1. *Sự khác nhau giữa một thread vả một process?*

**SOLUTION**

Processes và threads có liên quan tới nhau nhưng về cơ bản là khác nhau.

Một process có thể được nghĩ như là một ví dụ của chương trình trong thực thi. Một process là một thực thể độc lập mà tài nguyên hệ thống (vd: thời gian CPU và bộ nhớ) được cấp phát. Mỗi process được thực thi trong một không gian địa chỉ riêng biệt, và một process không thể truy cập cac biến và cấu trúc dữ liệu của process khác. Nếu một process muốn truy cập tới tài nguyên của process khác, inter-process communications (IPC : hành động trao đổi dữ liệu giữa các tiến trình riêng biệt, sử dụng giao thức kết nối) có thể được sử dụng. Chúng bao gồm pipes, files, sockets, và các forms khác.

Một thread tồn tại trong một process và chia sẻ tài nguyên của process (bao gồm không gian lưu trữ của nó). Nhiều threads trong cùng một quá trình sẽ chia sẻ cùng một không gian lưu trữ. Nó rất khác với processes, cái mà không thể truy cập địa chỉ vùng nhớ của process khác. Mỗi thread vẫn có các thanh ghi riêng và ngăn xếp riêng của nó, nhưng các threads khác có thể đọc và ghi bộ nhớ lưu trữ.

Một thread là một đường dẫn thực thi cụ thể của một process. Khi một thread sửa đổi một tài nguyên process, sự thay đổi sẽ hiện thị lập tức cho anh chị em threads.

1. *Bạn làm thế nào để đo thời gian dùng cho việc chuyển đổi một phạm vi?*

**SOLUTION**

Đây là một câu hỏi khó, nhưng bắt đầu với một giải pháp có thể giải quyết.

Chuyển đổi một phạm vi là thời gian chuyển đổi giữa hai processes (tức là, đưa một process đang đợi vào thực thi và gửi một process đang thực thi vào trạng thái đợi/chấm dứt). Điều này xảy ra trong đa nhiệm. Hệ điều hành phải mang thông tin của processes đang đợi vào bộ nhớ và lưu thông tin trạng thái của process đang chạy hiện tại.

Để giải quyết vấn đề này, chúng tôi muốn ghi lại các mốc thời gian của lệnh cuối và đầu tiên của processes chuyển đổi. Thời gian chuyển phạm vi là sự khác biệt về mốc thời gian giữa hai processes.

Hãy lấy một ví dụ đơn giản: Giả sử chỉ có hai processes, P1 và P2.

P1 đang thực thi và P2 đang chờ thực thi. Tại một thời điểm nào đó, hệ điều hành phải hoán đổi P1 và P2 – giả sử nó xảy ra ở lệnh thứ N của P1. Nếu tx,k biểu thị mốc thời gian tính bằng microseconds của lệnh thứ k của process x, thì chuyển đổi phạm vi sẽ mất t2,1 – t1,n microseconds.

Phần khó khăn là: Làm thế nào chúng ta biết khi nào sự hoán đổi xảy ra? Chúng ta không thể, tất nhiên, ghi lại mốc thời gian của mỗi lệnh trong process.

Một vấn đề khác là trao đổi được điều chỉnh bởi thuật toán lập lịch của hệ điều hành và có thể có nhiều threads cấp hạt nhân cũng đang thực hiện chuyển đổi phạm vi. Processes khác cũng có thể cạnh tranh cho CPU hoặc các ngắt xử lý hạt nhân. Người dùng không có bất kỳ kiểm soát nào đối với các chuyển đổi phạm vi không liên quan này. Ví dụ, nếu tại thời điểm t1,n hạt nhân quyết định xử lý ngắt, sau đó thời gian chuyển đổi phạm vi sẽ bị phóng đại.

Để khắc phục những trở ngại này, chúng ta phải xây dụng một môi trường như vậy sau khi P1 thực thi, trình lên lịch tác vụ lập tức chọn P2 để chạy. Điều này có thể thực hiện bằng cách xây dựng một kênh dữ liệu, như là một đường ống, giữa P1 và P2 và có hai processes đang chơi trò chơi ping-pong với mã thông báo dữ liệu.

Tức là, hãy cho phép P1 là người gửi ban đầu và P2 là người nhận. Ban đầu, P2 bị chặn (ngủ) vì nó chờ mã thông báo dữ liệu. Khi P1 thực hiện, nó cung cấp mã thông báo qua kênh dữ liệu cho P2 và ngay lập tức cố gắng đọc mã thông báo phản hồi. Tuy nhiên, vì P2 chưa có cơ hội để chạy, nên không có mã thông báo nào có sẵn cho P1 và process này bị chặn. Điều này từ bỏ CPU.

Một kết quả chuyển đổi phạm vi và trình lên lịch tác vụ phải chọn process khác để chạy. Vì P2 hiện đang ở trạng thái ready-to-run, nó là một ứng cử viên mong muốn được chọn bởi trình lên lịch tác vụ để thực hiện. Khi P2 chạy, các vai trò của P1 và P2 được chuyển đổi. P2 hiện đang đóng vai trò là người gửi và P1 là người bị chặn. Trò chơi kết thúc khi P2 trả lại mã thông báo cho P1.

Tóm lại, một lần lặp của trò chơi được thực hiện theo các bước sau:

1. Các khối P2 đang chờ dữ liệu từ P1.
2. P1 đánh mốc thời gian bắt đầu.
3. P1 gửi mã thông báo cho P2.
4. P1 cố gắng đọc mã thông báo phản hồi từ P2. Điều này gây ra chuyển đổi phạm vi.
5. P2 được lên lịch và nhận mã thông báo.
6. P2 gửi mã thông báo phản hồi tới P1.
7. P2 cố gắng đọc mã thông báo phản hồi từ P1. Điều này tạo ra chuyển đổi phạm vi.
8. P1 được lên lịch và nhận mã thông báo.
9. P1 đánh mốc thời gian kết thúc.

Điều quan trọng là việc cung cấp một mã thông báo dữ liệu gây ra một chuyển đổi phạm vi. Hãy để Td và Tr là thời gian cần để phân phối và nhận mã thông báo, tương ứng, và để Tc là lượng thời gian dành cho một chuyển đổi phạm vi. Tại bước 2, P2 ghi lại mốc thời gian của việc cung cấp mã thông báo, và tại bước 9, nó ghi lại mốc thời gian của phản hồi. Số lượng của thời gian trôi qua, T, giữa các sự kiện này được biểu thị bởi:

T = 2 \* (Td + Tc + Tr)

Công thức này phát sinh bởi vì các sự kiện sau: P1 gửi mã thông báo (3), CPU chuyển đổi phạm vi (4), P2 nhận nó (5). P2 sau đó gửi thông báo phản hồi (6), CPU chuyển đổi phạm vi (7), và cuối cùng P1 nhận nó (8).

P1 sẽ có thể dễ dàng tính toán T, vì đây chỉ là thời gian giữa sự kiện 3 và 8. Vì thế, để giải quyết cho Tc, chúng ta phải xác định giá trị của Td + Tr.

Làm thế nào chúng ta có thể làm điều này? Chúng ta có thể làm điều này bởi đo độ dài thời gian nó cần P1 để gửi và nhận một mã thông báo cho chính nó. Điều này sẽ không tạo ra một chuyển đổi phạm vi vì P1 đang chạy trên CPU tại thời gian nó gửi mã thông báo và sẽ không chặn để nhận nó.

Trò chơi được chơi một vài lần lặp để tính trung bình bất kì sự thay đổi nào trong thời gian trôi qua giữa bước 2 và 9 có thể là kết quả từ các hạt nhân ngắt bất ngờ và bổ sung các chuỗi hạt nhân threads đang cạnh tranh cho CPU. Chúng ta chọn thời gian chuyển đổi phạm vi được quan sát nhỏ nhất làm câu trả lời cuối cùng của chúng ta.

Tuy nhiên, tất cả chúng ta cuối cùng có thể nói rằng đây là một xấp xỉ phụ thuộc vào hệ thống cơ bản. Ví dụ, chúng ta giả định rằng P2 được chọn để chạy một dữ liệu mã thông báo khi có sẵn. Tuy nhiên, điều này phụ thuộc vào việc thực hiện trình trình lên lịch tác vụ và chúng ta không thể làm bất kì thỏa thuận nào.

Không sao đâu; đây là điều quan trọng trong một cuộc phỏng vấn để nhận ra khi giải pháp của bạn có thể không hoàn hảo.

1. *Trong vấn đề triết lý ăn uống nổi tiếng, một loạt các triết học đang ngồi quanh một chiếc bàn tròn với một chiếc đũa giữa mỗi người. Một nhà triết học cần cả hai chiếc đũa để ăn, và luôn nhấc chiếc đũa bên trái trước chiếc đũa bên phải. Một deadlock có thể có khả năng xảy ra nếu tất cả nhà triết học với lấy chiếc đũa bên trái cùng một lúc. Sử dụng threads và locks, thực hiện một mô phỏng của vấn đề triết lý ăn uống để ngăn ngừa deadlocks.*

**SOLUTION**

Đầu tiên, hãy thực hiện một mô phỏng đơn giản của vấn đề triết lý ăn uống mà trong đó chúng ta không quan tâm đến chính mình với deadlocks. Chúng ta có thể thực hiện giải pháp này bởi đang có Philosopher extend Thread, và Choplock gọi lock.lock() khi nó được chọn và lock.unlock() khi nó được đặt xuống.

public class Chopstick {

private Lock lock;

public Chopstick() {

lock = new ReentrantLock();

}

public void pickUp() {

void lock.lock();

}

public void putDown() {

lock.unlock();

}

}

public class Philosopher extends Thread {

private int bites = 10;

private Chopstick left;

private Chopstick right;

public Philosopher(Chopstick left , Chopstick right) {

this.left = left;

this.right = right;

}

public void eat() {

pickUp();

chew();

putDown();

}

public void pickUp() {

left.pickUp();

right.pickUp();

}

public void chew() { }

public void putDown() {

left.putDown();

right.putDown();

}

public void run() {

for (int i = 0; i < bites; i++) { eat(); }

}

}

Chạy đoạn mã trên có thể dẫn đến một deadlock nếu tất cả các nhà triết học có một chiếc đũa trái và đàng chờ một chiếc bên phải.

Để ngăn chặn deadlocks, chúng ta có thể thực hiện một chiến lược trong đó một nhà triết học sẽ đặt xuống chiếc đũa trái của anh ta nếu anh ta không thể có được một chiếc bên phải.

public class Chopstic k {

private Lock lock;

public Chopstick() {

lock = new ReentrantLock();

}

public void pickUp() {

void lock.lock();

}

public void putDown() {

lock.unlock();

}

public boolean pickUp() {

return lock.tryLock();

}

}

public class Philosopher extends Thread {

private int bites = 10;

private Chopstick left;

private Chopstick right;

public Philosopher(Chopstick left , Chopstick right) {

this.left = left;

this.right = right;

}

public void eat() {

if (pickUp()) {

chew();

putDown();

}

}

public void pickUp() {

left.pickUp();

right.pickUp();

}

public void chew() { }

public void putDown() {

left.putDown();

right.putDown();

}

public void run() {

for (int i = 0; i < bites; i++)

{

eat();

}

}

public boolean pickup() {

/\* attempt to pick up \*/

if (!left.pickUp()) { return false; }

if (!right.pickUp()) {

left.putDown();

return false;

}

return true;

}

}

Trong đoạn mã trên, chúng ta cần phải chắc chắn để thả chiếc đũa bên trái nếu chúng ta không thể lấy được một chiếc bên phải – và đừng gọi putDown() trên đũa nếu chúng ta chưa bao giờ có chúng ở nơi đầu tiên.

1. *Thiết kế một lớp chỉ cung cấp một lock nếu không có deadlocks khả thi.*

**SOLUTION**

Có một số cách phổ biến để ngăn chặn cho deadlocks. Một trong những cách phổ biến là yêu cầu một process khai báo trước những gì locks nó là cần thiết.

Sau đó, chúng ta có thể xác minh xem nếu một deadlock có được tạo ra bởi vấn đề locks, và chúng ta có thể thất bại nếu có. Với những ràng buộc trong tâm trí, hãy điều tra làm thế nào chúng ta có thể phát hiện deadlocks. Giả sử đây là thức tự của các locks được yêu cầu:

A = {1, 2, 3, 4}

B = {1, 3, 5}

C = {7, 5, 9, 2}

Điều này có thể tạo ra một deadlock bời vì chúng ta có thể có trường hợp sau đây:

A locks 2, waits on 3

B locks 3, waits on 5

C locks 5, waits on 2

Chúng ta có thể nghĩ về điều này như một đồ thị, trong đó 2 được kết nối với 3, 3 được kết nối với 5, và 5 được kết nối với 2. Một deadlock được biểu diễn bởi một chu trình. Một cạnh (w, v) tồn tại trong biểu đồ nếu một process tuyên bố rằng nó sẽ yêu cầu lock v ngay lập tức sau đó lock w. Đối với ví dụ trước, các cạnh sau sẽ tồn tại trong biểu đồ: (1, 2), (2, 3), (3, 4), (1, 3), (3, 5), (7, 5), (5, 9), (9, 2). “Chủ sở hữu” của cạnh không quan trọng.

Lớp này sẽ cần một phương thức khai báo, mà threads và processes sẽ sử dụng để khai báo thứ tự mà chúng sẽ yêu cầu tài nguyên. Phương thức khai báo sẽ lặp lại qua thứ tự khai báo, thêm mỗi cặp phần tử liền nhau (v, w) vào biểu đồ. Sau đó, nó sẽ kiểm tra xem nếu bất kì chu kì nào được tạo hay không. Nếu bất kì chu kì nào được tạo, nó sẽ quay lại, loại bỏ các cạnh khỏi biểu đồ, và sau đó thoát ra.

Chúng tôi chỉ có một thành phần cuối cùng để thảo luận: làm như thế nào để chúng ta phát hiện một chu kì? Chúng ta có thể phát hiện một chu kì bằng cách thực hiện một tìm kiếm sâu đầu tiên thông qua mỗi thành phần được kết nối (tức là, mỗi phần được kết nối của biểu đồ). Các thuật toán phức tạp tồn tại để chọn tất cả các thành phần được kết nối của biểu đồ, nhưng công việc của chúng ta trong vấn đề này không yêu cầu mức độ phức tạp này.

Chúng ta biết rằng nếu một chu kì được tạo, một trong những cạnh mới của chúng ta phải đổ lỗi. Vì vậy, miễn là tìm kiếm sâu đầu tiên của chúng ta chạm vào tất cả các cạnh tại một số điểm, thì chúng ta biết rằng chúng ta đã tìm kiếm đầy đủ một chu kì.

Mã giả cho sự phát hiện chu kì trường hợp đặc biệt này trông giống như sau:

boolean checkForCycle(locks[] locks ) {

touchedNode s = hash table(lock > boolean )

initialize touchedNodes to false for each lock in locks

for each (lock x in process.locks ) {

if (touchedNodes[x] = = false) {

if (hasCycle(x , touchedNodes) ) { return true; }

}

}

return false;

}

boolean hasCycle(node x, touchedNodes) {

touchedNodes[r] = true;

if (x.state == VISITING) {

return true;

} else if (x.state == FRESH) {

... (see full code below)

}

}

Trong đoạn mã trên, lưu ý rằng chúng ta có thể làm một số tìm kiếm sâu đầu tiên, nhưng touchedNodes chỉ được khởi tạo một lần. Chúng ta lặp lại cho đến khi tất cả các giá trị trong touchedNodes là sai.

Mã bên dưới cung cấp thêm chi tiết. Để đơn giản, chúng tôi giả định rằng tất cả locks và processes (chủ sở hữu) được sắp xếp theo thứ tự.

public class LockFactor y {

private static LockFactor y instance ;

private int numberOfLocks = 5; /\* default \*/

private LockNode[] locks;

/\* Maps from a process or owner to the order that the owner

\* claimed it would call the locks in \*/

private Hashtable<Integer,LinkedList<LockNode>> lockOrder;

private LockFactory(int count) { ... }

public static LockFactory getlnstance() { return instance; }

public static synchronized LockFactory initialize(int count ) {

if (instance == null) instance = new LockFactory(count) ;

return instance;

}

public boolean hasCycle(Hashtable<Integer,Boolean> touchedNodes, int[] resourcesInOrder) {

/\* check for a cycle \*/

for (int resource : resourcesInOrder) {

if (touchedNodes.get(resource) == false) {

LockNode n = locks[resource];

if (n.hasCycle(touchedNodes)) { return true; }

}

}

return false;

}

/\* To prevent deadlocks, force the processes to declare upfront

\* what order they will need the locks in. Verify that this

\* order does not create a deadlock (a cycle in a directed

\* graph)\*/

public boolean declare(int ownerId, int[] resourcesInOrder ) {

Hashtable<Integer,Boolean> touchedNodes = new Hashtable<Integer,Boolean>();

/\* add nodes to graph \*/

int index = 1;

touchedNodes.put(resourcesInOrder[0],false);

for(index = 1;index < resourcesInOrder.length; index++) {

LockNode prev = locks[resourcesInOrder[index - 1]];

LockNode curr = locks[resourcesInOrder[index]];

prev.joinTo(curr);

touchedNodes.put(resourcesInOrder[index],false);

}

/\* if we created a cycle, destroy this resource list and 53

\* return false \*/

if(hasCycle(touchedNodes,resourcesInOrder)) {

for (int j = 1;j < resourcesInOrder.length; j++) {

LockNode p = locks[resourcesInOrder[j - 1]];

LockNode c = locks[resourcesInOrder[j]];

p.remove(c);

}

return false;

}

/\* No cycles detected. Save the order that was declared, so

\* that we can verify that the process is really calling the

\* locks in the order it said it would. \*/

LinkedList<LockNode> list = new LinkedList<LockNode>();

for(int i = 0; i < resourcesInOrder.length; i++) {

LockNode resource = Iocks[resourcesInOrder[i]];

list.add(resource);

}

lockOrder.put(owner!d , list);

return true;

}

/\* Get the lock, verifying first that the process is really

\* calling the locks in the order it said it would. \*/

public Lock getLock(int ownerId, int resourceID) {

LinkedList<LockNode> list = lockOrder.get(ownerId);

if (list == null) return null;

LockNode head = list.getFirst();

if (head.getId() == resourceID) {

list.removeFirst();

return head.getLock();

}

return null;

}

}

public class LockNode {

public enum VisitState { FRESH, VISITING, VISITED };

private ArrayList<LockNode> children;

private int lockId;

private Lock lock;

private int maxLocks;

public LockNode(int id, int max) { ... }

/\* Join "this" to "node", checking to make sure that it doesn't

\* create a cycle \*/

public void joinTo(LockNode node ) { children.add(node); }

public void remove(LockNode node) { children.remove(node); }

/\* Check for a cycle by doing a depth-first-search. \*/

public boolean hasCycle(Hashtable<Integer,Boolean> touchedNodes) {

VisitState[] visited = new VisitState[maxLocks];

for (int i = 0; i < maxLocks; i++) { visited[i] = VisitState.FRESH; }

return hasCycle(visited, touchedNodes);

}

private boolean hasCycle(VisitState[] visited,Hashtable<Integer,Boolean> touchedNodes) {

if (touchedNodes.containsKey(lockId)) { touchedNodes.put(lockId,true); }

if (visited[lockId] == VisitState.VISITING) {

/\* We looped back to this node while still visiting it, so

\* we know there's a cycle. \*/

return true;

} else if (visited[lockId] == VisitState.FRESH) {

visited[lockId] = VisitState.VISITING;

for (LockNode n : children) {

if (n.hasCycle(visited, touchedNodes)) { return true; }

}

visited[lockId] = VisitState.VISITED;

}

return false;

}

public Lock getLock() {

if (lock == null) lock = new ReentrantLock();

return lock;

}

public int getId() { return lockId; }

}

Như mọi khi, khi bạn thấy mã này phức tạp và dài dòng, bạn sẽ không được kỳ vọng viết tất cả. Nhiều khả năng, bạn sẽ được yêu cầu phác thảo mã giả và có thể triển khai một trong các phương pháp này.

1. Giả sử chúng ta có đoạn mã sau:

public class Foo {

public Foo( ) { ... }

public void first() { ... }

public void second() { ... }

public void third() { ... }

}

*Ví dụ tương tự của Foo sẽ được chuyển đến ba luồng khác nhau. ThreadA sẽ gọi đầu tiên, threads sẽ gọi thứ hai, và threadC sẽ gọi thứ ba. Thiết kế một cơ chế để đảm bảo rằng đầu tiên được gọi trước thứ hai và thứ hai được gọi trước thứ ba.*

**SOLUTION**

Logic chung là để kiểm tra nếu firstQ đã hoàn thành trước khi thực hiện second(), và nếu second() đã hoàn thành trước khi gọi third(). Bởi vì chúng ta cần phải rất cẩn thận về an toàn thread, cờ luận lý đơn giản sẽ không thực hiện công việc.

Điều gì về việc sử dụng một lock để làm gì đó giống như mã dưới đây?

public class FooBad {

public int pauseTime = 1000;

public ReentrantLock lock1,lock2,lock3;

public FooBad() {

try {

lock1 = new ReentrantLock();

lock2 = new ReentrantLock();

lock3 = new ReentrantLock();

lock1.lock();

lock2.lock();

lock3.lock();

} catch(...) { ... }

}

public void first() {

try { lock1.unlock(); // mark finished with firstQ

} catch(...) { ... }

}

public void second( ) {

try {

lock1.lock() ; // wait until finished with first()

lock1.unlock();

lock2.unlock(); // mark finished with second()

} catch(...) { ... }

}

public void third() {

try {

lock2.lock(); // wait unti l finished with third()

lock2.unlock();

} catch(...) { ... }

}

}

Mã này sẽ không thật sự làm việc hiệu quả do khái niệm của quyền sở hữu lock. Một thread thật sự đang thực hiện lock (trong hàm xây dựng FooBad), nhưng threads khác nhau cố gắng mở khóa locks. Điều này không được phép, và mã của bạn sẽ tăng một ngoại lệ. Một lock trong Java được sỡ hữu bới cùng một thread khóa nó.

Thay vào đó, chúng ta có thể tái tạo hành vi này với semaphores. Logic là giống nhau.

public class Foo {

public Semaphore sem1,sem2,sem3;

public Foo() {

try {

sem1 = new Semaphore(1);

sem2 = new Semaphore(1);

sem3 = new Semaphore(1);

sem1.acquire();

sem2.acquire();

sem3.acquire();

} catch(...) { ... }

}

public void first() {

try {

...

sem1.release();

} catch(...) { ... }

}

public void second() {

try {

sem1.acquire();

sem1.release();

...

sem2.release();

} catch(...) { ... }

}

public void third() {

try {

sem2.acquire();

sem2.release();

...

} catch(...) { ... }

}

}

1. *Bạn được nhận được một lớp với phương thức A đồng bộ và phương thức B bình thường. Nếu bạn có hai threads trong một cá thể của chương trình. Họ có thể thực thi cả hai cùng một lúc không? Họ có thể thực thi A và B cùng một lúc?*

**SOLUTION**

Bằng cách áp dụng lệnh được đồng bộ hóa với một phương pháp, chúng tôi đảm bảo rằng hai threads không thể thực thi các phương thức trong cùng một cá thể tại cùng thời gian.

Vì vậy, câu trả lời cho phần đầu tiên thật sự phụ thuộc. Nếu hai threads có cùng một thực thể của đối tượng, thì không, chúng không thể thực hiện đồng thời phương thức A. Tuy nhiên, nếu chúng có các thực thể khác nhau của đối tượng, thì chúng có thể.

Về mặt khái niệm, bạn có thể thấy điều này bằng cách xem xét locks. Một phương thức đồng bộ ấp dụng một “lock” trên tất cả các phương thức được đồng bộ hóa trong thực thể của đối tượng. Điều này ngăn chặn threads khác thực hiện các phương thức đồng bộ hóa trong thực thể đó.

Trong phần thứ hai, chúng ta được hỏi nếu thread1 có thể thực thi phương thức A đồng bộ trong khi thread2 đang thực thi phương thức B không đồng bộ. Vì B không đồng bộ, không có gì ngăn chặn thread1 từ A đang thực thi trong khi khi thread2 đang thực thi B. Điều này đúng bất kể liệu thread1 và thread2 có cùng thực thể của đối tượng hoặc không.

Cuối cùng, khái niệm then chốt cần nhớ ở đây là chỉ một phương thức đồng bộ hóa có thể được thực thi cho mỗi thực thể của đối tượng. Threads khác có thể thực thi phương thức không đồng bộ trên thực thể đó, hoặc chúng có thể thực thi bất kì phương thức nào trên một thực thể khác của đối tượng.