BÀI TẬP CHƯƠNG 6 – ĐỒNG BỘ TIẾN TRÌNH

6.1 Trong Phần 6.4, chúng ta đã đề cập rằng việc vô hiệu hóa các ngắt thường xuyên có thể ảnh hưởng đến đồng hồ của hệ thống. Giải thích tại sao điều này có thể xảy ra và làm thế nào để giảm thiểu các hiệu ứng như vậy.

6.2 Ý nghĩa của thuật ngữ busy waiting (bận chờ đợi) là gì? Có những loại chờ đợi khác trong một hệ điều hành không? Có thể tránh "bận chờ đợi" hoàn toàn không? Giải thích.

6.3 Giải thích tại sao spinlocks không phù hợp với các hệ thống vi xử lý đơn nhân và thường được sử dụng trong các hệ thống vi xử lý đa nhân (hoặc đa vi xử lý).

6.4 Chỉ ra rằng, nếu các hoạt động semaphore wait() và signal() không được thực thi đơn nguyên, thì tính chất loại trừ lẫn nhau (mutual exclusion) có thể bị vi phạm.

6.5 Minh họa làm thế nào một semaphore nhị phân có thể được sử dụng để thực hiện loại trừ lẫn nhau (mutual exclusion) giữa các tiến trình.

6.6 Tình trạng cạnh tranh (race condition) có thể xuất hiện trong nhiều hệ thống máy tính. Hãy xem xét một hệ thống ngân hàng duy trì số dư tài khoản với hai hàm thực thi: deposit(số tiền) và withdraw(số tiền). Hai hàm này được truyền vào số tiền sẽ được gửi hoặc rút từ số dư tài khoản ngân hàng. Giả sử rằng người chồng và người vợ chia sẻ một tài khoản ngân hàng. Một cách đồng thời, người chồng gọi hàm withdraw() và người vợ gọi hàm deposit(). Mô tả làm thế nào một tình trạng cạnh tranh có thể xảy đến và làm cách nào để ngăn chặn tình trạng cạnh tranh này xảy ra.

6.7 Mã giả của Hình 6.15 minh họa các hoạt động push() và pop() cơ bản của ngăn xếp dựa trên mảng. Giả sử rằng thuật toán này có thể được sử dụng trong môi trường thực thi đồng thời, hãy trả lời các câu hỏi sau:

a. Dữ liệu nào có tình trạng cạnh tranh?

b. Làm thế nào tình trạng cạnh tranh có thể được sửa chữa?

```
push(item){
      if (top < SIZE){
            stack[top] = item;
            top++;
      }
      else
      ERROR
      }

pop(){
      if (!isempty()){
            top--;
            return stack[top];
```

```
      }
      else
            ERROR
      }

isempty(){
      if (top == 0)
            return true;
      else
            return false;
      }
```

6.8 Tình trạng cạnh tranh (race condition) có thể xuất hiện trong nhiều hệ thống máy tính. Hãy xem xét một hệ thống đấu giá trực tuyến trong đó giá đấu cao nhất hiện thời cho mỗi mặt hàng phải được duy trì. Một người muốn đặt giá đấu cho một mặt hàng sẽ gọi hàm bid(số tiền), hàm này sẽ so sánh số tiền đang được đặt giá đấu với giá đấu cao nhất hiện tại. Nếu số tiền vượt quá giá đấu cao nhất hiện thời, giá đấu cao nhất sẽ được đặt thành số tiền mới. Điều này được minh họa dưới đây:

```
void bid(double amount){
      if (amount > highestBid)
            highestBid = amount;
}
```

Mô tả làm thế nào một tình trạng cạnh tranh có thể xảy ra trong tình huống này và những gì có thể được thực hiện để ngăn chặn tình trạng cạnh tranh đó.

6.9 Ví dụ chương trình sau đây có thể được sử dụng để tính tổng các giá trị của N phần tử trong mảng một cách song song trên một hệ thống chứa N nhân tính toán (có một bộ xử lý riêng cho từng thành phần mảng):

```
for j=1 to log2(N){
      for k=1 to N {
            if ((k + 1) % pow(2,j) == 0){
                  values[k] += values[k - pow(2,(j-1))]
            }
      }
}
```

Điều này sẽ biến việc tính tổng các phần tử trong mảng thành một chuỗi các tổng thành phần, như trong hình dưới đây. Sau khi đoạn mã thực thi xong, tổng của tất cả các phần tử trong mảng được lưu trữ ở vị trí cuối cùng của mảng.

Có bất kỳ tình trạng cạnh tranh nào trong ví dụ này không? Nếu vậy, xác định vị trí chúng xảy ra và minh họa bằng một ví dụ. Nếu không, hãy chứng minh tại sao thuật toán này không bị tình trạng cạnh tranh.

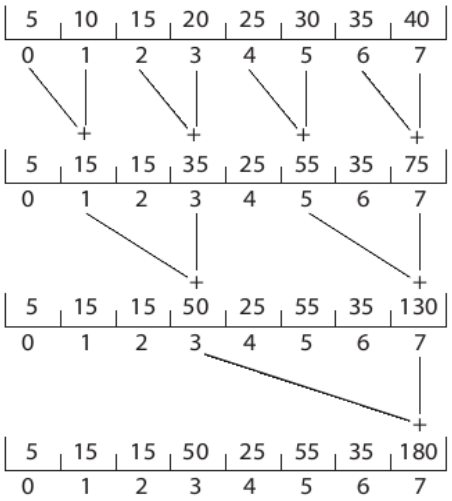

Figure 6.17 Summing an array as a series of partial sums for Exercise 6.14.

6.10 Lệnh CAS `compare_and_swap()` có thể được sử dụng để thiết kế các cấu trúc dữ liệu không khóa (lock-free) như ngăn xếp, hàng đợi và danh sách. Ví dụ chương trình được hiển thị sau đây trình bày một giải pháp khả thi cho ngăn xếp không khóa bằng cách sử dụng các lệnh CAS, trong đó ngăn xếp được biểu diễn dưới dạng danh sách các phần tử Node được liên kết với đỉnh đại diện cho đỉnh của ngăn xếp. Cách hiện thực này có tránh được tình trạng cạnh tranh không?

```
typedef struct node {
value_t data;
struct node *next;
} Node;

Node *top; // top of stack

void push(value_t item) {
      Node *old_node;
      Node *new_node;
      new_node = malloc(sizeof(Node));
      new_node->data = item;
      do {
            old_node = top;
            new_node->next = oldnode;
      } while (compare_and_swap(top,old_node,new_node) != old_node);
}

value_t pop() {
      Node *old_node;
      Node *new_node;
      do {
            old_node = top;
            if (old_node == NULL)
```

```
                return NULL;
            new_node = old_node->next;
      } while (compare_and_swap(top,old_node,newnode) != old_node);
      return old_node->data;
}
```

6.11 Một cách tiếp cận để ứng dụng lệnh compare_and_swap() vào hiện thực spin-lock được thể hiện như sau:

```
void lockspinlock(int *lock){
      while (compareandswap(lock, 0, 1) != 0)
      ; /* spin */
}
```

Một cách tiếp cận thay thế được đề xuất diễn đạt như là "So sánh và so sánh-và-hoán đổi", kiểm tra trạng thái của khóa trước khi gọi hoạt động compare_and_swap(). (Cơ sở lý luận đằng sau phương pháp này là chỉ gọi compare_and_swap() chỉ khi khóa đã có sẵn). Chiến lược này được thể hiển dưới đây:

```
void lock_spinlock(int *lock){
      {
      while (true){
      if (*lock == 0){
            /* lock appears to be available */
            if (!compare_and_swap(lock, 0, 1))
                  break;
            }
      }
}
```

Liệu cách diễn đạt "So sánh và so sánh-và-hoán đổi" có hoạt động phù hợp để hiện thực spinlocks không? Nếu có, giải thích. Nếu không, hãy minh họa khi nào thì tính toàn vẹn của khóa bị xâm phạm.

6.12 Một số hiện thực semaphore cung cấp hàm getValue() trả về giá trị hiện tại của semaphore. Ví dụ, hàm này có thể được gọi trước khi gọi wait() để một tiến trình sẽ chỉ gọi wait() nếu giá trị của semaphore đang là dương, do đó tránh bị khóa trong khi chờ semaphore. Ví dụ:

```
      if (getValue (& sem)> 0)
      wait(& sem);
```

Nhiều lập trình viên lập luận chống lại chức năng như vậy và không khuyến khích sử dụng nó. Hãy mô tả một vấn đề tiềm ẩn có thể xảy ra khi sử dụng hàm getValue() trong tình huống vừa mô tả.

6.13 Giải pháp phần mềm đúng đắn được biết đến đầu tiên để giải quyết bài toán vùng tranh chấp (critical-section) cho 2 tiến trình được Dekker phát triển. Hai tiến trình, P0 và P1, chia sẻ các biến sau:

```
boolean flag[2]; /* initially false */
int turn;
```

Cấu trúc của mỗi tiến trình như sau:

```
while (true) {
      flag[i] = true;
      while (flag[j]) {
            if (turn == j) {
                  flag[i] = false;
                  while (turn == j); /* do nothing */
                  flag[i] = true;
            }
      }
      /* critical section */
      turn = j;
      flag[i] = false;
      /* remainder section */
}
```

Chứng minh rằng thuật toán thỏa mãn tất cả ba yêu cầu cho vấn đề vùng tranh chấp.

6.14 Giải pháp phần mềm đúng đắn được biết đến đầu tiên để giải quyết bài toán vùng tranh chấp (critical-section) cho n tiến trình với giới hạn cận dưới chờ n-1 lượt được trình bày bởi Eisenberg và McGuire. Các tiến trình chia sẻ các biến sau:

```
enum pstate{idle, want_in, in_cs};
pstate flag[n];
int turn;
```

Tất cả các yếu tố bên ngoài enum ban đầu nhàn rỗi. Giá trị ban đầu của lượt là không quan trọng (trong khoảng từ 0 đến n-1). Cấu trúc của quá trình Pi được thể hiện như sau. Chứng minh rằng thuật toán thỏa mãn cả ba yêu cầu cho vấn đề vùng tranh chấp.

```
while (true) {
      while (true) {
            flag[i] = want_in;
            j = turn;
            while (j != i) {
                  if (flag[j] != idle) {
                        j = turn;
                  else
                  j = (j + 1) % n;
                  }
                  flag[i] = in cs;
            j = 0;
```

```
            while ( (j < n) && (j == i || flag[j] != in cs))
                  j++;
            if ( (j >= n) && (turn == i || flag[turn] == idle))
                  Break;
            }
      /* critical section */
      j = (turn + 1) % n;
      while (flag[j] == idle)
            j = (j + 1) % n;
      turn = j;
      flag[i] = idle;
      /* remainder section */
}
```

6.15 Giải thích tại sao hiện thực các nguyên hàm đồng bộ hóa bằng cách vô hiệu hóa các ngắt không phù hợp trong hệ thống vi xử lý đơn nhân nếu các nguyên tắc đồng bộ hóa được sử dụng trong các chương trình người dùng.

6.16 Xem xét cách thực hiện khóa mutex bằng cách sử dụng lệnh compare_and_swap(). Giả sử rằng cấu trúc dữ liệu sau đây đã định nghĩa khóa mutex có sẵn:

```
typedef struct {
      int available;
      } lock;
```

Giá trị (available == 0) chỉ ra rằng khóa có sẵn và giá trị 1 cho biết rằng khóa không khả dụng. Sử dụng cấu trúc này, minh họa cách các chức năng sau có thể được thực hiện bằng cách sử dụng lệnh compare_and_swap():

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Hãy chắc chắn bao gồm các khởi tạo cần thiết.

6.17 Giải thích tại sao các ngắt không thích hợp để hiện thực các nguyên hàm đồng bộ hóa trong các hệ thống đa vi xử lý.

6.18 Việc triển khai các khóa mutex được cung cấp trong Mục 6.5 phải busy waiting (bận chờ đợi). Mô tả những thay đổi nào là cần thiết để một tiến trình chờ đợi để có được khóa mutex sẽ bị chặn và được đặt vào hàng đợi cho đến khi khóa có thể dùng trở lại.

6.19 Giả sử rằng một hệ thống có nhiều nhân xử lý. Đối với mỗi trường hợp sau đây, hãy mô tả cơ chế khóa nào tốt hơn: khóa spinlock hay khóa mutex? Trong đó tiến trình chờ đợi đang ngủ trong khi chờ khóa có thể dùng trở lại.

- Khóa sẽ được giữ trong một thời gian ngắn.
- Khóa sẽ được giữ trong một thời gian dài.
- Một tiểu trình có thể được đưa vào trạng thái ngủ trong khi đang giữ khóa.

6.20 Giả sử rằng một lần chuyển ngữ cảnh mất T thời gian. Đề xuất cận trên của T để giữ một spinlock. Nếu spinlock bị giữ lâu hơn, thì nên chọn sang phương pháp khóa mutex (nơi các tiểu trình chờ được đưa vào chế độ ngủ).

6.21 Một máy chủ web đa luồng muốn theo dõi số lượng yêu cầu mà nó phục vụ (còn gọi là *hits*). Xem xét hai chiến lược sau đây để ngăn chặn tình trạng cạnh tranh trên các lượt truy cập biến *hits*. Chiến lược đầu tiên là sử dụng khóa mutex cơ bản khi cập nhật *hits*:

```
int hits;
mutex_lock hit_lock;
hit_lock.acquire();
hits++;
hit_lock.release();
```

Chiến lược thứ hai là sử dụng một biến số đơn nguyên:

```
atomic_t hits;
atomic_inc(&hits);
```

Giải thích chiến lược nào trong hai chiến lược trên hiệu quả hơn.

6.22 Xem xét đoạn mã ví dụ sau đây, để phân bổ và giải phóng các tiến trình được hiển thị đoạn mã bài 6.14.

a. Xác định (các) tình trạng cạnh tranh.

b. Giả sử bạn có khóa mutex được đặt tên với các thao tác acquire() và release(). Chỉ ra nơi khóa cần phải được đặt để ngăn chặn (các) tình trạng cạnh tranh.

c. Có thể thay thế biến số nguyên số

```
int number of processes = 0
```

với số đơn nguyên

```
atomict number of processes = 0
```

để ngăn chặn (các) tình trạng cạnh tranh?

6.23 Máy chủ có thể được thiết kế để giới hạn số lượng kết nối mở. Ví dụ, một máy chủ có thể chỉ muốn có kết nối N socket tại bất cứ thời điểm nào. Ngay khi kết nối N được thực hiện, máy chủ sẽ không chấp nhận kết nối đến khác cho đến khi có một kết nối hiện thời nào đó kết thúc. Minh họa cách sử dụng semaphores cho máy chủ để hạn chế số lượng kết nối đồng thời.

```
#define MAXPROCESSES 255
int numberofprocesses = 0;

/* the implementation of fork() calls this function */

int allocateprocess(){
```

```
      int newpid;

      if (numberofprocesses == MAXPROCESSES)
            return -1;
      else{
            /* allocate necessary process resources */
            ++numberofprocesses;
            return newpid;
      }
}
/* the implementation of exit() calls this function */
void releaseprocess(){
/* release process resources */
      --numberofprocesses;
}
```

6.24 Trong Mục 6.7, chúng tôi sử dụng hình minh họa sau đây là cách sử dụng không chính xác semaphores để giải quyết vấn đề phần quan trọng:

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

Giải thích tại sao đây là một ví dụ về sự thất bại của sự sống còn.

6.25 Chứng minh rằng bộ quan sát và semaphores tương đương với mức độ rằng chúng có thể được sử dụng để hiện thực các giải pháp cho các bài toán đồng bộ cùng loại với nhau.

6.26 Mô tả cách hoạt động signal() gắn liền trong bộ quan sát khác với các hoạt động tương ứng đã được xác định cho semaphores như thế nào.

6.27 Giả sử câu lệnh signal() chỉ có thể xuất hiện như câu lệnh cuối cùng trong một hàm quan sát. Đề xuất cách thực hiện được mô tả trong Mục 6.7 có thể được đơn giản hóa trong tình huống này.

6.28 Hãy xem xét một hệ thống hiện tại của các tiến trình P1, P2, … , Pn, mỗi tiến trình có một số ưu tiên duy nhất. Viết một bộ quan sát phân bổ ba máy in giống hệt nhau cho các tiến trình này, sử dụng các số ưu tiên để quyết định thứ tự phân bổ.

6.29 Một tập tin sẽ được chia sẻ giữa các tiến trình khác nhau, mỗi tiến trình có một chỉ số duy nhất. Tập tin có thể được truy cập đồng thời bởi một số tiến trình, tuân theo các ràng buộc sau: tổng của tất cả các chỉ số duy nhất được liên kết với tất cả các tiến trình hiện đang truy cập tập tin phải nhỏ hơn n. Viết một hàm quan sát để phối hợp các truy cập vào tập tin.

6.30 Khi tín hiệu được thực hiện trên một điều kiện bên trong bộ quan sát, tiến trình báo hiệu sẽ tiếp tục thực thi hoặc chuyển điều khiển sang tiến trình được báo hiệu. Làm thế nào giải pháp cho bài tập trước khác nhau với hai cách khác nhau trong đó tín hiệu có thể được thực hiện?

6.31 Thiết kế thuật toán cho một bộ quan sát thực hiện một đồng hồ báo thức cho phép một chương trình được gọi tự trì hoãn trong một số đơn vị thời gian xác định (ticks). Bạn có thể giả sử sự tồn tại của một đồng hồ phần cứng thực sự gọi hàm tick() trong bộ quan sát của bạn theo định kỳ.

6.32 Thảo luận về các cách giải quyết vấn đề đảo ngược ưu tiên trong hệ thống thời gian thực. Cũng thảo luận về việc các giải pháp có thể được thực hiện trong bối cảnh của một lịch trình chia sẻ tỷ lệ.