn với một hàng đợi f(c) và hai thao tác trên đó được định nghĩa như sau:

```
•Wait(c) :

    status(P)= blocked;
    enter(P,f(c));

Signal(c) :

    if (f(c) != NULL){

        exit(Q,f(c)); //Q là tiến trình chờ trên c
        statusQ) = ready;
        enter(Q,ready-list);

    }
```

- Cấu trúc một monitor và cấu trúc tiến trình trong giải pháp monitor

```
monitor <tên monitor >
condition <danh sách các biến điều kiện>;
<déclaration de variables>;

  procedure Action1();
  {

  }

              ....

  procedure Actionn();
  {

  }
end monitor;
```

```
while (TRUE) {

    Noncritical-section ();
    <monitor>.Actioni; //critical-section();
    Noncritical-section ();

}
```

→ Việc truy xuất độc quyền được đảm bảo bởi trình biên dịch, không phải do coder → giảm nguy cơ thực hiện đồng bộ hóa sai (cần NNLT định nghĩa monitor).

Các giải pháp đồng bộ hóa Sleep & Wakeup – Trao đổi thông điệp

- Dựa trên cơ sở trao đổi thông điệp với hai primitive Send và Receive để thực hiện sự đồng bộ hóa:
  - send(destination, message): gửi thông điệp đến P.
  - receive(source, message): nhận thông điệp từ 1 P hoặc bất kỳ P nào.
- P có yêu cầu tài nguyên sẽ gởi một thông điệp đến P kiểm soát và sau đó chuyển sang trạng thái blocked cho đến khi nhận được một thông điệp chấp nhận
- Khi sử dụng xong tài nguyên, P gởi một thông điệp khác đến P kiểm soát để báo kết thúc truy xuất.
- Về phía P kiểm soát, khi nhận được thông điệp yêu cầu tài nguyên, nó sẽ chờ đến khi tài nguyên sẵn sàng để cấp phát thì gởi một thông điệp đến P đang bị khóa trên tài nguyên đó để đánh thức tiến trình này.

Các giải pháp đồng bộ hóa Sleep & Wakeup – Trao đổi thông điệp

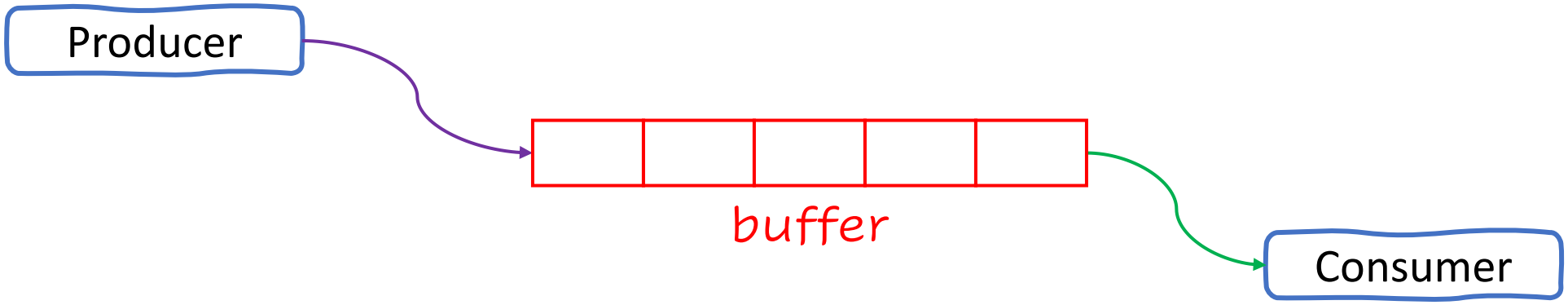
- Cấu trúc P yêu cầu tài nguyên, P sẽ bị block cho đến khi nhận được thông điệp chấp nhận cho truy xuất tài nguyên.

```
while (TRUE) {
    Send(process controler, request message);
    Receive(process controler, accept message);
    critical-section ();
    Send(process controler, end message);
    Noncritical-section ();
}
```

Nhận xét

- Semaphore và Monitor giải quyết được vấn đề truy xuất độc quyền trên các máy tính có một hoặc nhiều bộ xử lý chia sẻ một vùng nhớ chung.
- Nhưng các primitive không hữu dụng trong các hệ thống phân tán.
- Trong những hệ thống phân tán như thế, cơ chế trao đổi thông điệp tỏ ra hữu hiệu và được dùng để giải quyết bài toán đồng bộ hóa.

- Bài toán người sản xuất – tiêu thụ (Producer vs Consumer)
  - Vấn đề: hai tiến trình cùng chia sẻ một bộ đệm có kích thước giới hạn. Một trong hai tiến trình đóng vai trò người sản xuất – tạo ra dữ liệu và đặt dữ liệu vào bộ đệm- và tiến trình kia đóng vai trò người tiêu thụ – lấy dữ liệu từ bộ đệm ra để xử lý.
  - Để đồng bộ hóa hoạt động của hai tiến trình sản xuất tiêu thụ, cần tuân thủ:
    - ✓ P sản xuất (producer) không được ghi dữ liệu vào bộ đệm đã đầy (synchronisation)
    - ✓ P tiêu thụ (consumer) không được đọc dữ liệu từ bộ đệm đang trống (synchronisation)
    - ✓ Hai P sản xuất và tiêu thụ không được thao tác trên bộ đệm cùng lúc (mutual exclusion)



Các bài toán đồng bộ hóa

- Bài toán người sản xuất – tiêu thụ: sử dụng 3 semaphore
  - full: đếm số chỗ đã có dữ liệu trong bộ đệm
  - empty: đếm số chỗ còn trống trong bộ đệm
  - mutex: kiểm tra việc Producer và Consumer không truy xuất đồng thời đến bộ đệm

```
1 BufferSize = 3; // số chỗ trong bộ đệm
2 semaphore mutex = 1; // kiểm soát truy xuất độc quyền
3 semaphore empty = BufferSize; // số chỗ trống
4 semaphore full = 0; // số chỗ đầy
5
6 Producer() {
7    int item;
8    while (TRUE) {
9       produce_item( & item); // tạo dữ liệu mới
10      down( & empty); // giảm số chỗ trống
11      down( & mutex); // báo   hiệu vào miền găng
12      enter_item(item); // đặt dữ liệu vào bộ đệm
13      up( & mutex); // ra khỏi miền găng
14      up( & full); // tăng số chỗ đầy
15   }
16 }
17
18 Consumer() {
19   int item;
20
21   while (TRUE) {
22      down( & full); // giảm số chỗ đầy
23      down( & mutex); // báo   hiệu vào miền găng
24      remove_item( & item); // lấy dữ liệu từ bộ đệm
25      up( & mutex); // ra khỏi miền găng
26      up( & empty); // tăng số chỗ trống
27      consume_item(item); // xử lý dữ liệu
28   }
29 }
```

- Bài toán người sản xuất – tiêu thụ: sử dụng monitor:
  - Monitor ProducerConsumer có hai method enter và remove thao tác trên bộ đệm.
  - Xử lý của hai method trên phụ thuộc vào hai biến điều kiện full và empty.

```
23
24 Producer(); {
25     while (TRUE) {
26         produce_item( & item);
27         ProducerConsumer.enter;
28     }
29 }
30 Consumer(); {
31     while (TRUE) {
32         ProducerConsumer.remove;
33         consume_item(item);
34     }
35 }
```

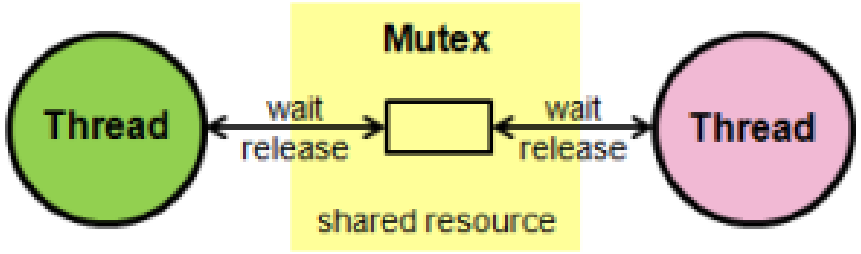
```
1 monitor ProducerConsumer
2 condition full, empty;
3 int count;
4
5 procedure enter(); {
6     if (count == N)
7         wait(full); // nếu bộ đệm  đầy, phải chờ
8     enter_item(item); // đặt dữ liệu vào bộ đệm
9     count++; // tăng số chỗ đầy
10    if (count == 1) // nếu bộ đệm không trống
11        signal(empty); // thì kích hoạt Consumer
12 }
13 procedure remove(); {
14    if (count == 0)
15        wait(empty) // nếu bộ đệm  trống, chờ
16    remove_item( & item); // lấy dữ liệu từ bộ đệm
17    count--; // giảm số chỗ đầy
18    if (count == N - 1) // nếu bộ đệm không đầy
19        signal(full); // thì kích hoạt Producer
20 }
21 count = 0;
22 end monitor;
```

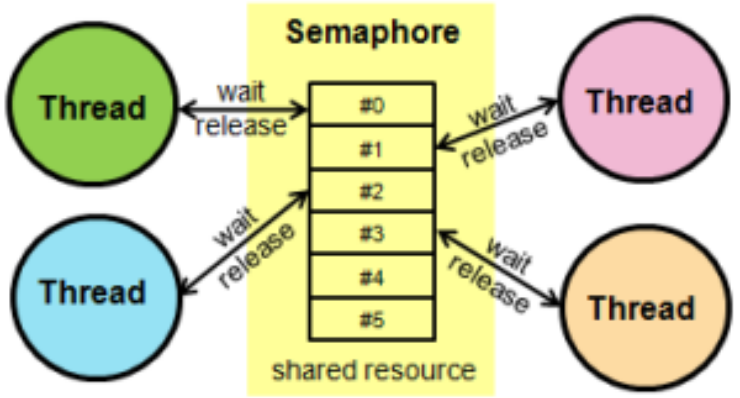
Các bài toán đồng bộ hóa

- Bài toán người sản xuất – tiêu thụ: sử dụng message:
  - Ý nghĩa message empty: có một chỗ trống trong buffer.
  - Consumer bắt đầu bằng việc gửi 4 message empty → Producer
  - Producer tạo ra 1 dữ liệu mới và chờ đến khi nhận 1 empty thì gửi ngược lại cho Consumer.

```
1  BufferSize = 4;
2
3  Producer() {
4     int item;
5     message m; // thông điệp
6
7     while (TRUE) {
8        produce_item( & item);
9        receive(consumer, & m); //  chờ thông điệp empty
10       create_message( & m, item); // tạo thông điệp dữ liệu
11       send(consumer, & m); // gởi dữ liệu đến Consumer
12    }
13 }
14
15 Consumer() {
16    int item;
17    message m;
18
19    for (0 to N)
20       send(producer, & m); // gởi N thông điệp empty
21    while (TRUE) {
22       receive(producer, & m); // chờ thông điệp dữ liệu
23       remove_item( & m, & item); // lấy dữ liệu từ thông điệp
24       send(producer, & m); // gởi thông điệp empty
25       consumer_item(item); // xử lý dữ liệu
26    }
27 }
```

- Mutex (MUTual EXclusion): ngăn cản hai task cùng truy cập tài nguyên, một dạng tương tự binary semaphore nhưng tài nguyên bị khó bởi task nào thì mở bởi task đó.
- Semaphore: advanced mutex





- Đọc thêm:
  - https://www.geeksforgeeks.org/difference-between-binary-semaphore-and-mutex/
  - https://adithyaravik.wordpress.com/2015/03/31/semaphore-vs-mutex-vs-monitor/

Tham khảo

- https://codegym.cc/groups/posts/220-whats-the-difference-between-a-mutex-a-monitor-and-a-semaphore
- http://dembinhyen.free.fr/UDS/Ebook/CD1/He%20Dieu%20Hanh/Htm/Bai051.htm