下列二线程同步机制是否有误？请给出分析：

（1）即17.4第一次尝试

CONCEPT: A shared variable named turn is used to keep track of whose turn it is to enter the critical section.

INITIALIZATION:

shared int turn;

...

turn = i ;

ENTRY PROTOCOL (for Process i ):

/\* wait until it's our turn \*/

while (turn != i ) {

}

EXIT PROTOCOL (for Process i ):

/\* pass the turn on \*/

turn = j ;

满足“忙则等待”，但是有时不满足“空闲则入”。Ti不在临界区，Tj想要继续运行，但是必须等待Ti进入过临界区后。

（2）即17.4第二次尝试

CONCEPT: A shared Boolean array named flags contains a flag for each process. The flag values are BUSY when the process is in its critical section (using the resource), or FREE when it is not.

INITIALIZATION:

typedef char boolean;

...

shared boolean flags[n - 1];

...

flags[i ] = FREE;

...

flags[j ] = FREE;

...

ENTRY PROTOCOL (for Process i ):

/\* wait while the other process is in its CS \*/

while (flags[j ] == BUSY) {

}

-->

/\* claim the resource \*/

flags[i ] = BUSY;

EXIT PROTOCOL (for Process i ):

/\* release the resource \*/

flags[i ] = FREE;

两个线程先后判断，之后，先后置位，就都进入临界区了，故不满足“忙则等待”。

（3）即17.4第三次尝试

CONCEPT: Again we use a shared Boolean array as in Algorithm 2. Each process sets its flag before testing the other flag, thus avoiding the problem of violating mutual exclusion.

INITIALIZATION:

typedef char boolean;

...

shared boolean flags[n -1];

...

flags[i ] = FREE;

...

flags[j ] = FREE;

...

ENTRY PROTOCOL (for Process i ):

/\* claim the resource \*/

flags[i ] = BUSY;

-->

/\* wait if the other process is using the resource \*/

while (flags[j ] == BUSY) {

}

EXIT PROTOCOL (for Process i ):

/\* release the resource \*/

flags[i ] = FREE;

满足“忙则等待”，但是不满足“空闲则入”

两个线程先后置位，之后，先后判断，都陷入while循环，都不能进入临界区，故不满足“空闲则入”。

（4）

CONCEPT: To avoid the deadlock problem of Algorithm 3, we periodically clear and reset our own flag while waiting for the other one.

INITIALIZATION:

typedef char boolean;

...

shared boolean flags[n -1];

...

flags[i ] = FREE;

...

flags[j ] = FREE;

...

ENTRY PROTOCOL (for Process i ):

/\* claim the resource \*/

flags[i ] = BUSY;

-->

/\* wait if the other process is using the resource \*/

while (flags[j ] == BUSY) {

flags[i ] = FREE;

delay a while ;

flags[i ] = BUSY;

}

EXIT PROTOCOL (for Process i ):

/\* release the resource \*/

flags[i ] = FREE;

依然有概率出现不能“空闲则入”的情况，但能够解决，效率较低

（5）

CONCEPT: Both the turn variable and the status flags are combined in a way which we (the requesting process) set our flag and then check our neighbor's flag.

INITIALIZATION:

typedef char boolean;

...

shared boolean flags[n -1];

shared int turn;

...

turn = i ;

...

flags[i ] = FREE;

...

flags[j ] = FREE;

...

ENTRY PROTOCOL (for Process i ):

/\* claim the resource \*/

flags[i ] = BUSY;

/\* wait if the other process is using the resource \*/

while (flags[j ] == BUSY) {

/\* if waiting for the resource, also wait our turn \*/

if (turn != i ) {

/\* but release the resource while waiting \*/

flags[i ] = FREE;

while (turn != i ) {

}

flags[i ] = BUSY;

}

}

EXIT PROTOCOL (for Process i ):

/\* pass the turn on, and release the resource \*/

turn = j ;

flags[i ] = FREE;

这个的问题和（1）的一样吧——满足“忙则等待”，但是有时不满足“空闲则入”。Ti不在临界区，Tj想要继续运行，但是必须等待Ti进入过临界区后。

（6）

CONCEPT: Both the turn variable and the status flags are used.

INITIALIZATION:

typedef char boolean;

...

shared boolean flags[n -1];

shared int turn;

...

turn = i ;

...

flags[i ] = FREE;

...

flags[j ] = FREE;

...

ENTRY PROTOCOL (for Process i ):

/\* claim the resource \*/

flags[i ] = BUSY;

/\* give away the turn \*/

turn = j ;

/\* wait while the other process is using the resource \*and\* has the turn \*/

while ((flags[j ] == BUSY) && (turn != i )) {

}

EXIT PROTOCOL (for Process i ):

/\* release the resource \*/

flags[i ] = FREE;

没看出来有什么问题……