# HỆ ĐIỀU HÀNH
# (OPERATING SYSTEM)

Trình bày:Nguyễn Hoàng Việt
Khoa Công Nghệ Thông Tin
Đại Học Cần Thơ

# Chương 5: Đồng bộ hóa quá trình
## (Process Synchronization)

- Tổng quan
- Miền tương trục
- Giải pháp phần cứng
- Semaphores
- Các bài toán cổ điển về đồng bộ hóa
- Monitors

# Tổng quan (1)
## Đặt vấn đề

- Các quá trình hợp tác có thể chia sẻ không gian địa chỉ luận lý (mã và dữ liệu) → truy cập cạnh tranh lên dữ liệu chia sẻ → tình trạng không nhất quán dữ liệu (inconsistency).
- Việc duy trì **sự nhất quán dữ liệu** yêu cầu các cơ chế để đảm bảo sự thực thi một cách **có thứ tự** của các quá trình có hợp tác với nhau.
- Xét trường hợp bộ nhớ chia sẻ với vùng đệm có kích thước giới hạn (bounded-buffer) – Vấn đề Người sản xuất – Người tiêu thụ (Producer – Consumer Problem).

# Tổng quan (2)
## Vùng đệm có kích thước giới hạn (bounded-buffer)

Dữ liệu chia sẻ:

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

# Tổng quan (3)
## Vùng đệm có kích thước giới hạn

Producer process:

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

# Tổng quan (4)
## Vùng đệm có kích thước giới hạn

Consumer process:

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

# Tổng quan (5)
## Vùng đệm có kích thước giới hạn

- Các phát biểu (statements)

  **counter++;**
  **counter--;**

  phải được thực hiện môt cách nguyên tử (atomically).
- Nguyên tử: không thể chia nhỏ, thao tác phải hoàn thành trước khi bị ngắt.
- Cài đặt "**count++**" bằng ngôn ngữ máy:

  **register1 = counter**
  **register1 = register1 + 1**
  **counter = register1**
- Cài đặt "**count--**" bằng ngôn ngữ máy:

  **register2 = counter**
  **register2 = register2 – 1**
  **counter = register2**

# Tổng quan (6)
## Vùng đệm có kích thước giới hạn

- Nếu cả hai producer và consumer cố gắng cập nhật vùng đệm đồng thời, các phát biểu assembly có thể bị phủ lấp.
- Sự phủ lấp phụ thuộc vào cách producer và consumer được định thời.
- Giả sử **counter** hiện là 5. Phủ lấp có thể như sau:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

- Giá trị **counter** có thể là 4 hoặc 6 (kết quả đúng phải là 5).

# Tổng quan (7)
## Tình trạng đua tranh (Race Condition)

- **Tình trạng đua tranh**: là tình trạng mà vài quá trình cùng truy cập và thay đổi lên dữ liệu được chia sẻ và giá trị cuối cùng của dữ liệu chia sẻ phụ thuộc vào quá trình nào hoàn thành cuối cùng.

⇨ Để ngăn chặn tình trạng đua tranh, các quá trình cạnh tranh phải được đồng bộ hóa.

# Miền tương trục (1)
## (Critical Section)

- *n* quá trình đang cạnh tranh để sử dụng dữ liệu chia sẻ.
- Mỗi quá trình có một đoạn mã lệnh, gọi là ***miền tương trục***, truy cập dữ liệu được chia sẻ (slide 12).
- Vấn đề – đảm bảo rằng khi một quá trình đang chạy trong miền tương trục, không có một quá trình nào khác được đi vào miền tương trục của mình → tránh được tình trạng đua tranh.

# Miền tương trục (2)
## Giải pháp cho vấn đề miền tương trục

- Giải pháp cho vấn đề miền tương trục phải thỏa các yêu cầu:
- **Loại trừ hỗ tương (Mutual Exclusion)**. Nếu quá trình $P_i$ đang thực thi trong miền tương trục, thì không có quá trình nào khác có thể thực thi trong miền tương trục của chúng.
- **Tiến triển (Progress)**. Nếu không có quá trình nào đang thực thi trong miền tương trục của nó và tồn tại vài quá trình đang mong muốn được thực thi trong miền tương trục của chúng, thì việc lựa chọn cho một quá trình bước vào miền tương trục của nó không thể bị trì hoãn mãi được.
- **Chờ đợi hữu hạn (Bounded Wait)**. Không có quá trình nào phải chờ đợi vĩnh viễn để có thể đi vào miền tương trục của nó.

# Miền tương trục (3)
## Những cố gắng đầu tiên giải quyết bài toán MTT

- Chỉ có 2 quá trình, $P_0$ và $P_1$
- Cấu trúc tổng quát của quá trình $P_j$ (quá trình kia là $P_{1-j}$)

```
do {
    entry section
            critical section
    exit section
            remainder section
} while (1);
```

- Các quá trình có thể chia sẻ một số biến chung để đồng bộ hóa hành động của chúng.

# Miền tương trục (4)
## Giải pháp chờ đợi bận (Busy Waiting) - Giải thuật 1

- Các biến chung:
  - **int turn**;

    khởi đầu **turn = 0**
  - **turn = i** $\Rightarrow$ $P_i$ có thể bước vào miền tương trục của nó
- Quá trình $P_i$

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

- Giải pháp này đạt yêu cầu về loại trừ hỗ tương nhưng không tiến triển được.

# Miền tương trục (5)
## Giải pháp chờ đợi bận - Giải thuật 2

- Các biến chia sẻ:
  - **boolean flag[2]**;
    khởi đầu **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow P_i$ sẵn sàng bước vào miền tương trục của nó
- Quá trình $P_i$

```
do {
  flag[i] := true;
  while (flag[j]) ;
                critical section
          flag [i] = false;
                remainder section
} while (1);
```

- Đạt yêu cầu loại trừ hỗ tương, nhưng không tiến triển

# Miền tương trục (6)
## Giải pháp chờ đợi bận - Giải thuật Peterson

- Kết hợp các biến chia sẻ được sử dụng trong các giải thuật 1 và 2.
- Quá trình $P_i$

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] && turn == j) ;
            critical section
        flag [i] = false;
            remainder section
} while (1);
```

- Đạt cả 3 điều kiện.

# Miền tương trục (7)
## Giải pháp nhiều quá trình - Giải thuật Bakery

Miền tương trục cho n quá trình:

- Trước khi bước vào miền tương trục của mình, quá trình nhận được một con số (ticket). Quá trình nào nhận được con số nhỏ nhất sẽ có quyền bước vào miền tương trục.
- Nếu quá trình $P_i$ và $P_j$ nhận được cùng một số, và nếu $i<j$, thì $P_i$ được phục vụ trước.
- Bộ sinh luôn luôn sinh ra các con số theo thứ tự tăng; ví dụ 1,2,3,3,3,3,4,5...

# Miền tương trục (8)
## Giải pháp nhiều quá trình - Giải thuật Bakery

- Chú thích < dùng để so sánh các cặp (ticket #, process id #)
  - $(a,b) < (c,d)$ if $(a < c)$ or if $(a = c$ and $b < d)$
  - max $(a_0,\ldots, a_{n-1})$ là số $k$, sao cho $k \geq a_i$ for $i = 0, \ldots, n-1$
- Dữ liệu chia sẻ:

  **boolean choosing[n];**

  **int number[n];**
- Các dữ liệu trên được khởi tạo tương ứng là **false** và **0**

# Miền tương trục (9)
## Giải pháp nhiều quá trình - Giải thuật Bakery

Quá trình $P_i$:

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

# Giải pháp phần cứng (1)
## TestAndSet

- Đọc và sửa đổi nội dung của một word một cách nguyên tử:

```
boolean TestAndSet(boolean &target) {
        boolean rv = target;
        target = true;
        return rv;
}
```

- TestAndSet có tính nguyên tử (atomically): nếu 2 chỉ thị TestAndSet được thực thi cùng một lúc (mỗi chỉ thị trên một CPU khác nhau), chúng sẽ được thực thi tuần tự.

# Giải pháp phần cứng (2)
## TestAndSet

### Loại trừ hỗ tương với TestAndSet

- Dữ liệu chia sẻ:

  **boolean lock = false;**

- Quá trình $P_i$

```
do {
    while (TestAndSet(lock)) ;
      critical section
    lock = false;
       remainder section
} while(1);
```

# Giải pháp phần cứng (3)
## Swap

- Tự động hoán chuyển (swap) hai biến:

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

- Swap có tính nguyên tử.

# Giải pháp phần cứng (4)
## Swap

Loại trừ hỗ tương với Swap

- Dữ liệu chia sẻ (khởi tạo là **false**):

  **boolean lock;**

- Quá trình $P_i$

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

# Giải pháp phần cứng (5)
## Lọai trừ hỗ tương và chờ đợi hữu hạn

- Các giải thuật trên không đạt điều kiện chờ đợi hữu hạn.
- Giải thuật đạt được các yêu cầu về miền tương trục dùng TestAndSet
- Dữ liệu chia sẻ (khởi tạo **false)**

  **boolean lock;**

  **boolean waiting[n];**

# Giải pháp phần cứng (6)
## Lọai trừ hỗ tương và chờ đợi hữu hạn

- Quá trình $P_i$

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
       key = TestAndSet(lock);
    waiting[i] = false;
            critical section
    j = (i + 1)%n;
    while ((j != i) && !waiting[j])
        j = (j + 1)%n;
    if (j == i)
        lock = false;
      else
        waiting[j] = false;
            remainder section
} while(1);
```

# Hiệu báo (Semaphore) (1)
## Định nghĩa

- Các giải pháp chờ đợi bận làm hao phí thời gian của CPU. Semaphore là công cụ đồng bộ hóa không gây ra tình trạng chờ đợi bận.
- Semaphore cho phép quá trình chờ đợi vào miền tương trục sẽ ngủ/nghẽn (sleep/blocked) và sẽ được đánh thức (wakeup) bởi một quá trình khác.
- Được thực hiện bằng 2 lời gọi hệ thống: block và wakeup().
- Semaphore *S:* là một biến integer, chỉ có thể được truy cập thông qua hai thao tác nguyên tử:
  - *wait*(*S*):

    **while *S*≤ 0 do *no-op*;**

    ***S*--;**
  - *signal*(*S*):

    ***S*++;**

# Hiệu báo (2)
## Miền tương trục của *n* quá trình

- Dữ liệu chia sẻ:

  **semaphore mutex;** Khởi tạo **mutex = 1**

- Quá trình $P_i$:

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

# Hiệu báo (3)
## Cài đặt Semaphore

- Định nghĩa một semaphore như là một record

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

- Giả sử ta đã có hai thao tác được cung cấp bởi hệ điều hành như những lời gọi hệ thống cơ bản:

  **block** – ngưng tạm thời quá trình gọi thao tác này.

  **wakeup(*P*)** khởi động lại quá trình đã bị blocked **P**.

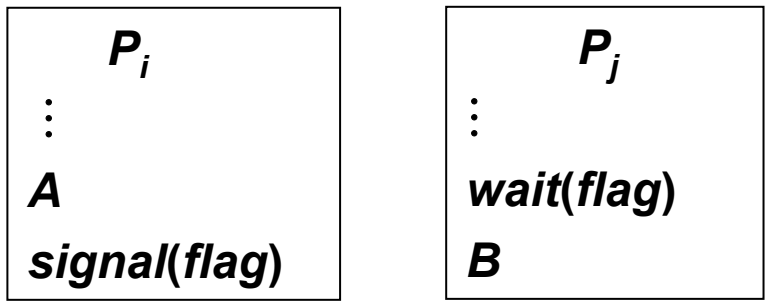
# Hiệu báo (4)
## Cài đặt Semaphore

- Các thao tác trên Semaphore bây giờ được định nghĩa 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);
        }
 }
```

# Hiệu báo (5)
## Semaphore như là công cụ đồng bộ hóa tổng quát

- Thực thi *B* trong $P_j$ chỉ sau khi *A* đã được thực thi trong $P_i$
- Sử dụng semaphore *flag* khởi tạo với giá trị 0
- Code:



# Hiệu báo (6)
## Khóa chết (Deadlock) và đói CPU (Starvation)

- **Khóa chết** – hai hoặc nhiều quá trình đang chờ đợi vô hạn một sự kiện nào đó, mà sự kiện đó chỉ có thể được tạo ra bởi một trong các quá trình đang chờ đợi khác.

| $P_0$ | $P_1$ |
|---|---|
| *wait*(*S*); | *wait*(*Q*); |
| *wait*(*Q*); | *wait*(*S*); |
| ⋮ | ⋮ |
| *signal*(*S*); | *signal*(*Q*); |
| *signal*(*Q*); | *signal*(*S*); |

*S* và *Q* được khởi tạo giá trị 1

- **Sự đói CPU** – bị nghẽn (block) không hạn định. Một quá trình có thể không bao giờ được xóa khỏi hàng đợi trong semaphore.

# Hiệu báo (7)
## Hai kiểu semaphore

- Semaphore đếm (counting semaphore): giá trị integer có thể trải qua nhiều miền giá trị.
- Semaphore nhị phân (binary semaphore): giá trị integer chỉ nhận một trong hai giá trị 0 và 1; có thể cài đặt đơn giản hơn.
- Có thể coi semephore nhị phân là trường hợp đặc biệt của semaphore đếm.

# Các bài toán đồng bộ hóa cổ điển (1)

- Bài toán người sản xuất-người tiêu dùng (Bounded-Buffer hay Producer-Consumer).
- Bài toán Readers and Writers.
- Bài toán “5 nhà triết gia ăn tối” (Dining-Philosophers).

# Các bài toán đồng bộ hóa cổ điển (2)
## Bounded-Buffer

- Hai quá trình producer và consumer chia sẻ vùng đệm có kích thước *n.*
- Dữ liệu chia sẻ:
  - Semaphore **mutex**: dùng cho loại trừ hỗ tương, khởi tạo 1.
  - Các Semaphore **empty** và **full**: dùng để đếm số khe trống và đầy. Khởi tạo **empty = n** và **full = 0**.

# Các bài toán đồng bộ hóa cổ điển (3)
## Bounded-Buffer – Quá trình Producer

```
do {
                    …
        produce an item in nextp
                    …
        wait(empty);
        wait(mutex);
                    …
        add nextp to buffer
                    …
        signal(mutex);
        signal(full);
} while (1);
```

# Các bài toán đồng bộ hóa cổ điển (4)
## Bounded-Buffer – Quá trình Consumer

```
do {
        wait(full)
        wait(mutex);
                …
        remove an item from buffer to nextc
                …
        signal(mutex);
        signal(empty);
                …
        consume the item in nextc
                …
} while (1);
```

# Các bài toán đồng bộ hóa cổ điển (5)
## Bài toán Bộ đọc-Bộ ghi (Readers-Writers)

- Là một đối tượng dữ liệu (tập tin, mẩu tin) được chia sẻ giữa nhiều quá trình đồng hành.
- Một số quá trình chỉ đọc (readers), một số khác cần cập nhật, đọc và ghi (writers) trên đối tượng được chia sẻ.
- Các bộ đọc, ghi phải có loại trừ hỗ tương trên đối tượng chia sẻ.
- Dữ liệu chia sẻ:
  - Biến **readcount**: ghi vết số quá trình đang đọc đối tượng. Khởi tạo **readcount = 0**.
  - Semaphore **mutex:** dùng cho loại trừ hỗ tương khi cập nhật **readcount**. Khởi tạo **mutex = 1.**
  - Semaphore **wrt:** dùng loại trừ hỗ tương cho các bộ ghi. Khởi tạo **wrt = 1**. Semaphore này cũng được dùng để ngăn cấm các bộ ghi truy cập vào đối tượng chia sẻ khi nó đang được đọc.

# Các bài toán đồng bộ hóa cổ điển (6)
## Bài toán Readers-Writers – Quá trình Writers

**wait(wrt);**

**...**

writing is performed

**...**

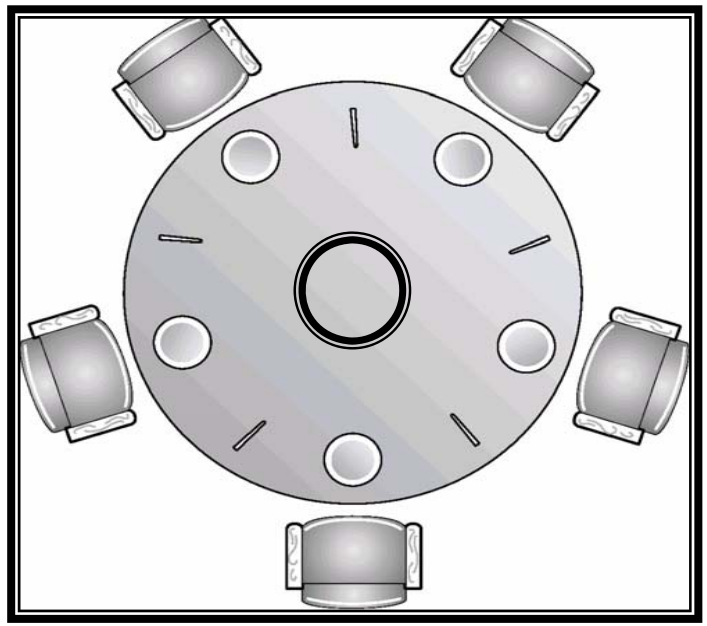
**signal(wrt);**

# Các bài toán đồng bộ hóa cổ điển (7)
## Bài toán Readers-Writers – Quá trình Readers

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

# Các bài toán đồng bộ hóa cổ điển (8)
## Bài toán các triết gia ăn tối (Dining-Philosophers)

- Năm triết gia ngồi trên bàn tròn, giữa là bát cơm và 5 chiếc đủa như hình, vừa ăn vừa suy nghĩ.
- Khi suy nghĩ, triết gia không giao tiếp với các triết gia khác.
- Khi đói, ông ta cố gắng chọn 2 chiếc đũa gần nhất (giữa ông ta và 2 láng giềng). Ông ta chỉ có thể lấy được 1 chiếc đũa tại mỗi thời điểm, và không thể lấy được đũa đang được dùng bởi láng giềng.



- Khi có 2 chiếc đũa, triết gia ăn và chỉ đặt đũa xuống khi ăn xong, sau đó suy nghĩ tiếp.
- Dữ liệu chia sẻ: semaphore **chopstick[5];** Khởi đầu, các giá trị là 1.

# Các bài toán đồng bộ hóa cổ điển (9)
## Bài toán các triết gia ăn tối – Giải thuật

- Philosopher *i*:

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

# Các bài toán đồng bộ hóa cổ điển (10)
## Bài toán các triết gia ăn tối – Ngăn khóa chết

- Giải thuật bảo đảm không có 2 láng giềng ăn cùng lúc, nhưng có thể gây khóa chết (tình huống 5 triết gia cùng đói và mỗi người cùng lấy đũa bên trái).
- Các giải pháp khả dụng:
  - Cho phép nhiều nhất 4 triết gia cùng ngồi trên bàn.
  - Cho phép một triết gia lấy đũa chỉ nếu cả hai chiếc đũa đều sẵn dùng.
  - Dùng giải pháp bất đối xứng. Ví dụ triết gia lẻ lấy đũa trái trước, rồi đến đũa phải, trong khi triết gia chẵn thao tác ngược lại.

# Monitors (1)
## Định nghĩa

- Cấu trúc đồng bộ hóa cấp cao cho phép chia sẻ an toàn một kiểu dữ liệu trừu tượng nào đó giữa nhiều quá trình cạnh tranh.
- Kiểu monitor bao gồm:
  - Tập hợp các thao tác được định nghĩa bởi lập trình viên, cung cấp loại trừ hỗ tương bên trong monitor.
  - Các biến mà giá trị của nó định nghĩa trạng thái của một thể hiện (instance) của monitor, cùng với thân của các thủ tục hoặc hàm thao tác trên các biến đó.
- Các thủ tục bên trong monitor chỉ có thể truy xuất các biến được khai báo cục bộ bên trong monitor và các tham số chính thức của nó.
- Các biến cục bộ chỉ có thể được truy xuất chỉ bởi các thủ tục cục bộ.

# Monitors (2)
## Định nghĩa

**monitor *monitor-name***

**{**

shared variable declarations

**procedure body** $P_1$ **(…) {**

. . .

**}**

**procedure body** $P_2$ **(…) {**

. . .

**}**

**procedure body** $P_n$ **(…) {**

. . .

**}**

**{**

initialization code

**}**

**}**

# Monitors (3)
## Định nghĩa

- Để cho phép một quá trình chờ đợi trong monitor, một biến điều kiện phải được khai báo như sau:

  **condition x, y;**

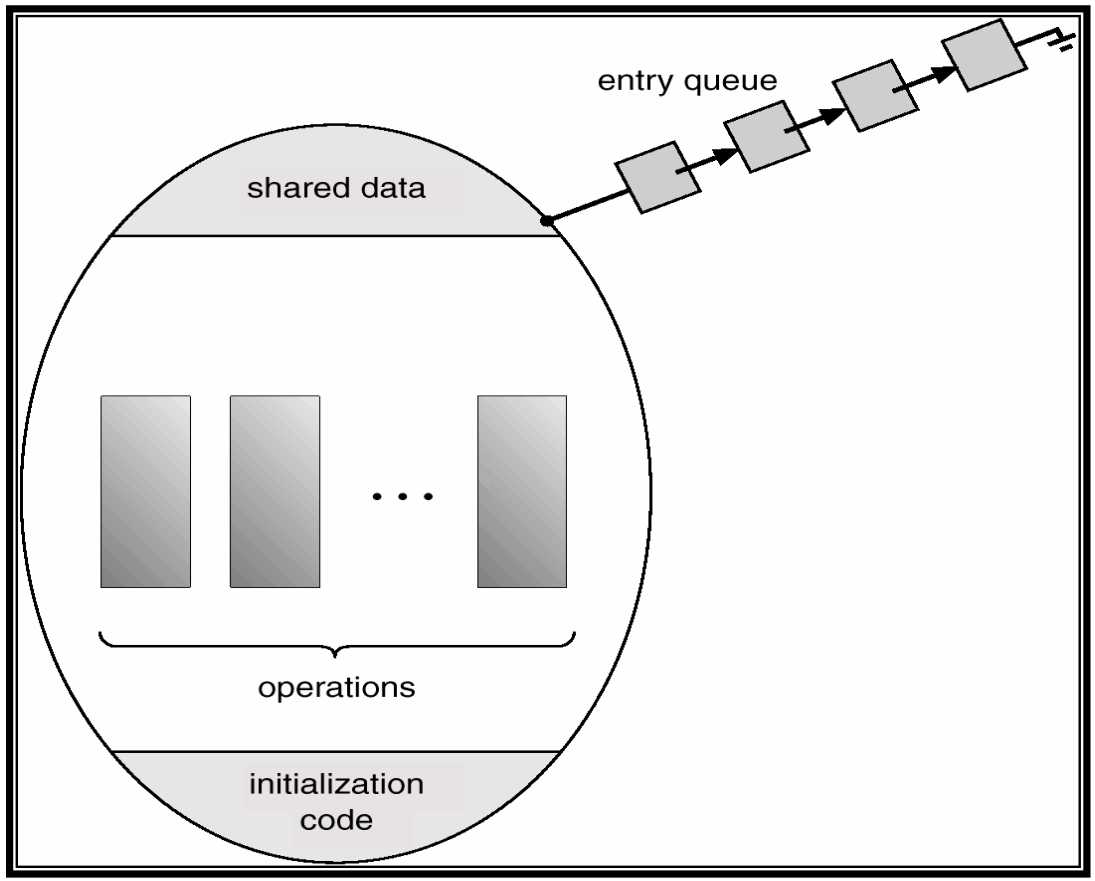
- Biến điều kiện chỉ có thể được sử dụng với hai thao tác **wait** và **signal**.
  - Thao tác:

    **x.wait();**

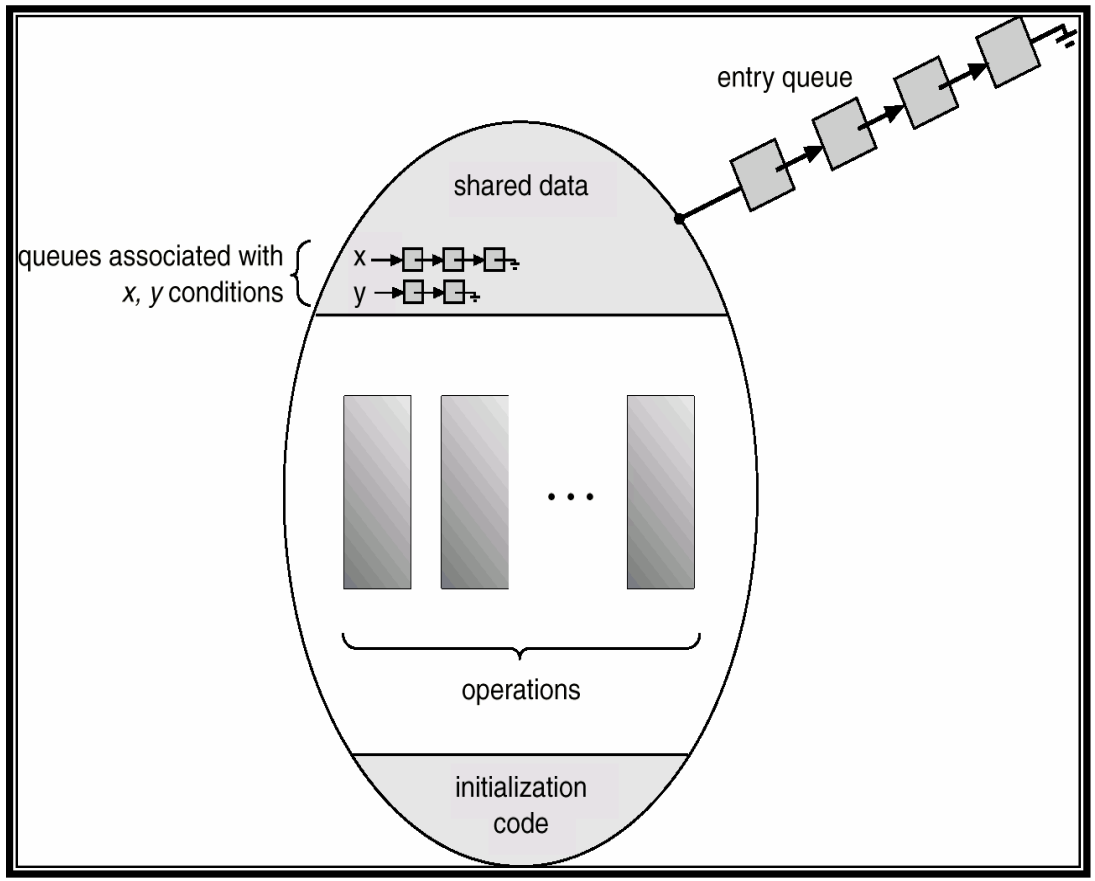
    có nghĩa là quá trình đang gọi thao tác này bị ngủ cho đến khi một quá trình khác gọi

    **x.signal();**
  - Thao tác **x.signal** khởi động lại chính xác một quá trình đang ngủ để chờ đợi trên biến x. Nếu không có quá trình nào đang ngủ trên x, thì thao tác **signal** không gây ảnh hưởng gì cả.

# Monitors (4)
## Cái nhìn một cách có sơ đồ về Monitor



# Monitors (5)
## Monitor với các biến điều kiện



# Monitors (6)
## Các triết gia ăn tối dùng monitor

Giải thuật dưới đây cho phép triết gia có thể lấy đũa chỉ nếu cả hai chiếc đũa đều sẵn dùng.

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

# Monitors (7)
## Các triết gia ăn tối dùng monitor

```
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);
}
```

# Monitors (8)
## Các triết gia ăn tối dùng monitor

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

Triết gia i sẽ dùng monitor dp theo thứ tự sau:

```
dp.pickup(i);
eat();
dp.putdown()
```