Bounded Waiting Solution of CS using TSL

// both variables are **shared** between processes

Boolean waiting[n]; // initially false  
Boolean lock;  
-------------------------------

// this is **local** variable

Boolean key;

Pi:

Waiting[i] = true;  
 key = true;  
 while (waiting[i] && key)  
 key = TSL(lock);  
 waiting[i] = false;  
 // CS  
 j = (i + 1) % n;  
 while ((j != i) && !waiting[j])  
 j = (j + 1) % n;  
 if (j == i)  
 lock = false;  
 else  
 waiting[j] = false;

Have the mod n because we want j to loop back around, because we want to be *bounded by length* ***n***.

If there is process k that is waiting, then instead of toggling **lock** we set k’s waiting to false so that it’s loop exits at the top.

Disjointness

P1 P2  
 -> [ ] -> [ ]  
 S1 S2  
 I(S1) I(S2) // each has input/output set  
 O(S1) O(S2)  
 S1 + S2 disjoint if the following three conditions are true:  
 I(S1) ^ O(S2) = null // ^ intersection  
 I(S2) ^ O(S1) = null  
 O(S1) ^ O(S2) = null  
  
*if this holds then okay to* ***run in parallel***

Example

S1: A = B + C  
S2: X = Y + Z

I(S1) = {B, C} // what it reads  
O(S1) = {A} // what it writes/modifies

I(S2) = {Y, Z}  
O(S2) = {x}

Thus, there are no similar elements in each set so it will run successfully in parallel **with no race condition.**

Example

S3: A = B + C  
S4: X = A + Y;  
  
I(S3) = {B, C}  
O(S3) = {A} // A is here

I(S4) = {A, Y} // and A is also here  
O(S4) = {X}

Thus, O(S3) ^ I(S4) != null, and **cannot** run without caution in parallel.  
----------------------------------------------------------------------------------------------------------

*As seen in the 3 conditions, even if variables are shared within the* ***input*** *sets, race condition does not occur between reads.*

*If a variable appears in its own assignment, it will only appear in that set number’s* ***output set****.*

Example

S9: A = A + B  
S10: C = A + X

I(S9) = {B} // took out A from here  
O(S9) = {A} // collision here

I(S10) = {A, X} // with here  
O(S10) = {C}

Read(a, b, c) => he means cin, thus write to a, b, c  
write(x,y,a) => he means cout, thus read from x,y, a // collision with a

**This will be on the test—we will verify disjointness.  
--------------**

Semaphore Implementation (Dijkstra’s biggest invention)

Common Drawback of all CS solutions discussed so far

While loop

Pi:  
 while (condition) do [skip];  
**busy waiting**

Stops busy waiting when:  
 1) condition is false  
 2) other process leaves timeslice early and gets CPU

Busy waiting wastes cpu cycles **and memory bandwidth** because uses memory **bus** to keep reading @ while loop

Pj can get into CS but not finish due to **running out of timeslice (interrupt)** and then a Pi could waste cycles and memory (because Pj still in CS). (this is the one Jake answered the opposite to)

Now Pi waste only because of timeslice.

**3rd** problem:

L low priority, H high

Priority scheduler is preemptive  
  
Principle rule: Run L **iff** H is not ready.

Example

H is not ready  
L is ready  
L is selected to run  
L runs  
L enters CS  
L is interrupted  
H becomes ready  
L is preempted  
H is scheduled  
H runs  
H cannot enter CS and **begins busy waiting**

Thus a **special type of deadlock (3rd problem of busy waiting)**: called *Priority Inversion Problem*

Spinlocking => method of implementing CS that *allows* busy waiting.