TƯƠNG TRANH VÀ ĐỒNG BỘ

ThS. Nguyễn Thị Hải Bình

Khoa CNTT, ĐH Giao thông vận tải

Email: calmseahn@gmail.com

Website: calmseahn.weebly.com

VÍ DỤ VỀ TƯƠNG TRANH

1. hiển thị giao diện & chờ người dùng ra lệnh
2. Người dùng ra lệnh nạp vào tài khoản A số tiền 700USD → xử lý :
   - 21a Đọc tài khoản A vào bộ nhớ,
   - 22a Tăng giá trị tài khoản trong bộ nhớ lên 700USD.
   - 23a Ghi lại giá trị mới.
3. Quay về bước 1

Tài khoản A

Vùng CS

1. hiển thị giao diện & chờ người dùng ra lệnh
2. Người dùng ra lệnh rút tiền từ tài khoản A 500USD → xử lý :
   - 21b Đọc tài khoản A vào bộ nhớ,
   - 22b Giảm giá trị tài khoản trong bộ nhớ đi 500USD.
   - 23b Ghi lại giá trị mới.
3. Quay về bước 1

TƯƠNG TRANH VÀ ĐỒNG BỘ

- Race condition
  - Thuật ngữ
    - Tranh đoạt điều khiển
    - Tình huống tương tranh
  - Xảy ra khi
    - Nhiều tiến trình cùng thao tác trên dữ liệu chung và kết quả các thao tác đó phụ thuộc vào thứ tự thực hiện của các tiến trình
- Process synchronization
  - Thuật ngữ: đồng bộ hoá các tiến trình
  - Để tránh các tình huống tương tranh, các tiến trình cần được đồng bộ theo một phương thức nào đó

BÀI TOÁN SẢN XUẤT – TIÊU THỤ

- Thuật ngữ
  - The producer – consumer problem
- Yêu cầu của bài toán
  - Tiến trình sản xuất (producer process) tạo ra thông tin
  - Còn tiến trình tiêu thụ (consumer process) sử dụng thông tin được tạo ra
  - Bộ đệm:
    - Chứa thông tin tạo ra bởi tiến trình sản xuất
    - Tiến trình tiêu thụ lấy thông tin từ bộ đệm để sử dụng
    - Bộ đệm cho phép 2 tiến trình thực thi đồng thời
  - Vấn đề
    - Tiến trình tiêu thụ không sử dụng thông tin chưa được tạo ra
    - Nếu bộ đệm rỗng thì tiến trình tiêu thụ phải chờ
    - Nếu bộ đệm đầy thì tiến trình sản xuất phải chờ

KHAI BÁO BIẾN

```
#define BUFFER_SIZE 10
typedef struct {
    DATA            data;
} item;
item   buffer[BUFFER_SIZE];
int    in = 0;              // Location of next input to buffer
int    out = 0;             // Location of next removal from buffer
int    counter = 0;         // Number of buffers currently full
```

TIẾN TRÌNH SẢN XUẤT

```
item newItem;

while( true ) {
   /* Produce an item and store it in newItem */
   newItem = makeNewItem( . . . );

   /* Wait for space to become available */
   while( counter == BUFFER_SIZE)
         ; /* Do nothing */

   /* And then store the item and repeat the loop. */
   buffer[in] = newItem;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

TIẾN TRÌNH TIÊU THỤ

```
item usedItem;

while( true ) {
   /* Wait for an item to become available */
   while( counter == 0)
         ; /* Do nothing */

   /* Get the next available item */
   usedItem = buffer[out];
   out = (out+1) % BUFFER_SIZE;
   counter--;

   /* Consume the item in usedItem (do something
   with it) */
}
```

TƯƠNG TRANH?

- Lệnh “counter++” và “counter--” có thể được cài đặt trên ngôn ngữ máy (typical machine language) như sau

counter++	**counter--**
$register_1$ = counter; $register_1$= $register_1$ + 1; counter = $register_1$;	$register_2$ = counter; $register_2$= $register_2$ - 1; counter = $register_2$;

TƯƠNG TRANH?

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Giải pháp phần mềm – Độc quyền truy xuất

Giải pháp phần cứng – Đồng bộ hoá

XỬ LÝ TƯƠNG TRANH

Giải pháp đồng bộ cơ bản

Giải pháp phần mềm – Độc quyền truy xuất

XỬ LÝ TƯƠNG TRANH

MIỀN GĂNG (CRITICAL SECTION)

- Còn được gọi là đoạn mã găng hay đoạn tới hạn
- Khái niệm miền găng
  - Xét hệ thống gồm n tiến trình $\{P_0, P_1, \ldots, P_{n-1}\}$
  - Mỗi tiến trình có một đoạn mã gọi là ***miền găng*** chứa các lệnh có thể thay đổi các biến dùng chung
- Vấn đề
  - Đảm bảo tại một thời điểm chỉ có một tiến trình được phép thi hành đoạn mã trong miền găng (*gọi là bước vào miền găng*)
  - Để các tiến trình có thể hợp tác với nhau, mỗi tiến trình cần phải *xin phép* trước khi bước vào miền găng và *thông báo* thoát khỏi miền găng

MIỀN GĂNG (CRITICAL SECTION)

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

Figure 5.1 General structure of a typical process P_i.

GIẢI PHÁP CHO MIỀN GĂNG

- Cần thoả mãn các yêu cầu sau
  - Độc quyền truy xuất (hay loại trừ lẫn nhau - Mutual Exclusion)
    - Nếu tiến trình P_i đang ở trong miền găng, thì không tiến trình nào được bước vào miền găng
  - Tiến triển (Progress)
    - Nếu không có tiến trình nào ở trong miền găng và có một số tiến trình muốn vào miền găng thì một tiến trình nào đó phải được vào miền găng
  - Chờ có giới hạn (Bounded waiting)
    - Thời gian từ khi tiến trình yêu cầu cho đến khi bước vào miền găng phải bị chặn bởi giới hạn nào đó

GIẢI PHÁP THỨ NHẤT CHO HAI TIẾN TRÌNH

- Giả sử có 2 tiến trình P_0 và P_1 muốn phối hợp với nhau vào miền găng
- Biến chung
  - int turn;
  - Khởi tạo: turn = 0 hoặc 1
  - turn = i → P_i được vào miền găng

GIẢI PHÁP THỨ NHẤT CHO HAI TIẾN TRÌNH

- Tiến trình P_i

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

- Vấn đề
  - Không thoả mãn yêu cầu tiến triển (progress)

GIẢI PHÁP THỨ HAI CHO HAI TIẾN TRÌNH

- Biến chung
  - boolean flag[2];
  - Khởi tạo: flag[0] = flag[1] = false
  - flag[i] = true → P_i sẵn sàng vào miền găng
- Tiến trình P_i

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

PHƯƠNG PHÁP KIỂM TRA VÀ XÁC LẬP

- Thuật toán Peterson
- Thuật toán Dekker
- Thuật toán tiệm bánh mỳ

THUẬT TOÁN PETERSON

- Thuật ngữ
  - Peterson’s solution
  - Peterson’s algorithm
- Biến chung
  - int turn;
    - Khởi tạo: turn = 0 hoặc 1
  - boolean flag[2];
    - Khởi tạo: flag[0] = flag[1] = false

THUẬT TOÁN PETERSON

- Tiến trình P_i

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

THUẬT TOÁN DEKKER

- Biến chung: Giống thuật toán Peterson
- Tiến trình P_i

```
do{
    flag[i] = true;
    while (flag[j]){
        if (turn=j){
            flag[i] = false;
            while (turn=j);
        }
        flag[i] = true;
    }
        critical section
    turn = j;
    flag[i] = false;
        reminder section
}while(true);
```

THUẬT TOÁN TIỆM BÁNH MỲ

- Thuật ngữ
  - Bakery algorithm
- Trước khi vào miền găng, mỗi tiến trình được cấp một số thứ tự
- Tiến trình có số thứ tự nhỏ nhất sẽ được vào miền găng trước
- Nếu P_i và P_j có cùng số thứ tự, nếu I < j thì P_i được vào miền găng trước

```
// declaration and initial values of global variables
Entering: array [NUM_THREADS] of bool = {false};
Number: array [NUM_THREADS] of integer = {0};

lock(integer i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
    Entering[i] = false;
    for (j = 0; j < NUM_THREADS; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) { /* nothing */ }
    }
}

unlock(integer i) {
    Number[i] = 0;
}

Thread(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
```

Giải pháp phần mềm – Độc quyền truy xuất

Giải pháp phần cứng – Đồng bộ hoá

XỬ LÝ TƯƠNG TRANH

Giải pháp đồng bộ cơ bản

Giải pháp phần cứng – Đồng bộ hoá

XỬ LÝ TƯƠNG TRANH

CHE NGẮT

- Không cho ngắt xảy ra khi chỉnh sửa biến chung
- → Chuỗi chỉ thị thao tác trên biến chung không bị gián đoạn bởi tiến trình khác
- Che ngắt trên hệ thống nhiều CPU tốn thời gian → hiệu suất hệ thống bị suy giảm

```
do {
    Prevent interrupts
        Critical section
    Allow interrupts
        Remainder section
} while(true);
```

CÁC CHỈ THỊ ĐẶC BIỆT

- Một số hệ thống cung cấp các chỉ thị phần cứng đặc biệt cho phép kiểm tra và chỉnh sửa nội dung của một từ hoặc tráo đổi nội dung hai từ trong bộ nhớ *một cách đơn nhất* (atomically)
- Các chỉ thị này là các đơn vị không thể bị ngắt (uninterruptible unit)

CHỈ THỊ TEST_AND_SET

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

   return rv;
}
```

Figure 5.3 The definition of the `test_and_set()` instruction.

CHỈ THỊ TEST_AND_SET

- Biến chung
  - Boolean lock = false;

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 5.4 Mutual-exclusion implementation with `test_and_set()`.

CHỈ THỊ COMPARE_AND_SWAP

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

CHỈ THỊ COMPARE_AND_SWAP

- Biến chung:
  - int lock = 0;

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

THUẬT TOÁN CHO MIỀN GĂNG

```
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);
```

Figure 5.7 Bounded-waiting mutual exclusion with `test_and_set()`.

Giải pháp phần mềm – Độc quyền truy xuất

Giải pháp phần cứng – Đồng bộ hoá

XỬ LÝ TƯƠNG TRANH

Giải pháp đồng bộ cơ bản

XỬ LÝ TƯƠNG TRANH

Giải pháp đồng bộ cơ bản

KHOÁ TRONG (MUTEX LOCK)

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}

release() {
   available = true;
}
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

KHOÁ TRONG (MUTEX LOCK)

- acquire() và release() phải thực thi một cách đơn nhất (atomically)
- Mutex lock thường được cài đặt bằng các cơ chế đồng bộ hoá (giải pháp phần cứng)
- Tình trạng chờ bận (busy waiting)
  - Khi có một tiến trình trong miền găng, tiến trình khác muốn vào miền găng phải thực hiện vòng lặp để gọi hàm acquire()
  - → Lãng phí thời gian CPU
- Dạng khoá này còn gọi là khoá xoay (spinlock) vì các tiến trình "spin" khi chờ khoá
- Ưu điểm của spinlock trong hệ thống đa xử lý là không cần chuyển ngữ cảnh khi một tiến trình đợi khoá

SEMAPHORE (PHƯƠNG PHÁP ĐÈN HIỆU)

- Semaphore S là một biến nguyên
- Ngoài toán tử khởi tạo, S chỉ được truy cập thông qua hai *toán tử nguyên tố* (standard atomic operation) wait() và signal()
  - wait() còn được gọi là P (proberen – kiểm tra)
  - signal() còn được gọi là V (verhogen – tăng)

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

SỬ DỤNG SEMAPHORE

- Binary semaphore (Semaphore nhị phân)
  - Có thể sử dụng để giải quyết vấn đề miền găng
  - Biến chung:
    - semaphore mutex;
    - Khởi tạo mutex = 1
  - Tiến trình P_i:

```
do{
    waiting(mutex);
      critical section
    signal(mutex);
      reminder section
}while(true);
```

SỬ DỤNG SEMAPHORE

- Counting semaphores
  - Nhận giá trị là một số nguyên bất kỳ
  - Thường dùng để đếm số lượng tài nguyên rảnh rỗi của hệ thống
  - Biến chung
    - semaphore S;
    - Khởi tạo: S = số lượng tài nguyên (resources) của hệ thống
  - Tiến trình P_i

```
do{
    waiting(S);
        using resource
    signal(S);
        reminder section
}while(true);
```

SỬ DỤNG SEMAPHORE

- Semaphore có thể dùng để giải quyết một số vấn đề đồng bộ hoá giữa các tiến trình
- Ví dụ
  - Tiến trình P1 có lệnh S1
  - Tiến trình P2 có lệnh S2
  - S1 phải được hoàn thành trước khi S2 thực thi
  - Biến chung: semaphore synch;
  - Khởi tạo synch = 0;

```
/* Tiến trình P1 */        /* Tiến trình P2 */

S1;                        wait(synch);
signal(synch);             S2;
```

CÀI ĐẶT SEMAPHORE

- Ý tưởng
  - Khi một tiến trình phải chờ vì semaphore âm, thay vì thực hiện lặp (vào tình trạng chờ bận – busy waiting), tiến trình phong toả (block) chính nó (*tiến trình chuyển từ trạng thái tích cực sang trạng thái không tích cực*)
  - Quá trình phong toả diễn ra như sau
    - Tiến trình được đặt vào hàng chờ semaphore
    - Tiến trình chuyển sang trạng thái chờ (waiting)
  - Khi semaphore sẵn sàng
    - Một trong số các tiến trình bị phong toả sẽ được đánh thức
    - Tiến trình đó sẽ được đưa vào hàng đợi ready hoặc được chuyển sang trạng thái running (tuỳ thuộc vào thuật toán điều phối CPU)

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

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}

signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

CÀI ĐẶT SEMAPHORE

- Toán tử wait() và signal() phải được thực thi một cách toàn vẹn và đơn nhất (atomically)
- Trên môi trường có một CPU có thể sử dụng giải pháp che ngắt
- Trên môi trường nhiều CPU, có thể sử dụng giải pháp khoá (compare_and_swap hoặc spinlocks)

BẾ TẮC (DEADLOCKS)

- Xảy ra khi có nhiều tiến trình bị phong toả, và mỗi tiến trình đó chỉ có thể bị đánh thức bởi một tiến trình bị phong toả khác
- Tiến trình P0 và P1 cùng truy cập vào semaphore S và Q, giá trị hiện thời của S và Q là 1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

CHẾT ĐÓI (STARVATION)

- Thuật ngữ khác: phong toả vĩnh viễn (indefinite blocking)
- Là hiện tượng xảy ra khi tiến trình chờ vô định trong semaphore
- Có thể xuất hiện nếu đánh thức tiến trình trong hàng chờ của semaphore theo thứ tự LIFO (last-in, first-out)

NHỮNG VẤN ĐỀ ĐỒNG BỘ KINH ĐIỂN

- Vấn đề bộ đệm giới hạn (the bounded-buffer proble)
- Vấn đề đọc – ghi (the readers and writers problem)
- Bữa ăn tối của triết gia (the dining-philosophers problem)

VẤN ĐỀ BỘ ĐỆM GIỚI HẠN

- Bài toán sản xuất – tiêu thụ
- Biến chung
  - n – kích thước của bộ đệm
  - mutex – kiểm soát việc truy cập vào bộ đệm
  - empty và full để đếm số ô trống và đầy trong bộ đệm

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

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

Figure 5.9 The structure of the producer process.

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

Figure 5.10 The structure of the consumer process.

VẤN ĐỀ ĐỌC GHI

- Vấn đề đọc ghi thứ nhất
  - Tiến trình đọc được ưu tiên
  - Nếu không có tiến trình ghi nào đang truy cập vào dữ liệu, thì tiến trình đọc sẽ được cấp quyền truy cập dữ liệu ngay khi yêu cầu
  - Tiến trình ghi có thể bị chết đói (starvation)
- Vấn đề đọc ghi thứ hai
  - Tiến trình ghi được ưu tiên
  - Khi tiến trình ghi muốn truy cập vào dữ liệu, nó được đưa vào đầu hàng đợi. Ngay khi dữ liệu sẵn sàng, tiến trình ghi được cấp quyền truy cập.
  - Tiến trình đợi có thể bị chế đói

VẤN ĐỀ ĐỌC GHI THỨ NHẤT

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

- Biến chung
  - read_count
    - Sử dụng bởi tiến trình đọc
    - Đếm số lượng tiến trình đọc đang truy cập vào dữ liệu
  - mutex
    - Sử dụng bởi tiến trình đọc
    - Kiểm soát truy cập vào readcount
  - rw_mutex
    - Sử dụng để block và release tiến trình ghi
    - Tiến trình đọc truy cập vào dữ liệu đầu tiên sẽ thiết lập giá trị để block
    - Tiến trình đọc cuối cùng truy cập vào dữ liệu sẽ thiết lập giá trị để release

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

Figure 5.11 The structure of a writer process.

```
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);
```

Figure 5.12 The structure of a reader process.

BỮA ĂN TỐI CỦA TRIẾT GIA

- Mỗi triết gia chỉ suy nghĩ (thinking) hoặc ăn (eat)
- Các triết gia không trao đổi với nhau

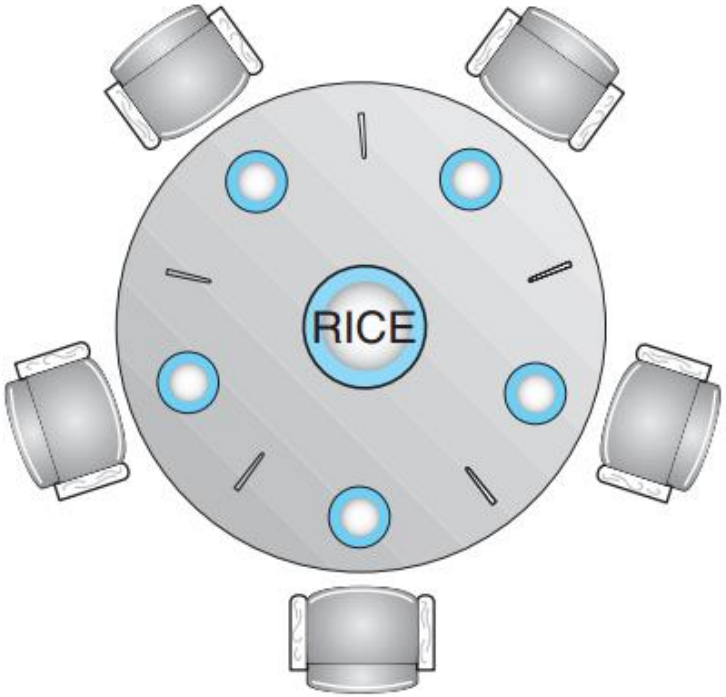

Figure 5.13 The situation of the dining philosophers.

GIẢI PHÁP DÙNG SEMAPHORE

```
semaphore chopstick[5];

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

Figure 5.14 The structure of philosopher *i*.

DEADLOCK?

GIẢI PHÁP CHO VẤN ĐỀ BẾ TẮC

- Chỉ cho phép tối đa 4 triết gia ăn cùng một lúc
- Chỉ cho phép triết gia nhặt đũa khi cả 2 chiếc đang không bị sử dụng
- Giải pháp phi đối xứng (asymmetric solution)
  - Đánh số các triết gia
  - Triết gia có số thứ tự lẻ phải lấy chiếc đũa phía bên trái trước
  - Triết gia có số thứ tự chẵn phải lấy chiếc đũa phía bên phải trước
- Vấn đề “chết đói” (starvation) chưa được giải quyết

HẠN CHẾ CỦA SEMAPHORE

- Sử dụng semaphore không đúng cách có thể dẫn đến bế tắc hoặc lỗi do trình tự thực hiện của các tiến trình
- Sử dụng không đúng cách gây ra bởi lỗi lập trình hoặc do người lập trình không cộng tác

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

MONITOR (PHƯƠNG PHÁP DÙNG TRÌNH THƯ KÝ)

- Cấu trúc trong ngôn ngữ lập trình bậc cao dùng để phục vụ các thao tác đồng bộ hoá
- Các thành phần của monitor
  - Initialization code: thực thi một lần duy nhất khi tạo monitor
  - Private data (bao gồm private procedures): chỉ có thể sử dụng *bên trong* monitor
  - Monitor procedures: hàm/thủ tục có thể được gọi từ bên ngoài monitor
  - Monitor entry queue: hàng đợi các tiến trình đang chờ thực thi monitor procedures

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 5.15 Syntax of a monitor.

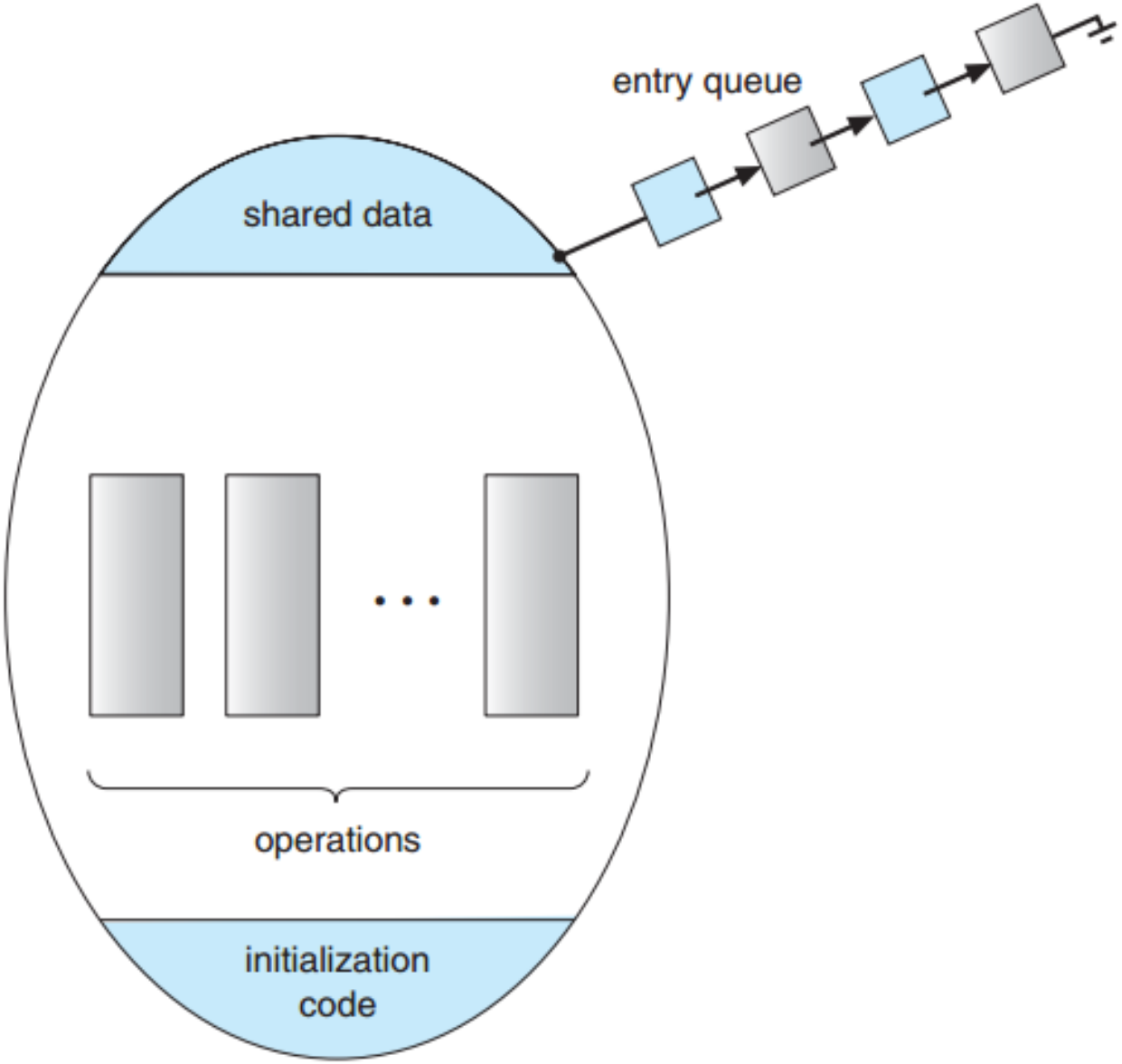

Figure 5.16 Schematic view of a monitor.

MONITOR

- Monitor đảm bảo tại một thời điểm chỉ có duy nhất một tiến trình được hoạt động bên trong monitor
- Các hàm/thủ tục trong monitor chỉ có thể truy cập vào các biến cục bộ và các tham số hình thức

CONDITION

- Khai báo

```
condition x, y;
```

- Sử dụng: 2 toán tử wait và signal
  - x.wait(): chuyển tiến trình sang trạng thái chờ
  - x.signal(): tiến trình gọi x.signal() sẽ đánh thức tiến trình đã gọi x.wait()
    - Đánh thức duy nhất một tiến trình đang chờ
    - Nếu không có tiến trình chờ, x.signal() không có tác dụng
    - *Chú ý: signal() trong semaphore luôn làm thay đổi giá trị semaphore*

SIGNAL WAIT/CONTINUE

- Giả sử có 2 tiến trình P và Q
  - Q gọi x.wait()
  - P gọi x.signal()
- Hai khả năng
  - Signal-and-wait: P chờ đến khi Q rời monitor hoặc chờ điều kiện khác
  - Signal-and-continue: Q chờ đến khi P rời monitor hoặc chờ điều kiện khác

TỰ ĐỌC

- Giải quyết bài toán bữa tối của triết gia bằng monitor
- Cài đặt monitor bằng semaphore