Chương 3

Đồng bộ và giải quyết tranh chấp (Process Synchronization)

Nội dung

- Khái niệm cơ bản
- Bài toán "Critical-Section"
- Các giải pháp phần mềm
  - Peterson, Bakery
- Đồng bộ bằng hardware
- Semaphore
- Các bài toán đồng bộ
- Critical Region
- Monitor

Khái niệm cơ bản

- Các process/thread thực thi đồng thời chia sẻ code, chia sẻ dữ liệu (qua shared memory, file).
- Nếu không có sự điều khiển khi truy cập các dữ liệu chia sẻ thì có thể xảy ra trường hợp không nhất quán dữ liệu (data inconsistent).
- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có thứ tự của các process đồng thời.
- Ví dụ Bounded-Buffer (ch.4) thêm biến đếm count

```
#define BUFFER_SIZE 10
# typedef struct {
      … } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0, count = 0;
```


Bounded Buffer (t.t)

- Producer

```
item nextProduced;
while (1){
    while ( count == BUFFER_SIZE ); /* do nothing */
    buffer[in] = nextProduced;
    count++;
    in = (in + 1) % BUFFER_SIZE;
}
```

- Consumer

```
item nextConsumed;
while (1){
    while ( count == 0 ); /* do nothing */
    buffer[in] = nextConsumed;
    count--;
    out = (out + 1) % BUFFER_SIZE;
}
```

Race Condition

- *Race condition*: nhiều process truy xuất và thao tác đồng thời trên 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.
- Chúng ta cần bảo đảm sao cho tại mỗi thời điểm có một và chỉ một process được truy xuất, thao tác trên dữ liệu chia sẻ. Do đó, cần có cơ chế ***đồng bộ hoạt động*** của các process này.

- Các lệnh tăng, giảm biến tương đương trong ngôn ngữ máy là:
- (P) count ++;
  - $register_1$:= count
  - $register_1$:= $register_1$ +1
  - count := $register_1$
- (C) count --;
  - $register_2$:= count
  - $register_2$:= $register_2$ -1
  - count := $register_2$
- Trong đó, $register_i$ là các thanh ghi của CPU.

Ví dụ về Race Condition

- Quá trình thực hiện xen kẽ của lệnh tăng/giảm biến count
- Hiện tại: count = 5

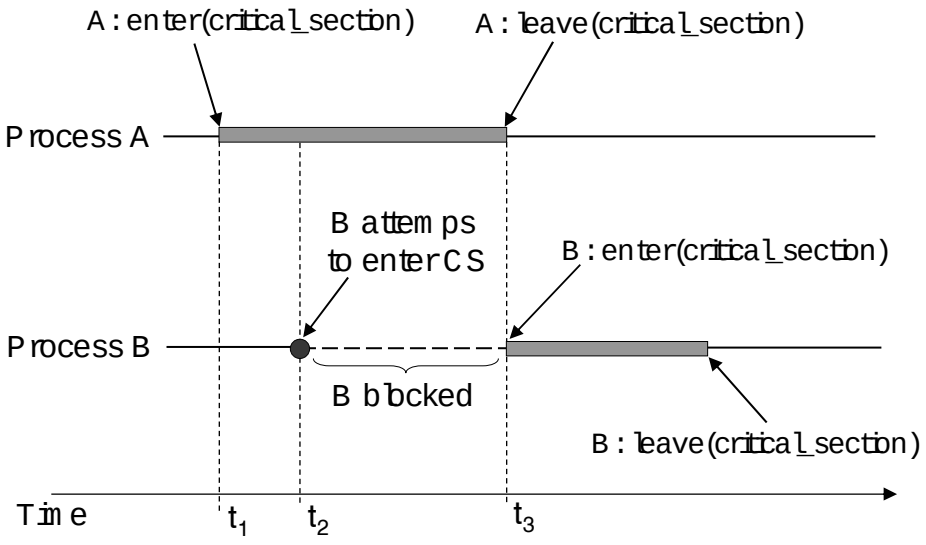
0: *producer* `register1 := count` {$register_1 = 5$}
1: *producer* `register1 := register1+1` {$register_1 = 6$}
2: *consumer* `register2 := count` {$register_2 = 5$}
3: *consumer* `register2 := register2-1` {$register_2 = 4$}
4: *producer* `count := register1` {count = 6}
5: *consumer* `count := register2` {count = 4}

- Cả hai process thao tác đồng thời trên biến chung *count*. Kết quả của biến chung này không nhất quán dưới các thao tác của hai process ⇒ lệnh count++, count-- phải là ***atomic***, nghĩa là thực hiện như một lệnh đơn, không bị ngắt nửa chừng.

Critical Section

- Giả sử có n process cùng truy xuất đồng thời dữ liệu chia sẻ
- Không phải tất cả các đoạn code đều phải được quan tâm giải quyết vấn đề *race condition* mà chỉ những đoạn code có chứa các thao tác trên dữ liệu chia sẻ. Đoạn code này được gọi là vùng tranh chấp - *critical section* (CS).
- ***Vấn đề đặt ra***: phải bảo đảm rằng khi một process đang thực thi trong vùng tranh chấp, không có một process nào khác được phép thực thi các lệnh trong vùng tranh chấp ⇒ *mutual exclusion* (mutex): sự loại trừ tương hỗ.

Critical Section và Mutual Exclusion



Cấu trúc tổng quát

- Giả sử mỗi process thực thi bình thường (i.e, nonzero speed) và không có sự tương quan giữa tốc độ thực thi của các process
- Cấu trúc tổng quát của một process:

```
DO {
        entry section
     critical section
        exit section
     remainder section
} WHILE (1);
```

Một số giả định:

- Có thể có nhiều CPU nhưng không cho phép có nhiều tác vụ truy cập một vị trí trong bộ nhớ cùng lúc (simultaneous)
- Không ràng buộc về thứ tự thực thi của các process
- Các process có thể chia sẻ một số biến chung nhằm mục đích đồng bộ hoạt động của chúng.
- Giải pháp của chúng ta cần phải đặc tả được các phần ***entry section*** và ***exit section***.

Ràng buộc của bài toán tranh chấp

- *Mutual Exclusion*
  - Tại mỗi thời điểm, chỉ có một process được phép thực thi trong vùng tranh chấp (CS)
- *Progress*: nếu không có process nào đang thực thi trong vùng tranh chấp và đang có một số process chờ đợi vào vùng tranh chấp thì:
  - Chỉ những process không phải đang thực thi trong vùng không tranh chấp mới được là ứng cử viên cho việc chọn process nào được vào vùng tranh chấp kế tiếp.
  - Quá trình chọn lựa này không được trì hoãn vô hạn (postponed indefinitely)
- *Bounded Waiting*
  - Mỗi process chỉ phải chờ trong để được vào vùng tranh chấp trong một khoảng thời gian nào đó. Không để xảy ra tình trạng "đói tài nguyên" (*starvation)*

Phân loại giải pháp

- Giải pháp phần mềm (software solutions)
  - user/programmer tự thực hiện (thông thường sẽ có sự hỗ trợ của các thư viện lập trình)
  - OS cung cấp một số công cụ (các hàm và cấu trúc dữ liệu) hỗ trợ cho programmer qua system calls.
- Giải pháp phần cứng (hardware solutions)
  - Dựa trên một số lệnh máy đặc biệt
    - » Interrupt disable, Test-and-Set

Giải pháp phần mềm

- Trường hợp 2 process đồng thời
  - Giải thuật 1 và 2
  - Giải thuật 3 (Peterson's algorithm)
- Giải thuật tổng quát cho n process
  - Bakery algorithm

Giải thuật 1

- Biến chia sẻ
  - **int turn**; /* khởi đầu **turn = 0** */
  - nếu **turn = i** $\Rightarrow P_i$ được phép vào ***critical section***
- Process P_i

```
do {
        while (turn != i) ;
            Critical_Section();
        turn = j;
            Remainder_Section();
} while (1);
```

- Thoả mãn mutual exclusion (1)
- Không thoả mãn yêu cầu progress (2) và bounded-waiting (3) vì tính chất strict alternation.

Giải thuật 1 (t.t)

```
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);
```

Ví dụ:
P0 có RS rất lớn và P1 có RS nhỏ. Nếu turn=0, P0 được vào CS và sau đó thực thi vùng RS (turn=1). Đến P1 vào CS và sau đó thực thi RS (turn=0) và tìm cách vào CS một lần nữa nhưng yêu cầu bị từ chối !!! P1 phải chờ P0 !!!.

Giải thuật 2

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

```
do {
        flag[i] := true;
        while (flag[j]) ;
        Critical_Section();
        flag [i] = false;
        Remainder_Section();
} while (1);
```

- Bảo đảm được mutual exclusion. Chứng minh?
- Không thoả mãn progress. Vì sao?

Giải thuật 3 (Peterson)

- Biến chia sẻ: kết hợp cả giải thuật 1 và 2.
- Process P_i

```
do {
   flag [i]:= true;
   turn = j;
   while (flag [j] and turn = j) ;
   Critical_Section();
   flag [i] = false;
   Remainder_Section();
} while (1);
```

- Thoả mãn được cả 3 yêu cầu (chứng minh - ?), giải quyết bài toán “critical-section” cho 2 process.

Giải thuật Peterson-2 process

```
PROCESS P0
DO {
  flag[0]:=true;
  /* 0 wants in  */
  turn:= 1;
  /* 0 gives a chance to 1 */
  WHILE
    ( flag[1] && turn=1 );
  CRITICAL_SECTION;
  flag[0]:=false;
  /* 0 no longer wants in */
  REMAINDER_SECTION;
WHILE (1);
```

```
PROCESS P1
DO {
  flag[1]:=true;
  /*  1 wants in  */
  turn:=0;
  /* 1 gives a chance to 0 */
  WHILE
    ( flag[0] && turn=0 );
  CRITICAL_SECTION;
  flag[1]:=false;
  /* 1 no longer wants in */
  REMAINDER_SECTION;
WHILE (1);
```


Giải thuật 3: Tính đúng đắn

- Mutual exclusion được bảo đảm bởi vì
  - P0 và P1 đều ở trong CS nếu và chỉ nếu *flag[0] = flag[1] = true* và chỉ nếu *turn = i* với mỗi Pi (không thể xảy ra)
- Chứng minh thoả 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.

Giải thuật 3: Tính đúng đắn (t.t)

- 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* thì Pi vào CS. Nếu *turn=j* thì Pj vào CS nhưng sẽ bật *flag[j]=false* khi thoát ra $\Rightarrow$ cho phép Pi vào 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)

Trường hợp process bị "chết"

- Nếu thỏa đồng thời 3 yêu cầu (mutex, progress, bounded waiting) thì giải pháp giải quyết tranh chấp có khả năng phát hiện một process bị "chết" tại những vùng không có tranh chấp (remainder section)
  - Khi đó, process bị "chết" tại các vùng không tranh chấp cũng có nghĩa là có một remainder section dài vô hạn.
- Không có giải pháp nào có thể cung cấp cơ chế đủ mạnh để giải quyết trường hợp process bị "chết" bên trong critical section (CS)

Giải thuật Bakery: N process

- Trước khi vào CS, process Pi nhận một con số. Process nào giữa 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, ngược lại Pj được vào trước.
- 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 if a == c và b < d
  - max(a_0,...a_k) là con số b sao cho b >= a_i với mọi i=0,..k

Giải thuật Bakery: N process

```
/* shared variable */
bool select[n];              /* initially, select[i] =false   */
integer  num[n];             /* initially, num[i] = 0         */

while (1) {
        select[i] = true;
        number[i] = max(num[0], num[1], …, num [n – 1]) + 1;
        select[i] = false;
        for (j = 0; j < n; j ++) {
                while (select[j]);
                while ((num[j] != 0) && (num[j], j) < num[i], i));
        }
        Critical_Section();
        num[i] = 0;
        Remainder_Section();
}
```

Từ software đến hardware

- 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), tiêu tốn lãng phí 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 đang đợi.
- Các giải pháp phần cứng (hardware)
  - Cấm ngắt (disable interrupts)
  - Dùng các lệnh đặc biệt

Cấm ngắt

- Trong hệ thống uniprocessor: mutual exclusion được bảo đảm tuy nhiên hiệu suất thực thi bị giảm sút vì khi có process trong CS, chúng ta không thể thực hiện cách thực thi xen kẽ đối với các process đang ở vùng không có tranh chấp (non-CS).
- Trên hệ thống multiprocessor: mutual exclusion không được đảm bảo
  - CS có tính atomic nhưng không mutual exclusive

```
Process Pi:
repeat
        disable_interrupts();
        Critical_Section();
        enable_interrupts();
        Remainder_Section()
forever
```

Dùng các lệnh đặc biệt

- Ý tưởng cơ sở
  - Việc truy xuất vào vào một địa chỉ của bộ nhớ vốn đã có tính loại trừ tương hỗ (chỉ có một thao tác truy xuất tại một thời điểm)
- Mở rộng
  - thiết kế một lệnh máy có thể thực hiện hai thao tác chập (atomic, indivisible) trên cùng một ô nhớ (vd: read và write)
  - Việc thực thi các lệnh máy như trên luôn bảo đảm mutual exclusive (ngay cả với hệ thống multiprocessor)
- Các lệnh máy đặc biệt có thể đảm bảo mutual exclusion tuy nhiên cũng cần kết hợp với một số cơ chế khác để thoả mãn hai yêu cầu còn lại là progress và bounded waiting cũng như tránh tình trạng starvation và deadlock.

Lệnh test-and-set

- Kiểm tra và cập nhật một biến trong một thao tác đơn (atomic).

```
boolTest&Set(booltarget)
{
        boolrv = target;
        target = true;
        return rv;
}
```

■Shared data:

```
boollock = false;
```

■Process P_i

```
while
{
      while (Test&Set(lock));
      Critical_Section;
      lock = false;
      Remainder_Section;
 }
```

Lệnh test-and-set (t.t)

- Mutual exclusion được bảo đảm: nếu P_i vào CS, các process P_j khác đều đang busy waiting
- Khi P_i ra khỏi CS, quá trình chọn lựa process P_j vào CS kế tiếp là tuỳ ý $\Rightarrow$ 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 test-and-set

Swap và Mutual Exclusion

- 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(boola,boolb)
{
        boolean temp = a;
        a = b;
        b = temp;
}
```

- Biến chia sẻ (khởi tạo là **false**)
  bool lock;
  bool waiting[n];
- Process P_i

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

Thoả mãn 3 yêu cầu

- 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 là key==false
  - key = false chỉ khi Test&Set (hay Swap) được thực thi
    - » Process đầu tiên thực thi Test&Set 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 exclusion
- Bounded Waiting: waiting in the cyclic order

Thoả mãn 3 yêu cầu (t.t)

```
while {
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
                    key = Test&Set(lock);
        waiting[i] = false;
          CriticalSection
        j = (i+1) % n;
        while ((j != i) && !waiting[j])
                    j = (j+1) % n;
        if (j = i)
                    lock = false;
        else
                    waiting[j] = false;
          RemainderSection
}
```

Semaphore

- Là công cụ đồng bộ cung cấp bởi OS mà không đòi hỏi busy waiting
- Semaphore S là một biến số nguyên, ngoài thao tác khởi động biến thì chỉ có thể được truy xuất qua hai tác vụ có tính đơn nguyên (atomic) và loại trừ (mutual exclusive)
  - wait(S) hay còn gọi là P(S): giảm giá trị semaphore. 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. Nếu giá trị này âm, 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.

Hiện thực Semaphore

- Định nghĩa semaphore như một record

```
typedef struct {
        int value;
        struct process *L; /* process queue */
} semaphore;
```

- Giả sử có hai tác vụ đơn:
  - **block:** tạm treo process nào thực thi lệnh này.
  - **wakeup(*P*)**: hồi phục quá trình thực thi của một process **P** đang blocked .

Hiện thực Semaphore (t.t)

- Các tác vụ semaphore được định nghĩa như sau

wait(*S*):

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block;
}
```

signal(*S*):

```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
}
```


Hiện thực Semaphore (t.t)

- 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 để chuyển một process từ hàng đợi và đưa vào hàng đợi ready
- block() và wakeup() thay đổi trạng thái của process – đó là các system call cơ bản.
  - Block: chuyển từ running sang waiting
  - wakeup: chuyển từ waiting sang ready

Hiện thực Mutex với Semaphore

- Dùng cho n process
- Khởi tạo S.value = 1
- Chỉ duy nhất một 1 process được vào CS (mutual exclusion)
- Để cho phép k process vào CS, khởi tạo S.value = k

- Shared data:

```
semaphore mutex;
/*initially mutex.value = 1*/
```

- Process *Pi:*

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```


Đồng bộ process bằng semaphore

- Hai process: P1 và P2
- Yêu cầu: lệnh S1 trong P1 cần được thực thi trước lệnh S2 trong P2
- Định nghĩa semaphore "synch" dùng đồng bộ
- Khởi động semaphore:
  synch.value= 0

- Để đồng bộ hoạt động theo yêu cầu, P1 phải định nghĩa như sau:

```
S1;
signal(synch);
```

- Và P2 định nghĩa như sau:

```
wait(synch);
S2;
```

Nhận xét

- 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 = |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)

  $\Rightarrow$ do đó, đoạn mã định nghĩa các lệnh wait(S) và signal(S) cũng chính là vùng xảy ra tranh chấp (critical section)

Nhận xét (t.t)

- wait() và signal() phải thực thi atomic và mutual execution
- Vùng tranh chấp của các tác vụ wait(S) và signal(S) thông thường rất nhỏ: khoảng 10 lệnh.
- Giải pháp cho vùng tranh chấp wait(S) và signal(S):
  - Uniprocessor: có thể dùng cơ chế cấm ngắt (disable interrupt) để giải quyết tranh chấp (trong một khoảng thời gian rất ngắn). Phương pháp này không làm việc trên hệ thống multiprocessor.
  - Multiprocessor: có thể dùng các giải pháp software (như Dekker, Peterson) hoặc giải pháp hardware (test-and-set, swap). Vì CS rất nhỏ nên chi phí cho busy waiting sẽ rất thấp.

Deadlock và Starvation

- Deadlock – hai hay nhiều process đang chờ đợi vô hạn định một sự kiện không bao giờ xảy ra (vd: sự kiện do một trong các process đang đợi tạo ra).
- Gọi S và Q là hai biến semaphore được khởi tạo = 1

P0	**P1**
P(S);	P(Q);
P(Q);	P(S);
⋮	⋮
V(S);	V(Q);
V(Q)	V(S);

- Starvation (indefinite blocking) có thể xảy ra khi chúng ta bỏ process vào hàng đợi và lấy ra theo cơ chế LIFO.

Các loại semaphore

- Counting semaphore: một số nguyên có giá trị không hạn chế.
- Binary semaphore: một số nguyên có giá trị nằm trong khoảng [0,1]. Binary semaphore rất dễ hiện thực.
- Chúng ta có thể hiện thực counting semaphore S bằng binary semaphore.

Các bài toán đồng bộ

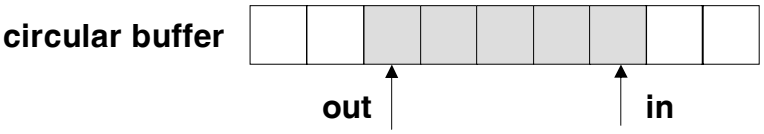
- Bài toán Bounded-Buffer
  - Dữ liệu chia sẻ:

    SEMAPHORE full, empty, mutex;
  - Khởi tạo:

    full = 0;

    empty = BUFFER_SIZE;

    mutex = 1;



Bounded Buffer (t.t)

PRODUCER

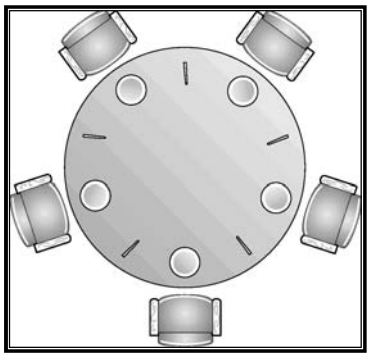
```
do {
…
      nextp = new_item();
      …
      wait(empty);
      wait(mutex);
       …
      insert_to_buffer(nextp);
       …
      signal(mutex);
      signal(full);
}while (1);
```

CONSUMER

```
do {
      wait(full)
      wait(mutex);
       …
      nextc = get_buffer_item (out);
      …
      signal(mutex);
      signal(empty);
       …
      consume_item (nextc);
       …
}while (1);
```

Bài toán "Dining Philosopher"

- 5 triết gia ngồi ăn và suy nghĩ
- Mỗi người cần 2 chiếc đũa để ăn
- Trên bàn chỉ có 5 đũa
- Bài toán này minh hoạ 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

Bài toán "Dining Philosopher"

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 "Dining Philosopher"

- Giải pháp trên có thể gây ra deadlock
  - Khi tất cả triết gia đói bụng cùng lúc và đồng thời cầm một chiếc đũa bên tay trái ⇒ deadlock
- Một số cách giải quyết deadlock
  - Cho phép nhiều nhất 4 triết gia ngồi vào cùng một lúc.
  - Cho phép triết gia cầm đũa khi cả hai chiếc đũa đều sẵn sàng (nghĩa là tác vụ cầm đũa phải xảy ra trong CS)
  - Triết gia ngồi ở vị trí lẻ cầm đũa bên trái trước, sau đó mới đến đũa bên phải, trong khi đó triết gia ở vị trí chẵn cầm đũa bên phải trước, sau đó mới đến đũa bên trái
- Có thể xảy ra trường hợp starvation (tại sao?)

Bài toán Readers-Writers

- Dữ liệu chia sẻ

```
semaphore mutex=1;
semaphore wrt = 1;
integer 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):
```

Bài toán Readers-Writers (t.t)

- 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), chúng ta có thể phục hồi thực thi của một trong các reader đang đợi hoặc writer đang đợi ⇒ dựa vào scheduler

Các vấn đề với semaphore

- Semaphore cung cấp một công cụ mạnh mẽ để bảo đảm mutual exclusion và phối hợp đồng bộ các process
- Tuy nhiên, 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.

```
signal(mutex)
…
criticalsection
…
wait(mutex)
```

```
wait(mutex)
…
criticalsection
…
wait(mutex)
```

```
signal(mutex)
…
criticalsection
…
signal(mutex)
```

Critical Region (CR)

- Critical Region còn được gọi là conditional critical region (CCR)
- *Cơ sở*: dựa trên semaphore, xây dựng cấu trúc ngôn ngữ cấp cao, thuận tiện hơn cho người lập trình.
- Một biến chia sẻ ***v*** kiểu dữ liệu ***T***, khai báo như sau:

  v: shared T
- Biến chung v chỉ có thể được truy xuất qua phát biểu sau

  region v **when** B **do** S
- Các region tham chiếu đến cùng một biến chia sẻ thì bị loại trừ tương hỗ.
- Khi một process muốn thực thi các lệnh trong region, biểu thức Boolean *B* được kiểm tra. Nếu *B*=true, lệnh *S* được thực thi. Nếu B= false, process bị trì hoãn cho đến khi *B* = true và không có process nào đang ở trong region cùng tham chiếu đến biến *v*.

CR và bài toán bounded-buffer

Dữ liệu chia sẻ:

```
struct buffer
{
        int pool[n];
        int count;
        int in;
        int out;
}
```

Producer

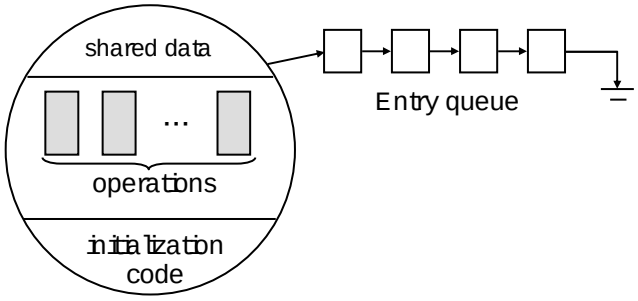
```
region buffer when (count < n) {
        pool[in] = nextp;
        in = ( in + 1 ) % n;
        count++;
}
```

Consumer

```
region buffer when (count > 0){
        nextc = pool[out];
        out = ( out + 1 ) % n;
        count--;
}
```

Monitor

- Cũng là một cấu trúc ngôn ngữ cấp cao (high-level language construct) tương tự CR, có chức năng như semaphore nhưng dễ điều khiển hơn
- Xuất hiện trong nhiều ngôn ngữ lập trình đồng thời như
  - Concurrent Pascal, Modula-3, uC++, Java...
- Có thể hiện thực bằng semaphore (!!!)



Monitor

- Là một module phần mềm trong đó bao gồm
  - Một hoặc nhiều thủ tục/hàm (procedure)
  - Một đoạn code khởi tạo (initialization code)
  - Các biến dữ liệu cục bộ (local data variable)
- Đặc tính của monitor:
  - Local variable chỉ có thể truy xuất bởi các thủ tục của monitor
  - Process vào monitor bằng cách gọi một trong các thủ tục đó
  - Chỉ có một process có thể vào monitor tại một thời điểm $\Rightarrow$ Mutual Exclusion được bảo đảm
- Hiện thực monitor (tham khảo tài liệu)

Cấu trúc của Monitor

```
monitor monitor-name
{
        shared variable declarations
        procedure body P1 (… ){
                ...
        }
        procedure body P2 (… ){
                ...
        }
        procedure body Pn (… ){
                ...
        }
        {
                initialization code
        }
}
```

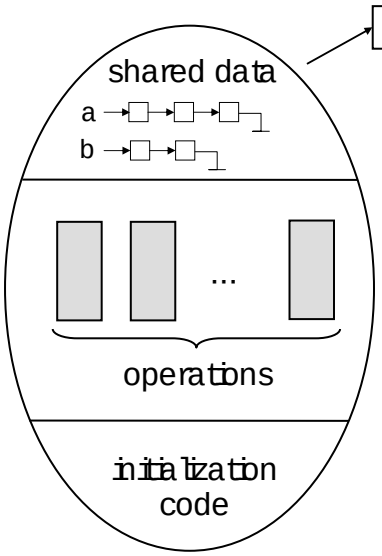

Condition Variable

- Nhằm cho phép một process đợi trong monitor, phải khai báo biến điều kiện (condition variable)

  condition a, b;

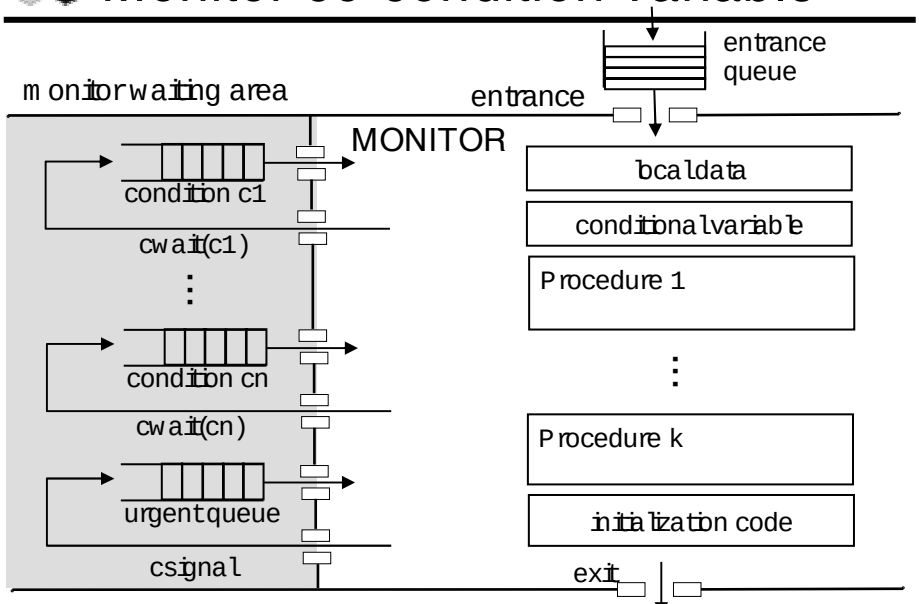
- Các biến điều kiện đều cục bộ và chỉ được truy cập bên trong monitor.
- Chỉ có thể truy xuất và thay đổi bằng hai thủ tục:
  - **cwait(a)**: process gọi tác vụ này sẽ bị block trên biến điều kiện a
    - » process này chỉ có thể tiếp tục thực thi khi có process khác thực hiện tác vụ **csignal(a);**
  - **csignal(a)**: phục hồi quá trình thực thi của process bị block trên biến điều kiện a.
    - » Nếu có nhiều process: chỉ chọn một
    - » Nếu không có process: không có tác dụng.

Monitor có condition variable



- Các process có thể đợi ở *entrance queue* hoặc đợi ở các *condition queue (a, b,...)*
- Khi thực hiện lệnh cwait(a), process sẽ được chuyển vào *condition queue a*
- Lệnh csignal(a) chuyển một process trong *condition queue a* vào monitor
- Do đó, process gọi csignal(cn) sẽ bị blocked và được đưa vào urgent queue

Monitor có condition variable



Monitor và Dining Philosopher

```
type dining-philosophers = monitor
{
   var state : array [0..4] of :(thinking, hungry, eating);
   var self : array [0..4] of condition;

   procedure pickup (i: 0..4);
      begin
            state[i] := hungry;
            test (i);
            if state[i] ≠ eating then self[i].wait;
      end;
}
```


Monitor và Dining Philosopher

```
monitor dp
{
      enum {thinking, hungry, eating} state[5];
      condition self[5];
      void pickup(int i);
      void putdown(int i);
      void test(int i);
      void init()
      {
            for (int i = 0; i < 5; i++)
                  state[i] = thinking;
      }
}
```

Dining Philosopher (t.t)

```
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
        self[i].wait();
}
void putdown(int i) {
    state[i] = thinking;
    // test left and right neighbors
    test((i+4) % 5);
    test((i+1) % 5);
}
```


Dining Philosopher (t.t)

```
void test(int i) {
    if ( (state[(I + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}
```

- Trước khi ăn, mỗi triết gia phải gọi hàm pickup(), ăn xong rồi thì phải gọi hàm putdown()

```
dp.pickup(i);
/* yum...yum...yum */
dp.putdown(i);
```