Chương cũ · Chỉ có 1 CPU nên cần chia sẻ ·

CT107. Hệ Điều Hành
Chương 5. Đồng Bộ Hóa Tiến Trình

Tại sao lại cần?

Giảng viên: Trần Công Án (tcan@cit.ctu.edu.vn)

Bộ môn Mạng máy tính & Truyền thông

Khoa Công Nghệ Thông Tin & Truyền Thông

Đại học Cần Thơ

2015

Mục Tiêu

Giới thiệu vấn đề miền tương trục và các giải pháp để giải quyết vấn đề miền tương trục, nhằm đảm bảo sự nhất quán của dữ liệu được chia sẻ giữa các tiến trình cạnh tranh trong miền tương trục.

Nội Dung

Cạnh Tranh Và Sự Nhất Quan Dữ Liệu

- Các tiến trình thực thi đồng thời, chia sẻ dữ liệu dùng chung có thể dẫn đến tình trạng không nhất quán (inconsistency) của dữ liệu.
- Nhất quán = đúng đắn và chính xác; tùy thuộc vào ngữ cảnh, giao dịch.
- Có 2 lý do chính để thực hiện đồng thời (cạnh tranh) các tiến trình:
  - Tăng hiệu suất sử dụng tài nguyên hệ thống.
  - Giảm thời gian đáp ứng trung bình của hệ thống.
- Việc duy trì sự nhất quán của dữ liệu yêu cầu một cơ chế để đảm bảo sự thực thi một cách có thứ tự của các tiến trình có hợp tác với nhau.

Ví Dụ 1 – Giao Dịch Cạnh Tranh

- Cho hai giao dịch:
  - T1: A mua hàng trị giá 50$ của P (50$: A → P)
  - T2: B mua hàng trị giá 100$ của P (100$: B → P)
- Khởi tạo ban đầu: A=500; B=500; P=1000

T1	T2
R(A)	R(B)
A = A - 50	B = B - 100
W(A)	W(B)
R(P)	R(P)
P = P + 50	P = P + 100
W(P)	W(P)

- Yêu cầu về tính nhất quán: (A + B + P) không đổi; cụ thể hơn:
  - giá trị A, B, P sau khi thực hiện T1, T2 là: A=450; B=400; P=1150
- **Nhận xét**:
  - Nếu thực hiện tuần tự T1 → T2 hoặc T2 → T1, dữ liệu sẽ nhất quán.
  - Nếu thực hiện cạnh tranh (đồng thời), dữ liệu sẽ nhất quán???

Ví Dụ 1 – Giao Dịch Cạnh Tranh

T1	T2	A/B	P
R(A)			
A = A - 50			
W(A)		450	
	R(B)		
	B = B - 100		
	W(B)	400	
R(P)			
P = P + 50			
	R(P)		
W(P)			1050
	P = P + 100		
	W(P)		1100

Schedule 1

T1	T2	A/B	P
R(A)			
A = A – 50			
	R(B)		
	B = B - 100		
	W(B)	400	
W(A)		450	
R(P)			
P = P + 50			
W(P)			1050
	R(P)		
	P = P + 100		
	W(P)		1150

Schedule 2

Ví Dụ 2 – Bài Toán Nhà SX - Người Tiêu Thụ

- **Dữ liệu chia sẻ** (kho hàng, có giới hạn):

```
#define struct {
   ...
} item;

item buffer[BUFFER_SIZE];
int in_item = 0;
int out_item = 0;
int counter = 0;
```

Ví Dụ 2 – Bài Toán Nhà SX - Người Tiêu Thụ

- **Nhà sản xuất (S)**:

```
while (true) {
  /* produce an item in next produced */

  while (counter == BUFFER SIZE) ; /* do nothing */

  buffer[in_item] = next_produced;
  in_item = (in_item + 1) % BUFFER SIZE;
  counter++;
}
```

Ví Dụ 2 – Bài Toán Nhà SX - Người Tiêu Thụ

- **Người tiêu thụ (T)**:

```
while (true) {
  while (counter == 0) ; /* do nothing */

  next_consumed = buffer[out_item];
  out_item = (out_item + 1) % BUFFER SIZE;
  counter--;

  /* consume the item in next consumed */
}
```

Ví Dụ 2 – Bài Toán Nhà SX - Người Tiêu Thụ

- Dữ liệu chia sẻ giữa producer và consumer: biến `counter`.
- Điều kiện để đảm bảo tính nhất quán của biến `counter`: các câu lệnh `counter++` và `counter--` phải được thực thi một cách "**nguyên tử**" và "**cô lập**".
  - **Nguyên tử**: không thể chia nhỏ (hoặc "hoặc tất cả, hoặc không")
  - **Cô lập**: các t/trình không truy xuất các g/trị không nhất quán của nhau
- Vấn đề gì có thể xảy ra đối với `counter`?
  - `counter++` trong ngôn ngữ máy:

    ```
    register1 = counter
    register1 = register1 + 1
    counter = register1
    ```

  - `counter--` trong ngôn ngữ máy:

    ```
    register2 = counter
    register2 = register2 - 1
    counter = register2
    ```

Ví Dụ 2 – Bài Toán Nhà SX - Người Tiêu Thụ

- Xét một lịch trình thực thi xen kẽ (cạnh tranh) của hai tiến trình S và T trong hệ thống, với giá trị counter = 5:

  S1: `register1 = counter` (register1 = 5)

  S2: `register1 = register1 + 1` (register1 = 6)

  P1: `register2 = counter` (regster2 = 5)

  P2: `register2 = register2 - 1` (register2 = 4)

  S3: `counter = register1` (counter = 6)

  P3: `counter = register2` (counter = 4)

  ⇒ giá trị cuối cùng của `counter` là 4 (hoặc 6 nếu S3 sau P3), trong khi giá trị nhất quán của `counter` trong trường hợp này là 5.

Tình Trạng "Tranh Đua" (Race Condition)

- Là tình trạng mà nhiều tiến 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 hoàn thành sau cùng.
  - giá trị của P trong ví dụ 1
  - hoặc giá trị biến `counter` trong ví dụ 2
- Tình trạng tranh đua có thể dẫn đến tình trạng không nhất quán.
- Để ngăn chặn tình trạng tranh đua, các quá trình cạnh tranh cần phải được đồng bộ hóa (synchronize).

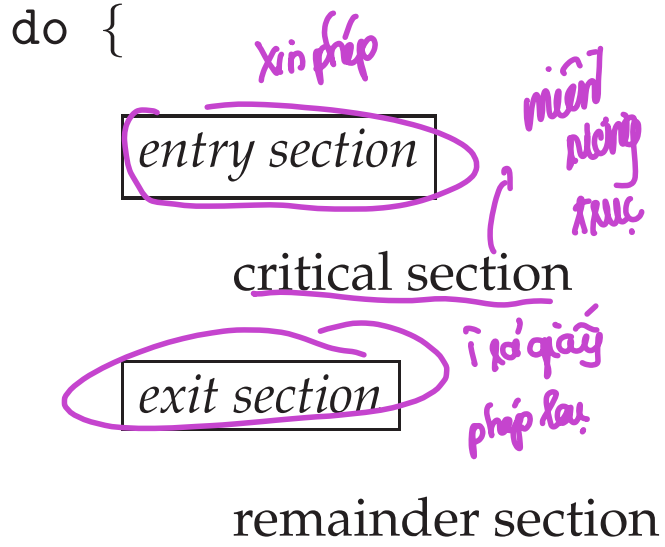
Vấn Đề Miền Tương Trục (CSP)

- Xét 1 hệ thống có n tiến trình đang cạnh tranh $\{P_0, P_1, \ldots, P_{n-1}\}$
- **Miền tương trục (critical section)**: là một đoạn mã lệnh của các tiến trình có chứa các hành động truy cập dữ liệu được chia sẻ như: thay đổi các biến dùng chung, cập nhật CSDL, ghi tập tin, ...

  $\Rightarrow$ Để tránh tình trạng tranh đua, các hệ thống phải đảm bảo khi một tiến trình đang trong miền tương trục, không có một tiến trình nào khác được phép chạy trong miền tương trục của nó.
- **Vấn đề miền tương trục (critical-section problem)**: Thiết kế các giao thức để các tiến trình có thể sử dụng để hợp tác/cạnh tranh với nhau.

Vấn Đề Miền Tương Trục (CSP)

- Cần phải xác định được phần entry section và exit section.
- Mỗi tiến trình phải *xin phép* để được vào miền tương trục (đi qua vùng entry section), và sau đó thoát khỏi miền tương trục (đi qua vùng exit section) và thực hiện phần còn lại (remainder section).
- Giải pháp cho vấn đề miền tương trục tương đối phức tạp với với các hệ thống định thời trưng dụng.



Yêu Cầu Đối Với Các Giải Pháp Cho CSP

- Một giải pháp cho vấn đề miền tương trục phải thỏa 3 yêu cầu:
  1. **Loại trừ hỗ tương** (mutual exclusion): Nếu 1 t/trình đang thực thi trong miền tương trục, không một tiến trình nào khác được đi vào miền tương trục của chúng.
  2. **Tiến triển** (progress): Nếu không có tiến trình nào đang thực thi trong miền tương trục và tồn tại tiến trình đang chờ được thực thi trong miền tương trục của chúng, thì việc lựa chọn cho một quá trình bước vào miền tương trục không thể bị trì hoãn vô hạn.
  3. **Chờ đợi hữu hạn** (bounded wait): Mỗi t/trình chỉ phải chờ để được vào miền tương trục trong một khoảng t/gian có hạn định (không xảy ra tình trạng "chết đói" – starvation).

Phân Loại Các Giải Pháp

- **Các giải pháp phần mềm**: dựa trên các giải thuật phần mềm, như:
  - Cho trường hợp chỉ có 2 tiến trình cạnh tranh:
    - Giải thuật 1 và 2
    - Giải thuật Peterson (Peterson's algorithm)
  - Cho trường hợp có $n \geq 2$ tiến trình cạnh tranh:
    - Giải thuật Bakery
- **Các giải pháp phần cứng**:
  - Lệnh vô hiệu hóa ngắt (disable interrupt)
  - Lệnh máy đặc biệt: TestAndSet

GT1 – Giải Thuật Chờ Bận 1 (Busy Wait)

- Điều khiển cạnh tranh giữa 2 tiến trình P_i và P_j.
- Dùng 1 biến khóa chia sẻ để đ/khiển việc vào miền tương trục.
  - `int turn = 0; //initialise turn=0`
  - `turn = i` $\Rightarrow$ P_i có thể vào miền tương trục.
- Tổ chức đoạn mã của 1 tiến trình P_i:

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

- **Nhận xét**:
  - Loại trừ hỗ tương: ✓
  - Tiến triển: ✗

  $\Rightarrow$ **Không thỏa yêu cầu**.

GT2 – Giải Thuật Chờ Bận 2 (Busy Wait)

- Dùng các biến khóa riêng để đ/khiển việc vào miền tương trục.
  - `boolean flag[2];`
  - Khởi tạo: `flag[0] = flag[1] = false`
  - `flag[i] = true` $\Rightarrow P_i$ sẵn sàng vào miền tương trục.
- Tổ chức đoạn mã của 1 tiến trình P_i:

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

- **Nhận xét**:
  - Loại trừ hỗ tương: ✓
  - Tiến triển: ✗

    $\Rightarrow$ **Giống giải thuật 1**.
  - Tuy nhiên, mức độ cạnh tranh cao hơn.

Giải Thuật Peterson

- Kết hợp cả biến khóa chia sẻ (GT1) và biến khóa riêng (GT2):
  - `int turn;` //turn = i: P_i được phép vào miền tương trục.
  - `boolean flag[2];` //flag[i] = true: P_i sẵn sàng vào miền tương trục.
- Tổ chức đoạn mã của 1 tiến trình P_i:

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

- **Nhận xét**:
  - Loại trừ hỗ tương: ✓
  - Tiến triển: ✓
  - Chờ đợi hữu hạn: ✓

Giải Thuật Bakery

- Miền tương trục cho n tiến trình:
  - Mỗi t/trình sẽ nhận được 1 số trước khi vào miền tương trục.
  - Tiến trình có số nhỏ nhất sẽ có quyền ưu tiên cao nhất.
  - Nếu hai tiến trình P_i và P_j nhận được cùng một số, nếu $i < j$ thì P_i được phục vụ trước.
  - Bộ sinh số luôn sinh các số theo thứ tự tăng, ví dụ: 1, 2, 3, 3, 3, 4, ...
- Dữ liệu chia sẻ:

```
boolean choosing[n]

int number[n]
```

  - Khởi tạo: tất cả các phần tử của `choosing = false` và `number = 0`.

Giải Thuật Bakery Cho Tiến 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)) ;
  } //for
    critical section
  number[i] = 0;
    remainder section
} while (true);
```

- (number #1, i) < (number #2, j) nếu (number #1 < number #2) hoặc (number #1 = number #2) AND (i < j)

Đồng Bộ Hóa Bằng Phần Cứng

- Ý tưởng cơ bản của hầu hết các giải pháp bằng phần cứng là bảo vệ miền tương trục bằng khóa.
- **Giải pháp đơn giản nhất**: vô hiệu hóa các ngắt – cho phép tiến trình người dùng vô hiệu hóa các ngắt khi vào miền tương trục, cho đến khi t/trình ra khỏi miền tương trục.
  - Không có tiến trình nào khác có thể thực thi khi một tiến trình đã vào miền tương trục $\Rightarrow$ tránh tình trạng cạnh tranh.
  - Chỉ có thể áp dụng cho hệ thống không trưng dụng.
  - Không khả thi cho hệ thống đa xử lý (vô hiệu hóa các ngắt trên hệ thống đa xử lý mất nhiều chi phí $\Rightarrow$ hiệu năng giảm).

Đồng Bộ Hóa Bằng Phần Cứng

- **Các giải pháp khả thi hơn**: cung cấp các thao tác nguyên tử từ phần cứng (atomic hardware instructions):
  - `test_and_set`
  - `compare_and_swap`
- Các chỉ thị này được hỗ trợ trong các hệ thống hiện đại, có nhiều bộ xử lý.
- Nếu các chỉ thị nguyên tử được thực thi cùng lúc (có thể trên các CPU khác nhau) thì chúng sẽ được thực thi tuần tự (theo một thứ tự bất kỳ).

Chỉ Thị test_and_set

- Cho phép đọc và sửa nội dung của một word một cách nguyên tử.
- Định nghĩa của chỉ thị test_and_set:

```
boolean test_and_set(boolean *target) {
  boolean rv = *target;
  *target = true;

  return rv;
}
```

Loại Trừ Hỗ Tương Với test_and_set

- Dữ liệu chia sẻ: `boolean lock = false;`
- Tiến trình P_i:

```
do {
  while (test_and_set(lock)) ; //do nothing
    critical section
  lock = false;
    remainder section
} while (true);
```

- Tính chất:
  - Loại trừ hỗ tương và tiến triển: ✓
  - Chờ đợi hữu hạn: ✗

Chỉ Thị compare_and_swap

- Dùng để hoán chuyển (swap) hai biến.
- Định nghĩa của chỉ thị swap:

```
void swap(boolean &oldValue, boolean expected, boolean newValue) {
  boolean temp = *oldValue;

  if (*oldValue == expected)
   *oldValue = newValue;

  return temp;
}
```

- Chỉ thị được thực thi một cách nguyên tử.

Loại Trừ Hỗ Tương Với compare_and_swap

- Dữ liệu chia sẻ: `boolean lock = false;`
- Tiến trình P_i:

```
do {
  while (compare_and_swap(&lock, false, true) != false) ;
      critical section
    lock = false;
      remainder section
} while (true);
```

- Tính chất:
  - Loại trừ hỗ tương: ✓
  - Chờ đợi hữu hạn: ✗

Chờ Đợi Hữu Hạn Với test_and_set

- Dữ liệu chia sẻ:

```
boolean lock;
boolean waiting[n];
```

- Khởi tạo:

```
lock = false;
waiting[1..n] = false;
```

- Tiến trình P_i:

```
do {
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
    key = test_and_set(&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 (true);
```

Hiệu Báo (Semaphores)

- Semephore là công cụ đồng bộ hóa tránh được chờ đợi bận:
  - Tiến trình chờ đợi vào miền tương trục sẽ ngủ/nghẽn.
  - Tiến trình đang ngủ/nghẽn sẽ được đánh thức bởi các tiến trình khác.
- Semaphore S: là một biến integer, được truy cập qua 2 thao tác nguyên tử:
  - `wait(S)`:

    ```
    while (S ≤ 0) ; //do nothing (busy wait)
    S--;
    ```

  - `signal(S)`: `S++`
- Ưu điểm: ít phức tạp hơn các giải pháp khác.

Đồng Bộ Hóa Dùng Semaphore

- Các loại semaphore:
  - Semaphore đếm (counting semaphore): giá trị semaphore không giới hạn.
  - Semaphore nhị phân (binary semaphore): có giá trị 0 hoặc 1, còn gọi là mutex lock; cài đặt đơn giản hơn.
- Dữ liệu chia sẻ:

```
semaphore mutex = 1;
```

- Tiến trình P_i:

```
do {
  wait(mutex);
     critical section
  singal(mutex);
     remainder section
} while (true);
```

Đồng Bộ Hóa Dùng Semaphore – Ví Dụ

- Có 2 tiến trình: P_1 với lệnh S_1 và P_2 với lệnh S_2.
- Yêu cầu: S_2 phải thực thi sau khi S_1 hoàn thành.
- Cài đặt:
  - Sử dụng semaphore `flag`, khởi tạo với giá trị 0.

Tiến trình P_i:

```
⋮
S1;
signal(flag);
```

Tiến trình P_2:

```
⋮
wait(flag);
S2
```

Cài đặt Semaphore Không Chờ Đợi Bận

- Thao tác `signal(S)` và `wait(S)` không tránh được chờ đợi bận.
- Cần một cách tiếp cận khác để tránh tình trạng này:
  - Cấu trúc dữ liệu semaphore:

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

  - Hai thao tác cần được cung cấp bởi HĐH:
    - `block()`: ngưng (block) tạm thời tiến trình gọi chỉ thị này.
    - `wakeup(P)`: khởi động lại tiến trình đang bị ngưng P.

Các Thao Tác Semaphore Không Chờ Đợi Bận

- Các thao tác semaphore `wait(S)` và `signal(S)` được định nghĩa lại như sau:

```
void wait(semaphore S) {
  S.value--;
  if (S.value < 0) {
    add this process into
        S.L;
    block();
  } //if
} //wait()
```

```
void signal(semaphore S) {
  S.value++;
  if (S.value ≤ 0) {
    remove a process P from
        S.L;
    wakeup(P);
  } //if
} //signal()
```

  - Giá trị semaphore có thể âm (trong tr/hợp chờ đợi bận thì không).
  - Giá trị âm thể hiện số lượng tiến trình đang chờ đợi trên semaphore.

Cài Đặt `wait()` Và `signal()`

- **Yêu cầu**: không thể có nhiều hơn 1 tiến trình thực thi `wait()` hoặc `signal()` đồng thời $\Rightarrow$ **critical-section problem**.
  - Hệ thống đơn xử lý: vô hiệu hóa các ngắt khi thực thi `wait()` và `signal()`.
  - Hệ thống đa xử lý: dùng chờ đợi bận (spinlock) hay các lệnh nguyên tử (e.g. `compare_and_swap()`).

    $\Rightarrow$ Không hoàn toàn loại bỏ được chờ đợi bận: di chuyển chờ đợi bận vào bên trong chỉ thị `wait()` và `signal()` – thường là ngắn – nên thời gian chờ đợi bận được giảm đi.

Khóa Chết Và Chết Đói

- **Khóa chết**: hai hay nhiều t/trình chờ đợi vô hạn một sự kiện, mà sự kiện đó chỉ có thể tạo ra bởi một trong các t/trình đang chờ đợi khác.
- Ví dụ: cho hai semaphore S và Q được khởi tạo bằng 1

P_0

```
wait(S);
wait(Q);
:
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
:
signal(Q);
signal(S);
```

- **Chết đói**: tiến trình bị ngưng (block) không hạn định (không bao giờ được xóa khỏi hàng đợi semaphore).

Các Bài Toán Đồng Bộ Hóa

1. Bài toán Nhà sản xuất – Người tiêu dùng với vùng đệm giới hạn (Producer–Consummer with Bounded-Buffer).
2. Bài toán Bộ đọc – Bộ ghi (Readers–Writers).
3. Bài toán Các triết gia ăn tối (Dining-Philosophy).

Bài Toán Nhà Sản Xuất – Người Tiêu Dùng

- Hai tiến trình (nhà SX, người TD) chia sẻ chung vùng đệm có kích thước n (xem mô tả trong phần giới thiệu).
- Dữ liệu chia sẻ đồng bộ hóa:
  - Semaphore `mutex`: dùng để loại trừ hỗ tương, khởi tạo bằng 1.
  - Semaphore `empty` và `full`: dùng để đếm số khe trống và đầy trên bộ đệm, khởi tạo `empty = n` và `full = 0`.

Cấu Trúc Tiến Trình Nhà Sản Xuất

```
do {
  ...
  produce an item: "next_produced"
  ...
  wait(empty);
  wait(mutex);
  ...
  add "next_produced" to the buffer
  ...
  signal(mutex);
  signal(full);
} while (true);
```

Cấu Trúc Tiến Trình Người Tiêu Dùng

```
do {
  wait(full);
  wait(mutex);
  ...
  remove an item from the buffer: "next_consumed"
  ...
  signal(mutex);
  signal(empty);
  ...
  consume the item "next_consumed"
  ...
} while (true);
```

Bài Toán Bộ Đọc – Bộ Ghi

- Nhiều tiến trình thực thi đồng thời cạnh tranh một đối tượng dữ liệu (tập tin, mẫu tin):
  - Bộ đọc (readers): tiến trình chỉ đọc dữ liệu.
  - Bộ ghi (writers): tiến trình đọc và ghi dữ liệu.
- Vấn đề:
  - Nhiều readers có thể đọc dữ liệu chia sẻ đồng thời.
  - Chỉ có 1 writer được phép truy cập dữ liệu chia sẻ tại 1 thời điểm.

$\Rightarrow$ Các bộ đọc, ghi phải loại trừ hỗ tương trên các đối tượng chia sẻ.

Giải Pháp

- Dữ liệu chia sẻ:
  - Đối tượng dữ liệu chia sẻ (tập tin, mẫu tin, . . .).
  - Biến `read_count`: đếm số tiến trình đang đọc, khởi tạo bằng 0.
  - Semaphore `mutex`: dùng để loại trừ hỗ tương khi cập nhật biến `read_count`, khởi tạo bằng 1.
  - Semaphore `rw_mutex`: dùng để loại trừ hỗ tương cho các bộ ghi, khởi tạo bằng 1. Semaphore này cũng được sử dụng bởi các bộ đọc đầu tiên vào vùng tương trục và bộ đọc cuối cùng ra khỏi vùng tương trục.

Cấu Trúc Các Tiến Trình Đọc – Ghi

⊛ Bộ ghi

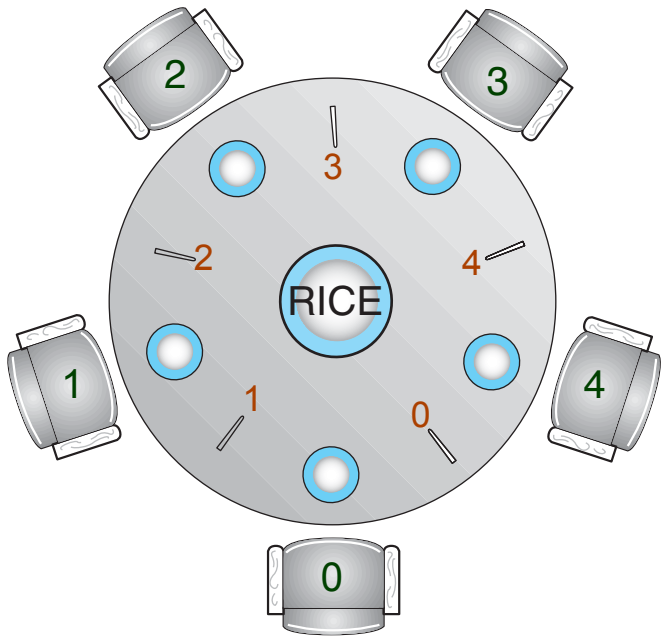
```
do {
  wait(rw_mutex);
    ...
  writing is performed
    ...
  signal(rw_mutex);
} while (true);
```

⊛ Bộ đọc

```
do {
  wait(mutex);
  read_count++;
  if (read_count == 1)
    wait(rw_mutex);
  signal(mutex);
    ...
  reading is performed
    ...
  wait(mutex);
  read_count--;
  if (read_count == 0)
    signal(rw_mutex);
  signal(mutex);
} while (true);
```

Bài Toán Năm Triết Gia Ăn Tối

- Các triết gia ngồi suy nghĩ, không giao tiếp với các triết gia khác.
- Khi đói, họ sẽ cố gắng lấy 2 chiếc đũa (lần lượt mỗi chiếc) gần nhất để ăn cơm.
- Một triết gia không thể lấy đũa đang được dùng bởi triết gia khác.
- Khi có được 2 chiếc đũa, triết gia sẽ ăn và đặt đũa xuống sau khi ăn xong; sau đó suy nghĩ tiếp.



- Dữ liệu chia sẻ: semaphore `chopstick[5]`, khởi tạo bằng 1.

Cấu Trúc Của Tiến Trình Triết Gia i

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

Giải thuật trên còn gặp phải vấn đề gì?

Ngăn Khóa Chết (Deadlock)

- Giải thuật đảm bảo không có trường hợp hai láng giềng ăn cùng lúc (sử dụng cùng đũa).
- Tuy nhiên, có thể xảy ra tình trạng khóa chết – 5 triết gia cùng đói và mỗi người lấy được 1 chiếc đũa.
- Giải pháp:
  - Cho phép nhiều nhất 4 triết gia ngồi trên bàn 
  - Chỉ cho phép một triết gia lấy đũa nếu cả 2 đũa đều sẵn sàng.
  - Dùng giải pháp bất đối xứng: triết gia lẻ luôn lấy đũa trái trước, còn triết gia chẵn thì luôn lấy đũa phải trước.

Monitors

- Là một cấu trúc dữ liệu trừu tượng, được cung cấp bởi các ngôn ngữ lập trình cấp cao, cho phép thực hiện việc đồng bộ hóa một cách dễ dàng, hiệu quả.
- Một cấu trúc monitor bao gồm:
  - Tập hợp các thao tác (hàm) cho phép truy xuất đến các dữ liệu bên trong monitor.
  - Các biến dữ liệu (biến) cục bộ.
  - Một đoạn mã khởi tạo (initialization code).

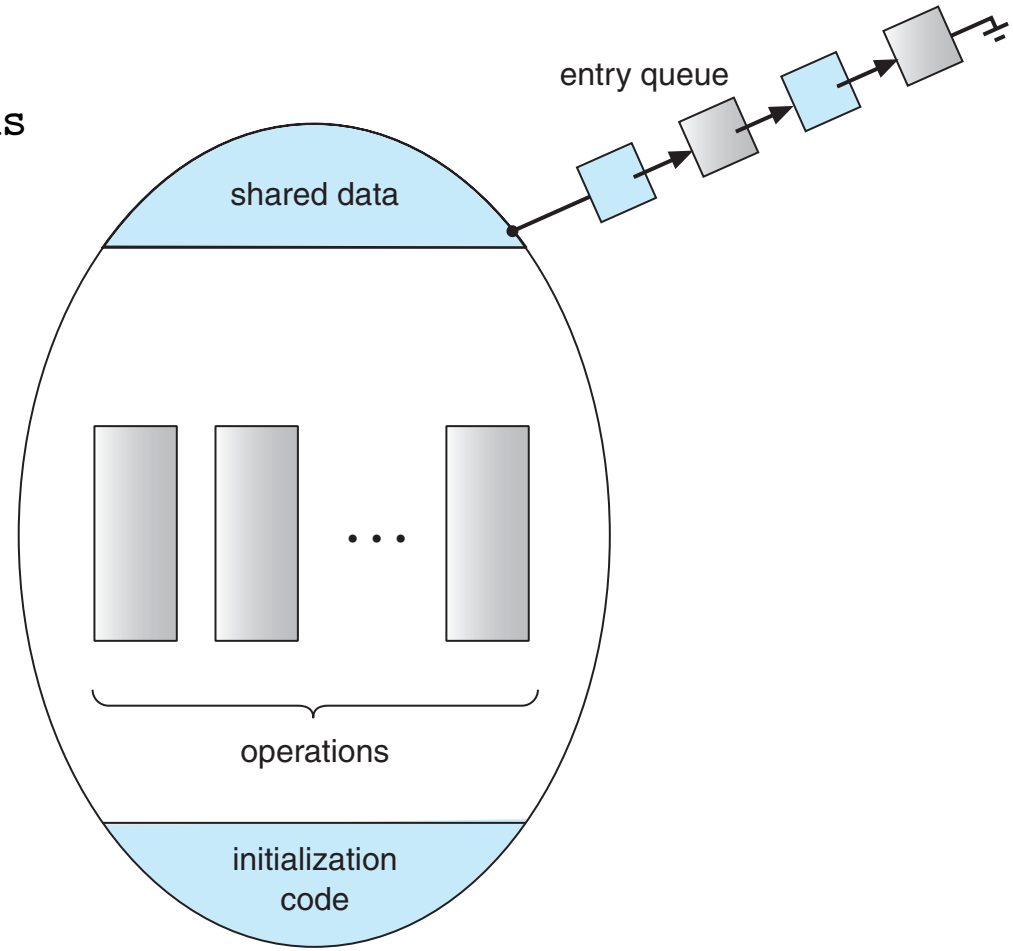
Cấu Trúc Monitor

```
monitor monitor_name
{
  shared variable declarations

  procedure P1(...) {
    ...
  }

  .
  .
  .
  procedure Pn(...) {
    ...
  }

  initialization_code (...) {
    ...
  }
} //monitor
```



Đặc Tính Của Monitor

- Các biến cục bộ chỉ có thể được truy xuất bởi các thủ tục của monitor.
- Tiến trình vào monitor bằng cách gọi một trong các thủ tục đó.
- Chỉ có thể có tối đa 1 tiến trình có thể vào monitor tại 1 thời điểm
  $\Rightarrow$ điều kiện loại trừ hỗ tương được đảm bảo.

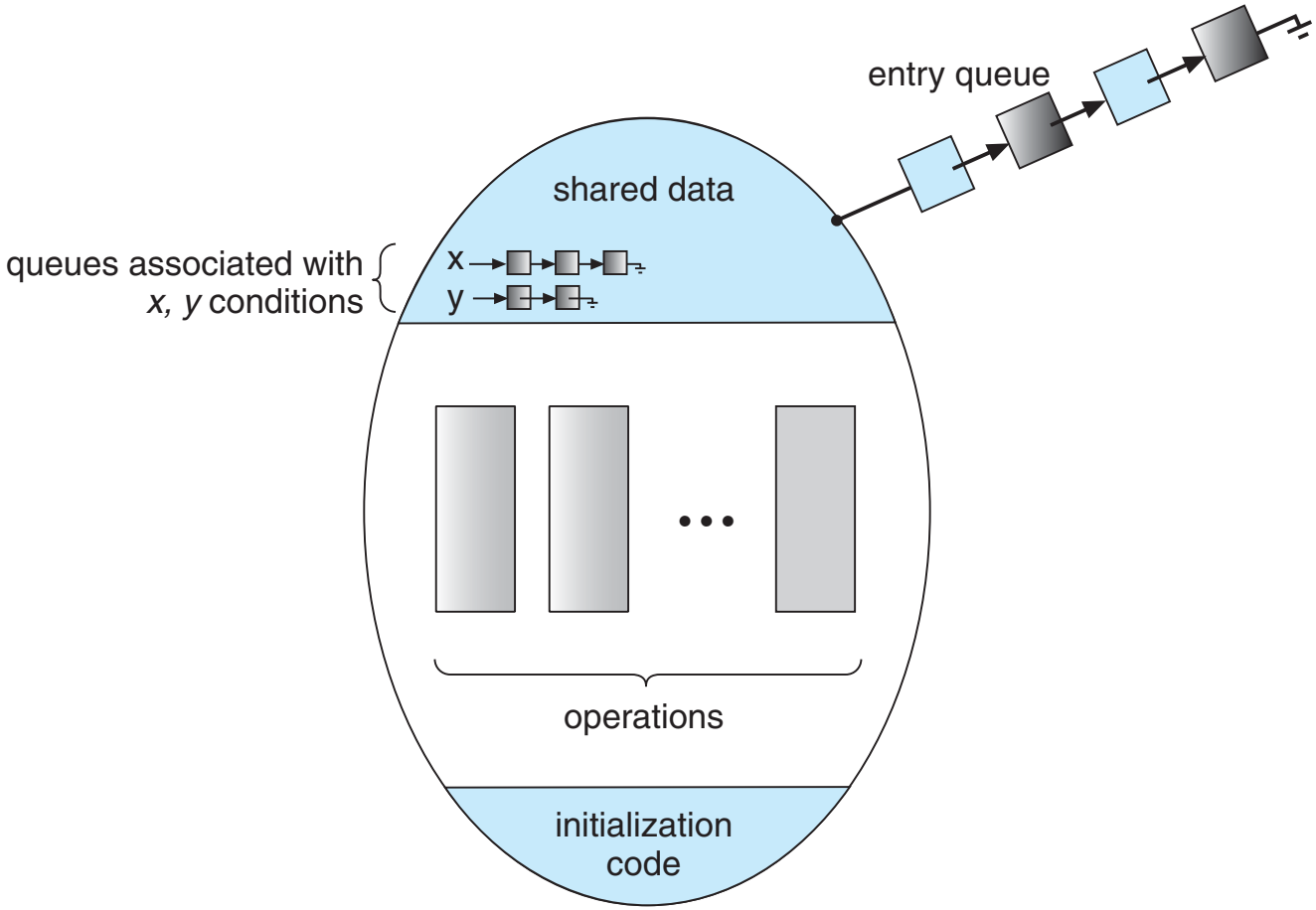
Biến Điều Kiện (Condition Variable)

- Nhằm cho phép một tiến trình đợi trong monitor, ta dùng các biến điều kiện:

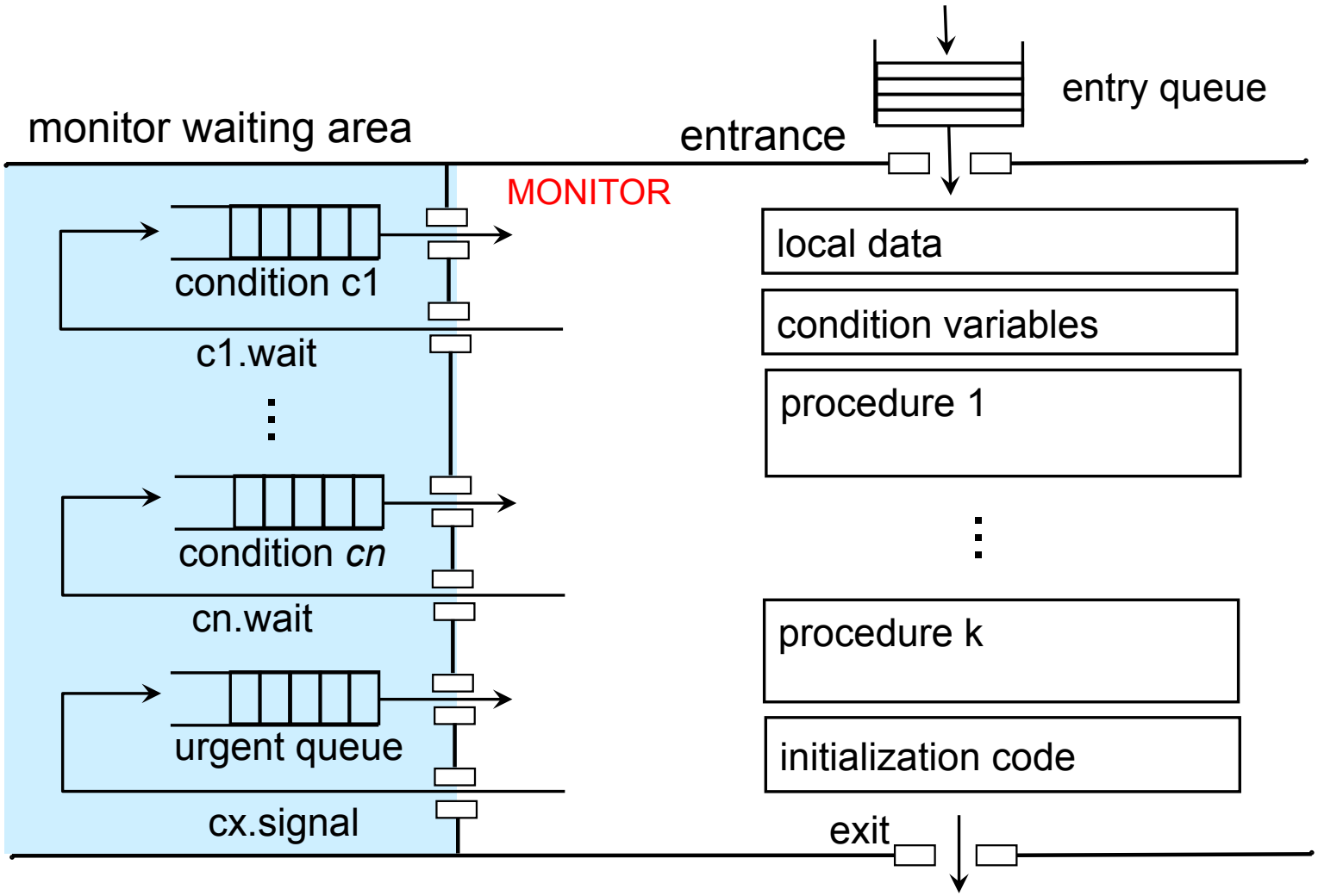
```
condition x, y;
```

- Biến điều kiện là cục bộ (chỉ được truy xuất bên trong monitor).
- Hai thao tác trên biến điều kiện:
  - `x.wait()`: tiến trình gọi chỉ thị này sẻ bị ngưng cho đến khi `x.signal()` được gọi.
  - `x.signal()`: "đánh thức" một trong các tiến trình đang ngưng do gọi `x.wait()` (nếu không có tiến trình đang ngưng thì không làm gì cả).

Biến Điều Kiện (Condition Variable)



Biến Điều Kiện (Condition Variable)



Các Tùy Biến Khi Gọi Hàm `signal()`

- Nếu tiến trình P gọi `x.signal()` trong khi tiến trình Q đang đợi trên biến này (Q gọi `x.wait()` trước đó), để tránh hai tiến trình thực thi đồng thời trong monitor:
  - **Signal and wait**: P chờ cho đến khi Q rời khỏi monitor.
  - **Signal and continue**: Q chờ cho đến khi P rời khỏi monitor.
- Nhiều ngôn ngữ lập trình cài đặt cơ chế đồng bộ hóa dựa trên ý tưởng monitor (Java, C#)

Bài Toán Các Triết Gia Ăn Tối Với Monitor

```
monitor DiningPhilosopiers {
  enum {thinking, hungry, eating} state[5];
  condition self[5]; //wait on chopstick i

  void pickup(int i); //pick up the chopstick i
  void putdown(int i); //put down the chopstick i
  void test(int i); //test the availability of the chopstick i

  void init() {
    for (int i = 0; i < 5; i++)
      state[i] = thinking;
  }
}
```

Bài Toán Các Triết Gia Ăn Tối Với 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();
  }
}
```

Bài Toán Các Triết Gia Ăn Tối Với 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 right+left neighbors
  test((i+4) % 5);
  test((i+1) % 5);
}
```

- **Tiến trình triết gia *i*:**

```
DiningPhilosophiers.pickup(i);
  ...
  EAT
  ...
DiningPhilosophiers.putdown(i);
```

- **Tính chất**:
  - Loại trừ hỗ tương: ✓
  - Tránh deadlock: ✓
  - Tránh chết đói: ✗


MARATHON
FINISH LINE