Tên: Lê Anh Thư
MSSV: 20521985
Lớp: IT007.M21.HTCL

MÔN HỌC: HỆ ĐIỀU HÀNH
BÀI TẬP CHƯƠNG 5

Câu 1: Khi nào thì xảy ra tranh chấp?

Race condition là tình trạng nhiều process truy xuất và thao tác đồng thời lên dữ liệu chia sẻ. Kết quả cuối cùng của việc truy xuất đồng thời này phụ thuộc thứ tự thực thi của các lệnh thao tác dữ liệu.
Ví dụ điển hình về bounded buffer :

Producer	Consumer

while (true)
{
counter++;
//Tăng biến counter lên 1 đơn vị.
}
	while (true)
	{
	counter--;
	//Giảm biến counter đi 1 đơn vị.
	}

Code C++ demo Critical Section.

Cho rằng biến counter là biến được chia sẻ giữa Producer và Consumer (tức là, biến counter có thể được đọc và được ghi bởi cả Producer và Consumer). Và Producer và Consumer chạy đồng thời cùng lúc với nhau (Concurrent).

Vậy việc gì sẽ xảy ra ?

Thực ra, mã giả trên mà bạn thấy sẽ được biên dịch lại thành hợp ngữ. Và dù chỉ 1 dòng lệnh counter++ (hoặc counter--), thực tế mã hợp ngữ được biên dịch ra bao gồm 3 mã như sau :

Producer	Consumer
1. register1 = counter 2. register1 = register1 + 1 3. counter = register1	1. register2 = counter 2. register2 = register2 − 1 3. counter = register2

Hợp ngữ của việc tăng và giảm biến counter.

Như các bạn thấy, để tăng (hoặc giảm) biến counter cần đến những 3 lệnh. Việc đa chương của máy tính chẳng qua chỉ là thực hiện 1 tác vụ tại một thời điểm nhất định, nhưng việc chuyển qua lại tác vụ rất nhanh giữa chúng đã tạo ra một cảm giác cho người sử dụng là máy tính của chúng ta đang thực hiện nhiều công việc cùng 1 lúc.
Khi máy tính thực hiện một tác vụ, nó đang thực hiện các lệnh hợp ngữ trên. Và Scheduler có thể cắt (interrupt) một tác vụ bất kỳ ngay sau khi thực hiện xong dòng lệnh của hợp ngữ (VD sau khi thực hiện xong dòng 2, Scheduler ngắt không cho thực hiện tiếp lệnh 3 và chuyển qua process khác, sau đó mới quay lại thực hiện lệnh 3).
Vậy giả sử như ta có các thời điểm với các lệnh như sau (biến counter đang có giá trị là 5) :
*T*0: *producer* thực hiện dòng 1 : *register*1 = counter {*register*1 = 5}
*T*1: *producer* thực hiện dòng 2 : *register*1 = *register*1 + 1 {*register*1 = 6} *(ngắt)*
*T*2: ***consumer*** thực hiện dòng 1 : *register*2 = counter {*register*2 = 5} (Oh-No).
*T*3: ***consumer*** thực hiện dòng 2 : *register*2 = *register*2 − 1 {*register*2 = 4} *(ngắt)*
*T*4: *producer* thực hiện dòng 3 : counter = *register*1 {*counter* = 6} *(ngắt)*
*T*5: ***consumer*** thực hiện dòng 3 : counter = *register*2 {*counter* = 4} (Oh-No).
Kết quả trả ra không đúng như ta mong muốn. Producer chạy 1 lần (counter++), Consumer chạy 1 lần (counter--) thì lẽ ra kết quả của chúng ta phải là **counter=5**. Thế mà kết quả cuối cùng của chúng ta đã trở thành **4** (kết quả không chính xác).
Ngược lại, nếu chúng ta đổi chỗ T4 và T5 với nhau, chúng ta sẽ được **counter = 6**.
Vậy dữ liệu của chúng ta không được nhất quán.
Để dữ liệu chia sẻ được nhất quán, cần bảo đảm sao cho tại mỗi thời điểm chỉ có một process được thao tác lên dữ liệu chia sẻ. Do đó, cần có cơ chế đồng bộ hoạt động của các process này.

Câu 2: Vấn đề vùng tranh chấp (critical section) là gì?

Đầu tiên, chúng ta hãy xét một hệ thống có n tiến trình (tạm đặt tên của n tiến trình này là {P0, P1, ..., Pn−1}). Từng tiến trình đều có một đoạn mã, gọi là **critical section (CS),** tên tiếng Việt là **vùng tranh chấp.** Trong CS, các đoạn mã thao tác lên dữ liệu chia sẻ giữa các tiến trình.

Một đặc tính quan trọng mà chúng ta cần quan tâm, đó chính là khi process P0 đang chạy đoạn mã bên trong CS thì không một process nào khác được chạy đoạn mã bên trong CS

(để đảm bảo cho dữ liệu được nhất quán). Hay nói cách khác là 1 CS trong một thời điểm nhất định, chỉ có 1 process được phép chạy.

Và **vấn đề vùng tranh chấp (Critical Section Problem)** là vấn đề về việc tìm một cách thiết kế một giao thức (một cách thức) nào đó để các process có thể phối hợp với nhau hoàn thành nhiệm vụ của nó.

Câu 3: Có những yêu cầu nào dành cho lời giải của bài toán vùng tranh chấp?

Một lời giải cho vấn đề vùng tranh chấp (CS Problem) phải đảm bảo được 3 tính chất sau :

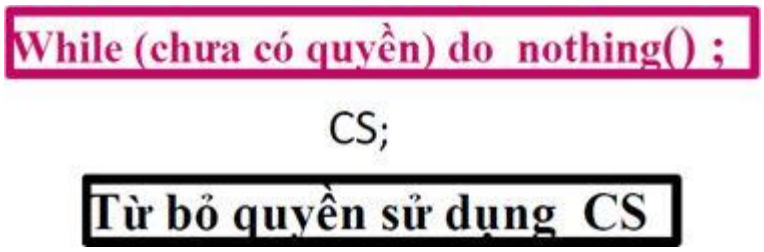
- Loại trừ tương hỗ (Mutual Exclusion) : Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.
- Phát triển (Progress) : Một tiến trình tạm dừng bên ngoài CS không được ngăn cản các tiến trình khác vào CS.
- Chờ đợi giới hạn (Bounded Waiting) : Mỗi process chỉ phải chờ để được vào CS trong một khoảng thời gian có hạn (finite wait time). Không được xảy ra tình trạng đói tài nguyên (starvation).

Câu 4: Có mấy loại giải pháp đồng bộ? Kể tên và trình bày đặc điểm của các loại giải pháp đó?

Có 2 nhóm giải pháp chính :

Nhóm giải pháp Busy Waiting :

- Tính chất :
    - Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng (thông qua việc kiểm tra điều kiện vào CS liên tục).
    - Không đòi hỏi sự trợ giúp của hệ điều hành.
- Cơ chế chung :

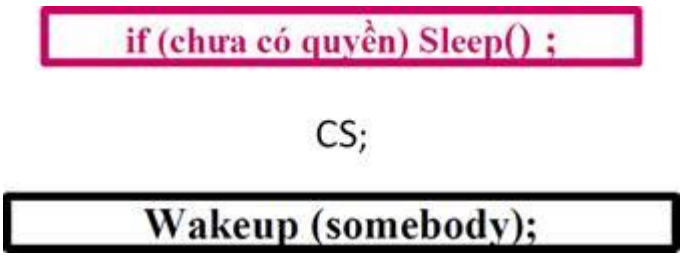


Cơ chế chung của nhóm giải pháp Busy Waiting.

- Bao gồm một vài loại :
  - Sử dụng các biến cờ hiệu.
  - Sử dụng việc kiểm tra luân phiên.
  - Giải pháp của Peterson.
  - Cấm ngắt (giải pháp phần cứng – hardware).
  - Chỉ thị TSL (giải pháp phần cứng – hardware).

Nhóm giải pháp Sleep & Wakeup.

- Tính chất :
  - Từ bỏ CPU khi chưa được vào CS.
  - Cần sự hỗ trợ từ hệ điều hành (để đánh thức process và đưa process vào trạng thái blocked).
- Cơ chế chung :



Cơ chế chung của nhóm giải pháp Sleep & Wakeup.

- Bao gồm một vài loại :
  - Semaphore.
  - Monitor.

- Message.

Câu 5: Phân tích và đánh giá ưu, nhược điểm của các giải pháp đồng bộ busy waiting (cả phần cứng và phần mềm)?

Các giải pháp phần mềm

Sử dụng giải thuật kiểm tra luân phiên

- Sử dụng các biến cờ hiệu
- Giải pháp của Peterson
- Giải pháp Bakery

Các giải pháp phần cứng

- Cấp ngắt
- Chỉ thị TSL

Giải thuật 1

- Biến chia sẻ
  - int turn; /* khởi đầu turn = 0 */
  - nếu turn = i thì Pi được phép vào critical section, với i = 0 hay 1
- Process Pi

```
do {
    while (turn != i);
        critical section
    turn = j;
        remainder section
} while (1);
```

- Thỏa mãn Mutual exclusion (1)
- Nhưng không thoả mãn yêu cầu về progress (2) và bounded waiting (3) vì tính chất strict alternation của giải thuật

```
Process P0:
do
 while (turn != 0);
        critical section
 turn := 1;
        remainder section
while (1);
```

```
Process P1:
do
 while (turn != 1);
        critical section
 turn := 0;
        remainder section
while (1);
```

- **Điều gì xảy ra nếu P0 có RS (remainder section) rất lớn còn P1 có RS nhỏ?**

Giải thuật 2

- Biến chia sẻ
  - boolean flag[2]; /* khởi đầu flag[0] = flag[1] = false */
  - Nếu flag[i] = true thì Pi "sẵn sàng" vào critical section.
- Process Pi

```
do {
    flag[ i ] = true;       /* Pi "sẵn sàng" vào CS */
    while ( flag[ j ] );    /* Pi "nhường" Pj       */
        critical section
    flag[ i ] = false;
        remainder section
} while (1);
```

- Thỏa mãn Mutual exclusion (1)
- Không thỏa mãn progress. Vì sao?

Giải thuật 3 (Peterson)

- Biến chia sẻ
  - Kết hợp cả giải thuật 1 và 2
- Process Pi, với i = 0 hoặc i = 1

```
do {
```

```
flag[ i ] = true;       /*  Process i sẵn sàng  */
turn = j;               /*  Nhường process j    */
while (flag[ j ] and turn == j);

        critical section

flag[ i ] = false;

        remainder section

} while (1);
```

- Thoả mãn được cả 3 yêu cầu ?

⇒ giải quyết bài toán critical section cho 2 process

Giải thuật 3 (Peterson) cho 2 tiến trình

Process P_0

```
do {
  /*  0 wants in  */
  flag[0] = true;
  /*  0 gives a chance to 1  */
  turn = 1;
  while (flag[1] &&turn == 1);
    critical section
  /*  0 no longer wants in  */
  flag[0] = false;
    remainder section
} while(1);
```

Process P_1

```
do {
  /*  1 wants in  */
  flag[1] = true;
  /*  1 gives a chance to 0  */
  turn = 0;
  while (flag[0] && turn == 0);
    critical section
  /*  1 no longer wants in  */
  flag[1] = false;
    remainder section
} while(1);
```

- Giải thuật 3 thỏa mutual exclusion, progress, và bounded waiting
- Mutual exclusion được đảm bảo bởi vì
  - P0 và P1 đều ở trong CS nếu và chỉ nếu flag[0] = flag[1] = true và turn = I cho mỗi Pi (không thể xảy ra)
- Chứng minh thỏa yêu cầu về progress và bounded waiting
  - Pi không thể vào CS nếu và chỉ nếu bị kẹt tại vòng lặp while() với điều kiện flag[j] = true và turn = j

- Nếu Pj không muốn vào CS thì flag[j] = false và do đó Pi có thể vào CS
- Nếu Pj đã bật flag[j] = true và đang chờ tại while() thì có chỉ hai trường hợp là turn = i hoặc turn = j
- Nếu turn = i và Pi vào CS. Nếu turn = j thì Pj vào CS nhưng sẽ bật flag[j] = false khi thoát ra -> cho phếp Pi và CS
- Nhưng nếu Pj có đủ thời gian bật flag[j] = true thì Pj cũng phải gán turn = i
- Vì Pi không thay đổi trị của biến turn khi đang kẹt trong vòng lặp while(), Pi sẽ chờ để vào CS nhiều nhất là sau một lần Pj vào CS (bounded waiting)

- Trước khi vào CS, process Pi nhận một con số. Process nào giữ con số nhỏ nhất thì được vào CS
- Trường hợp Pi và Pj cùng nhận được một chỉ số:
  - Nếu i < j thì Pi được vào trước. (Đối xứng)
- Khi ra khỏi CS, Pi đặt lại số của mình bằng 0
- Cơ chế cấp số cho các process thường tạo các số theo cơ chế tăng dần, ví dụ 1, 2, 3, 3, 3, 3, 4, 5,…
- Kí hiệu
  - (a,b) < (c,d) nếu a < c hoặc nếu a = c và b < d
  - max(a0,…,ak) là con số b sao cho b ≥ ai với mọi i = 0,…, k

```
/*  shared variable  */
boolean     choosing[ n ]; /*  initially, choosing[ i ] = false  */
int         num[ n ];      /*  initially, num[ i ] = 0           */

do {
       choosing[ i ] = true;
       num[ i ]      = max(num[0], num[1],…, num[n □ 1]) + 1;
       choosing[ i ] = false;
       for (j = 0; j < n; j++) {
           while (choosing[ j ]);
           while ((num[ j ] != 0) && (num[ j ], j) < (num[ i ], i));
       }
          critical section
       num[ i ] = 0;
          remainder section
} while (1);
```

- Khuyết điểm của các giải pháp software:
  - Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (busy waiting), tốn nhiều thời gian xử lý của CPU
  - Nếu thời gian xử lý trong vùng tranh chấp lớn, một giải pháp hiệu quả nên có cơ chế block các process cần đợi.

- Các giải pháp phần cứng:
  - Cấm ngắt (disable interrupts)
  - Dùng các lệnh đặc biệt
- Trong hệ thống uniprocessor: mutual exclusion được đảm bảo
  - Nhưng nếu system clock được cập nhật do interrupt thì…
- Trong hệ thống multiprocessor: mutual exclusion không được đảm bảo
  - Chỉ cấm ngắt tại CPU thực thi lệnh disable_interrupts
  - Các CPU khác vẫn có thể truy cập bộ nhớ chia sẻ

- Mutual exclusion được bảo đảm: nếu Pi vào CS, các process Pj khác đều đang busy waiting
- Khi Pi ra khỏi CS, quá trình chọn lựa process Pj vào CS kế tiếp là tùy ý ⇒ không bảo đảm điều kiện bounded waiting. Do đó có thể xảy ra starvation (bị bỏ đói)
- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh đơn là Swap(a, b) có tác dụng hoán chuyển nội dung của a và b.
  - Swap(a, b) cũng có ưu nhược điểm như TestAndSet

- Biến chia sẻ **lock** được khởi tạo giá trị false
- Mỗi process P_i có biến cục bộ **key**
- Process P_i nào thấy giá trị **lock** = **false** thì được vào CS.
  - Process P_i sẽ loại trừ các process P_j khác khi thiết lập **lock = true**

```
void Swap(boolean *a,
          boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

- Biến chia sẻ (khởi tạo là **false**)

```
bool lock;
bool key;
```

- Process P_i

```
do {
    key = true;
    while (key == true)
        Swap(&lock, &key);
    critical section
    lock = false;
    remainder section
} while (1)
```

Không thỏa mãn bounded waiting

- Cấu trúc dữ liệu dùng chung (khởi tạo là false)

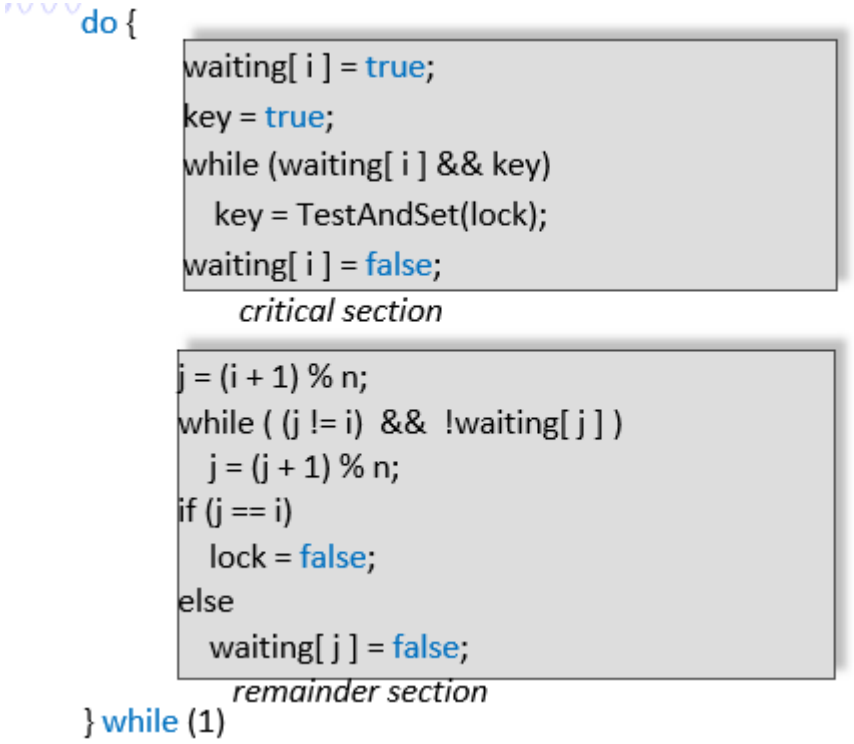
  bool waiting[n];

  bool lock;

- Mutual exclusion: Pi chỉ có thể vào CS nếu và chỉ nếu hoặc waiting[i] = false, hoặc key = false
  - key = false chỉ khi TestAndSet (hay Swap) được thực thi
    - Process đầu tiên thực thi TestAndSet mới có key == false; các process khác đều phải đợi
  - waiting[i] = false chỉ khi process khác rời khỏi CS

- Chỉ có một waiting[i] có giá trị false

- Progress: chứng minh tương tự như mutual exclusion
- Bounded waiting: waiting in the cyclic order



Câu 6: Semaphore là gì? Đặc điểm của semaphore? Cách thức hiện thực semaphore? Có mấy loại semaphore? Khi sử dụng semaphore cần lưu ý những vấn đề gì?

Semaphore là một công cụ đồng bộ cung cấp bởi OS mà không đòi hỏi Busy Waiting.

Đặc điểm của semaphore:

- Semaphore S là một biến số nguyên.
- Ngoài thao tác khởi động biến thì chỉ có thể được truy xuất qua hai tác vụ cố tính đơn nguyên (atomic) và loại trừ (mutual exclusive)
  - wait(S) hay còn gọi là P(S): giảm giá trị semaphore (S=S-1) . Kế đó nếu giá trị này âm thì process thực hiện lệnh wait() bị blocked.

- signal(S) hay còn gọi là V(S): tăng giá trị semaphore (S=S+1) . Kế đó nếu giá trị này không dương, một process đang blocked bởi một lệnh wait() sẽ được hồi phục để thực thi.

- Tránh busy waiting: khi phải đợi thì process sẽ được đặt vào một blocked queue, trong đó chứa các process đang chờ đợi cùng một sự kiện.
- P(S) hay wait(S) sử dụng để giành tài nguyên và giảm biến đếm S=S-1
- V(S) hay signal(S) sẽ giải phóng tài nguyên và tăng biến đếm S= S+1
- Nếu P được thực hiện trên biến đếm <= 0 , tiến trình phải đợi V hay chờ đợi sự giải phóng tài nguyên

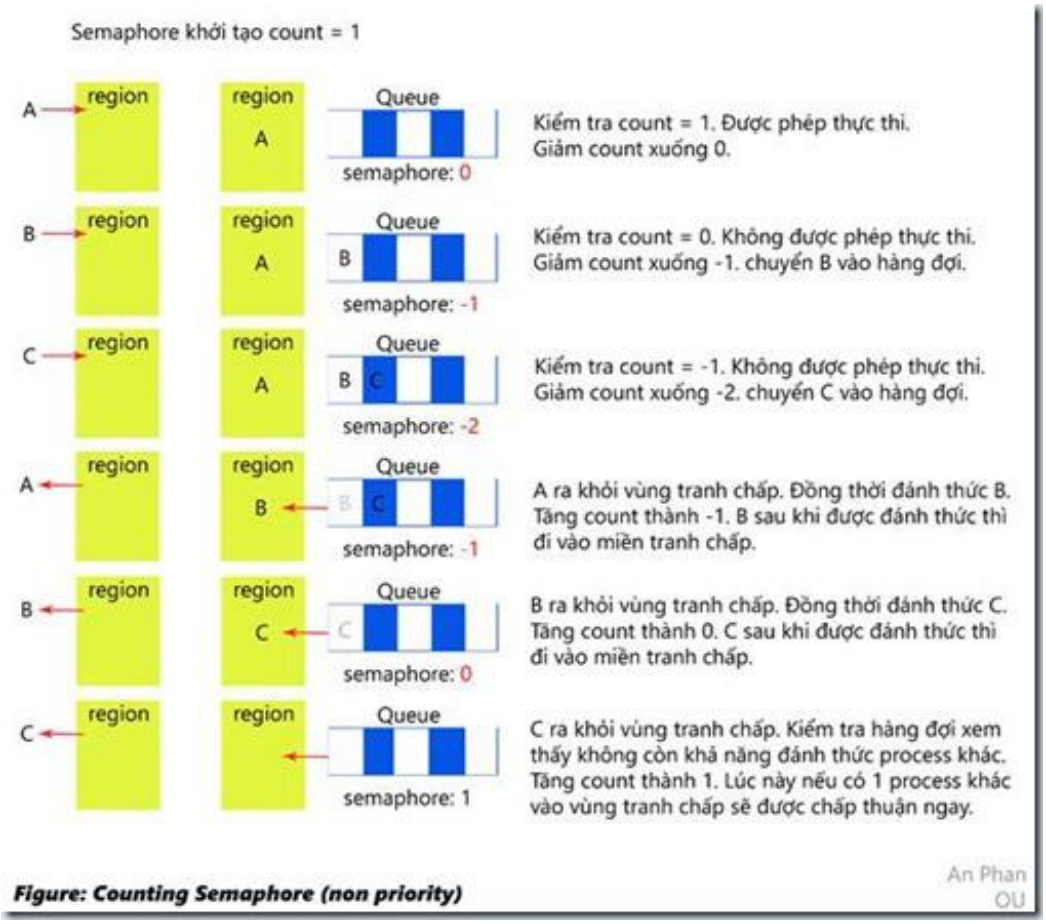


Figure: Counting Semaphore (non priority)

- Định nghĩa semaphore là một record

```
typedef  struct {
        int  value;
```

```
        struct  process *L; /* process queue */
    } semaphore;
```

- Giả sử hệ điều hành cung cấp hai tác vụ (system call):
  - block(): tạm treo process nào thực thi lệnh này
  - wakeup(P): hồi phục quá trình thực thi của process P đang blocked
- Các tác vụ semaphore được hiện thực như sau

```
void wait(semaphore S) {
        S.value--;
            if (S.value < 0) {
                    add this process to S.L;
                block();
            }
}
void signal(semaphore S) {
        S.value++;
            if (S.value <= 0) {
                    remove a process P from S.L;
                wakeup(P);
            }
        }
```

- Khi S.value ≥ 0: số process có thể thực thi wait(S) mà không bị blocked = S.value
- Khi S.value < 0: số process đang đợi trên S là |S.value|
- Atomic và mutual exclusion: không được xảy ra trường hợp 2 process cùng đang ở trong thân lệnh wait(S) và signal(S) (cùng semaphore S) tại một thời điểm (ngay cả với hệ thống multiprocessor)

⇒ do đó, đoạn mã định nghĩa các lệnh wait(S) và signal(S) cũng chính là vùng tranh chấp

- Vùng tranh chấp của các tác vụ wait(S) và signal(S) thông thường rất nhỏ: khoảng 10 lệnh.

- Khi một process phải chờ trên semaphore S, nó sẽ bị blocked và được đặt trong hàng đợi semaphore
  - Hàng đợi này là danh sách liên kết các PCB
- Tác vụ signal() thường sử dụng cơ chế FIFO khi chọn một process từ hàng đợi và đưa vào hàng đợi ready
- block() và wakeup() thay đổi trạng thái của process
  - block: chuyển từ running sang waiting
  - wakeup: chuyển từ waiting sang ready
- Giải pháp cho vùng tranh chấp wait(S) và signal(S)
  - Uniprocessor: có thể dùng cơ chế cấm ngắt (disable interrupt). Nhưng phương pháp này không làm việc trên hệ thống multiprocessor.
  - Multiprocessor: có thể dùng các giải pháp software (như giải thuật Dekker, Peterson) hoặc giải pháp hardware (TestAndSet, Swap).
  - Vì CS rất nhỏ nên chi phí cho busy waiting sẽ rất thấp.

Có 3 loại semaphore:

- Counting semaphore: một số nguyên có giá trị không hạn chế.
- Binary semaphore: có trị là 0 hay 1. Binary semaphore rất dễ hiện thực.
- Có thể hiện thực counting semaphore bằng binary semaphore

Khi sử dụng semaphore cần lưu ý những vấn đề gì:

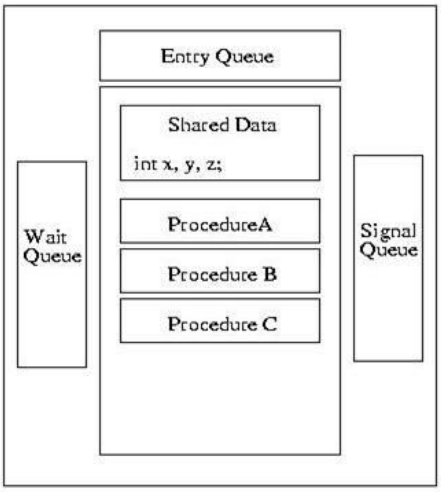
- Semaphore cung cấp một công cụ mạnh mẽ để bảo đảm mutual exclusion và phối hợp đồng bộ các process. Tuy nhiên, nếu các tác vụ wait(S) và signal(S) nằm rải rác ở rất nhiều processes ⇒ khó nắm bắt được hiệu ứng của các tác vụ này. Nếu không sử dụng đúng ⇒ có thể xảy ra tình trạng deadlock hoặc starvation.

- Một process bị “die” có thể kéo theo các process khác cùng sử dụng biến semaphore.

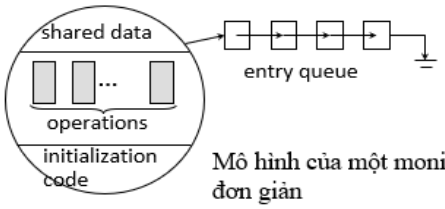
Câu 7: Monitor và Critical Region là gì?

Monito:

- Monito là một cấu trúc ngôn ngữ cấp cao tương tự CR, có chức năng như semaphore nhưng dễ điều khiển hơn
- Monitor, là một kiểu dữ liệu trừu tượng, lưu lại các biến chia sẻ và các phương thức dùng để thao tác lên các biến chia sẻ đó. Các biến chia sẻ trong monitor chỉ có thể được truy cập bởi các phương thức được định nghĩa trong monitor.
- Xuất hiện trong nhiều ngôn ngữ lập trình đồng thời như
  - Concurrent Pascal, Modula-3, Java,…
- Có thể hiện thực bằng semaphore
- Câu trúc của một monitor:



- Là một module phần mềm, bao gồm
  - Một hoặc nhiều thủ tục (procedure)
  - Một đoạn code khởi tạo (initialization code)
  - Các biến dữ liệu cục bộ (local data variable)



Mô hình của một monitor đơn giản

- Một monitor chỉ có 1 process hoạt động, tại một thời điểm bất kỳ đang xét.
- Bằng cách này, monitor sẽ đảm bảo mutual exclusion và dữ liệu chia sẻ của bạn sẽ được an toàn.

Critical Region:

- Critical Region là một cấu trúc ngôn ngữ cấp cao (high-level language construct, được dịch sang mã máy bởi một compiler), thuận tiện hơn cho người lập trình.
- Một biến chia sẻ v kiểu dữ liệu T, khai báo như sau

  v: shared T;

- Biến chia sẻ v chỉ có thể được truy xuất qua phát biểu sau

region v when B do S; /* B là một biểu thức Boolean */

- Ý nghĩa: trong khi S được thực thi, không có quá trình khác có thể truy xuất biến v.

Câu 8: Đặc điểm và yêu cầu đồng bộ của các bài toán đồng bộ kinh điển?

Ba bài toán đồng bộ kinh điển:

- Bounded Buffer Problem
- Dining-Philosophers Problem
- Readers and Writers Problem

❖ **Bounded Buffer Problem**

Producer sản suất một sản phẩm và đặt vào buffers, buffers giới hạn chỉ chứa được n sản phẩm. Consumer tiêu thụ mỗi lần một sản phẩm, sản phẩm được lấy ra từ buffers. Khi

buffers đã chứa n sản phẩm, Producer không thể đưa tiếp sản phẩm vào buffers nữa mà phải chờ đến khi buffers có chỗ trống.

Khi buffers rỗng, Consumer không thể lấy sản phẩm để tiêu thụ mà phải chờ đến khi có ít nhất 1 sản phẩm vào buffers.

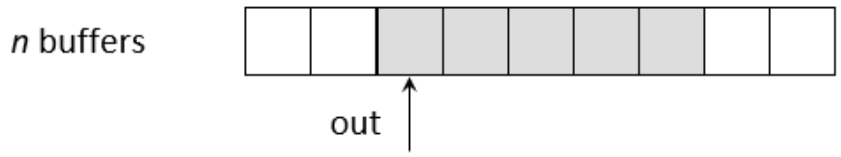
Để hiện thực bài toán trên, các biên chia sẻ giữa Producer và Consumer như sau: int n; semaphore mutex = 1; semaphore empty = n; semaphore full = 0;

♣ Buffers có n chỗ (n buffer con/vị trí) để chứa sản phẩm

♣ Biến semaphore mutex cung cấp khả năng mutual exclusion cho việc truy xuất tới buffers. Biến mutex được khởi tạo bằng 1 (tức value của mutex bằng 1).

♣ Biến semaphore empty và full đếm số buffer rỗng và đầy trong buffers.

♣ Lúc đầu, toàn bộ buffers chưa có sản phẩm nào được đưa vào: value của empty được khởi tạo bằng n; và value của full được khởi tạo bằng 0



❖ **Dining-Philosophers Problem**

- 5 triết gia ngồi ăn và suy nghĩ
- Mỗi người cần 2 chiếc đũa (chopstick) để ăn
- Trên bàn chỉ có 5 đũa
- Bài toán này minh họa sự khó khăn trong việc phân phối tài nguyên giữa các process sao cho không xảy ra deadlock và starvation

- Dữ liệu chia sẻ:
- Semaphore chopstick[5];
- Khởi đầu các biến đều là 1

Giải pháp

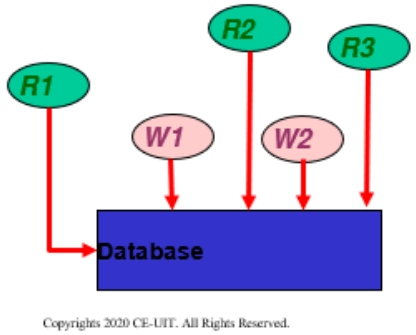
Triết gia thứ i*:*

do {

wait(chopstick [i]**)**
wait(chopstick [(i + 1) % 5]**)**
…
eat
…
signal(chopstick [i]**);**
signal(chopstick [(i + 1) % 5]**);**
…
think
…
} while (1);

❖ **Bài toán Reader-Writers**

- Writer không được cập nhật dữ liệu khi có một Reader đang truy xuất CSDL
- Tại một thời điểm, chỉ cho phép một Writer được sửa đổi nội dung CSDL



- Bộ đọc trước bộ ghi (first reader-writer)
- Dữ liệu chia sẻ

  semaphore mutex = 1;
  semaphore wrt = 1;
  int readcount = 0;
- Writer process

  wait(wrt);
  ...
  writing is performed
  ...
  signal(wrt);

Reader process

```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex);
```

- mutex: "bảo vệ" biến readcount
- wrt
  - Bảo đảm mutual exclusion đối với các writer
  - Được sử dụng bởi reader đầu tiên hoặc cuối cùng vào hay ra khỏi vùng tranh chấp.
- Nếu một writer đang ở trong CS và có n reader đang đợi thì một reader được xếp trong hàng đợi của wrt và n − 1 reader kia trong hàng đợi của mutex
- Khi writer thực thi signal(wrt), hệ thống có thể phục hồi thực thi của một trong các reader đang đợi hoặc writer đang đợi.

Câu 9: (Bài tập mẫu) Xét giải pháp phần mềm do Dekker đề nghi để tổ chức truy xuất độc quyền cho 2 tiến trình. Hai tiến trình P0 và P1 chia sẻ các biến sau:

```
boolean flag[2]; /* initially false */
int turn;
```

Cấu trúc một tiến trình Pi (với i = 0 hay 1 và j là tiến trình còn lại) như sau:

```
while (true) {
        flag[i] = true;

        while (flag[j]) {
                if (turn == j) {
                        flag[i] = false;
                        while (turn == j)
                            ; /* do nothing */
                        flag[i] = true;
                }
        }
                /* critical section */
        turn = j;
        flag[i] = false;
                /* remainder section */
}
```

Giải pháp này có thỏa 3 yêu cầu trong viêc giải quyết tranh chấp không?

Trả lời:

Giải pháp này thỏa 3 yêu cầu trong giải quyết tranh chấp vì:

- Loại trừ tương hỗ: Tiến trình Pi chỉ có thể vào vùng tranh chấp khi flag[j] = false. Giả sử P0 đang ở trong vùng tranh chấp, tức là flag[0] = true và flag[1] = false. Khi đó P1 không thể vào vùng tranh chấp (do bị chặn bởi lệnh while (flag[j])). Tương tự cho tình huống P1 vào vùng tranh chấp trước.
- Progress: Giá trị của biến turn chỉ có thể thay đổi ở cuối vùng tranh chấp. Giả sử chỉ có 1 tiến trình Pi muốn vào vùng tranh chấp. Lúc này, flag[j] = false và tiến trình Pi sẽ được vào vùng tranh chấp ngay lập tức. Xét trường hợp cả 2 tiến trình đều muốn vào vùng tranh chấp và giá trị của turn đang là 0. Cả flag[0] và flag[1] đều bằng true. Khi đó, P0 sẽ được vào vùng tranh chấp, bởi tiến trình P1 sẽ thay đổi flag[1] = false (lệnh kiểm tra điều kiện if (turn == j) chỉ đúng với P1). Tương tự cho trường hợp turn = 1.
- Chờ đợi giới hạn: Pi chờ đợi lâu nhất là sau 1 lần Pj vào vùng tranh chấp (flag[j] = false sau khi Pj ra khỏi vùng tranh chấp). Tương tự cho trường hợp Pj chờ Pi.

Câu 10: Xét giải pháp đồng bộ hóa sau:

```
while (TRUE) {
        int j = 1-i;
        flag[i]= TRUE;
        turn = i;
        while (turn == j && flag[j]==TRUE);
        critical-section ();
        flag[i] = FALSE;
        Noncritical-section ();
}
```

Giải pháp này có thỏa yêu cầu độc quyền truy xuất không?

Trả lời:

Giải pháp này không thoả mãn yêu cầu độc quyền truy xuất

Đoạn j = 1 – i => đề chỉ đang nói tới xét 2 tiến trình P1 và P0 vì khi i = 0 thì j = 1 và ngược lại.

Xét tình huống khi:

flag[0] =1;

turn =0;

lúc này P0 vào CS

Nếu lúc đó flag[1] = 1, P1 có thể gán turn = 1.

P1 sẽ vào luôn CS (2 tiến trình cùng vào CS một lúc).

Câu 11: Giả sử một máy tính không có chỉ thị TSL, nhưng có chỉ thị Swap có khả năng hoán đổi nội dung của hai từ nhớ chỉ bằng một thao tác không thể phân chia:

```
procedure Swap(var a,b: boolean){
        var temp : boolean;
        begin
                temp := a;
                a:= b;
                b:= temp;
```

```
        end;
}
```

Sử dụng chỉ thị này có thể tổ chức truy xuất độc quyền không? Nếu có, xây dựng cấu trúc chương trình tương ứng.

Trả lời:

```
while (TRUE) {

key = TRUE;

while ( key = TRUE)

Swap (lock, key);

critical-section();

lock = false;

Noncritical-section();

}
```

Câu 12: Xét hai tiến trình sau:

process A {while (TRUE) na = na +1;}
process B {while (TRUE) nb = nb +1;}

a. Đồng bộ hóa xử lý của 2 tiến trình trên, sử dụng 2 semaphore tổng quát, sao cho tại bất kỳ thời điểm nào cũng có nb <= na <= nb +10.

Trả lời:

```
Semaphore Semaphore_1, Semaphore_2
Semaphore_1.value = 0;
Semaphore_2.value = 10;
Process A:
while (1) {
    wait (Semaphore_ 2) ;
    na = na + 1;
```

```
    signa1 (Semaphore__1) ;
}
Process B:
while (1) {
    wait (Semaphore_ 1) ;
    nb = nb + 1;
    signa1 (Semaphore_2) ;
}
```

b. Nếu giảm điều kiện chỉ còn là na <= nb +10, cần sửa chữa giải pháp trên như thế nào?

Trả lời:

```
Semaphore s2
Semaphore_2.value = 10;
Process A:
while (1) {
wait (Semaphore_ 2) ;
na = na + 1;
}
Process B:
while (1) {
nb = nb + 1;
signa1 (Semaphore__2) ;
}
```

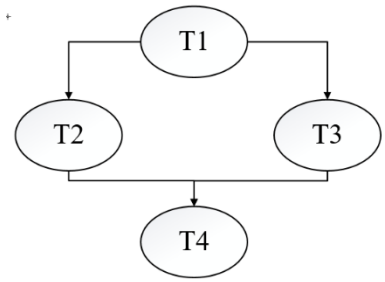
c. Giải pháp trên còn đúng nếu có nhiều tiến trình loại A và B cùng thực hiện?

Trả lời:

Đúng, vì có thể có nhiều tiến trình loại A hoặc loại B cùng thực hiện nhưng chỉ có 2 biến Semaphore toàn cục mà chúng sẽ thao tác (đối với câu b là 1 Semaphore).

Câu 13: (Bài tập mẫu) Xét một hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ bên dưới, với mũi tên từ tiểu trình (Tx) sang tiểu trình (Ty) có nghĩa là tiểu trình Tx phải kết thúc quá trình hoạt động của nó trước khi tiểu trình Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi

tạo và sẵn sàng để thực thi. Hãy sử dụng semaphore để đồng bộ hoạt động của các tiểu trình sao cho đúng với sơ đồ đã cho.



Trả lời:

Khai báo và khởi tạo các semaphore:

init(sem1,0); //khởi tạo semaphore sem1 có giá trị bằng 0

init(sem2,0); //khởi tạo semaphore sem2 có giá trị bằng 0

```
void T1(void)
{



//T1 thực thi


signal(sem1)
signal(sem1)
}
```

```
void T2(void)
{


wait(sem1)


//T2 thực thi


signal(sem2)


}
```

```
void T3(void)
{


wait(sem1)


//T3 thực thi


signal(sem2)


}
```

```
void T4(void)
{


wait(sem2)
wait(sem2)
//T4 thực thi




}
```

Câu 14: Một biến X được chia sẻ bởi hai tiến trình cùng thực hiện đoạn code sau:

```
do
        X = X + 1;
        if (X == 20) X = 0;
while (TRUE);
```

Bắt đầu với giá trị X = 0, chứng tỏ rằng giá tri X có thể vượt quá 20. Cần sửa chữa đoạn chương trình trên như thế nào để đảm bảo X không vượt quá 20?

Trả lời:

Do X được chia sẻ chung ở cả hai tiến trình và chỉ bị reset khi X == 20 nên X có thể vượt quá 20 khi có một lý do trong máy đột ngột khiến tiến trình 2 dừng lại. Sau đó tại tiến trình 1, khi X = 19 thì tiến trình 2 được release đúng ngay đoạn X = X + 1, và cộng dồn với X = 20 sau lệnh X = X + 1 của tiến trình 1 dẫn tới X vượt quá 20 và còn bị cộng tới vô cùng.

- Để đảm bảo X không vượt quá 20 thì ta có thể giải quyết bằng cách đưa 2 tiến trình về làm 1, tức là làm cho cùng một lúc chỉ có một trong hai tiến trình được chạy bằng một biến Semaphore khởi tạo bằng 1.

Code:

```
Semaphore mutex = 1;
do {
down(mutex);
X = X + 1;
if (X == 20 )
    X = 0;
up (mutex);
} while (true);
}
```

Câu 15: Xét 2 tiến trình xử lý đoạn chương trình sau:

process P1 {A1 ; A2 }
process P2 {B1 ; B2 }

Đồng bộ hóa hoạt động của 2 tiến trình này sao cho cả A1 và B1 đều hoàn tất trước khi A2 và B2 bắt đầu.

Trả lời:

```
Semaphore Semaphore 1, Semaphore 1;
Semaphore_1.value = 0;
```

Semaphore2.value = 0;
process Pl {
A1;
wait (Semaphore 1) ;
signa1 (Semaphore2) ;
A2;
}
process P2 {
B1;
wait (Semaphore 2) ;
Signa1 (Semaphore1) ;
B2 ;
}

Câu 16: Tổng quát hóa bài tập 14 cho các tiến trình có đoạn chương trình sau:

process P1 { for (i = 1; i <= 100; i ++) A_i }
process P2 { for (j = 1; j <= 100; j ++) B_j }
Đồng bộ hóa hoạt động của 2 tiến trình này sao cho với k bất kỳ (2<=k<=100), A_k chỉ có thể bắt đầu khi $B_{(k-1)}$ đã kết thúc và B_k chỉ có thể bắt đầu khi $A_{(k-1)}$ đã kết thúc.

Trả lời:

Semaphore Semaphore_1, Semaphore_1

Semaphore_1.value = 1;

Semaphore_2.value = 1;

Process P1:

for(int i = 1; i <= 100; i++) {

wait (Semaphore _1) ;

Ai;

signa1 (Semaphore_2)

}

Process B:

for(i = 1; i <= 100; i++) {

wait (Semaphore_2) ;

Bi;

signa1 (Semaphore_1) ;

}

1. Sử dụng semaphore để viết lại chương trình sau theo mô hình xử lý đồng hành:

w := x1 * x2
v := x3 * x4
y := v * x5
z := v * x6
x := w * y
z := w * z
ans := y + z

<u>Trả lời:</u>

Semaphore s1,s2,s3,s4,s5,s6,s7,s8,s9

S1.value=s2.value=s3.value=s4.value=….= s8.value =0

Process P1:

w := x1* x2

signal(s1)

signal(s2)

Process P2:

v := x3*x4

signal(s3)

signal(s4)

Process P3:

Wait(s3)

Y := v * x5

Signal(s5)

Signal(s6)

Process P4:

Wait(s4)

Z := v * x6

Signal(s7)

Process P5:

Wait(s2)

Wait(s5)

X := w *y

Process P6:

Wait(s1)

Wait(s7)

Z := w * z

Signal(s8)

Process P7:

Wait(s6)

Wait(s8)

Ans := y + z