13. * The Bounded-Buffer problem *

* It is commonly used to illustrate the power of synchronization primitives.
* We present here a general structure of this scheme without committing ourselves to any particular implementation.
* We provide a related programming project in the exercise at the end of the Synchronization.
* We assume that the pool consists of 'n' buffers.
* each capable of holding one item. The mutex Semaphore provides mutual exclusion for accesses to the buffer pool and is intialized to the value 1.
* The empty and full semaphores Count the number of empty and full buffers.
* The semaphore empty is intialized to the value n; the Semaphore full is intialized to the value 0.
* The code for the procedure process is the Code for the consumer process.
* Note the symmetry between the producer and the consumer.
* We can interpret this Code as the producer empty buffers for the producer.

```
do {
    wait (full);
    wait(mutex);
    . . .
    // remove an item from buffer to nextc
    . . .
    signal (mutex);
    signal (empty);
    . . .
    // Consume the item in nextc
    . . .
} while (TRUE);
```

- producer must block if the buffer is full.
- Consumers must block if the buffer is empty.

* <u>producer</u>

```
#Define B-S=8;
do {
    While ( Counter == B-S);
        Buffer [in] = i-p;
            in= (in+1) % B-S;
                Counter++;
```

}
while (TRUE);

* Consumer

```
do {
    while (Counter == 0);
        // buffer is empty;
    I-C = Buffer [out];
    out = (out + 1) % B-S;
    Counter --;
}
while (TRUE);
```

14) * PETERSON's SOLUTION *

*, We illustrate a Classic software-based Solution to the Critical Section problem known as peterson's Solution.

*, Because of the way modern Computer architectures perform basic Machine-language instructions, such as load and store, there are no guarantees that peterson's Solution will work correctly on such architectures.

*, Good algorithmic description of Solving the problem.

*, It Contains two process Solution.

*, Assume that the load and Store machine language instruction are atomic; that cannot be interrupted.

*, In this the two process share two variables.

- int turn;
- Boolean flag[2]

*, The variable turn indicates whose turn it is to enter the Critical Section.

*, The flag array is used to indicate if a process is ready to enter the Critical Section flag[i] = TRUE this value indicates that P_i is ready to enter its Critical Section.

* To enter the Critical section, process P_i first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can doso.

* If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time only one of these assignments will last; the other will occur but will be overwritten immediately.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        Critical section

    flag[i] = FALSE;

        remainder section

} While (TRUE);
```

The Structure of process P_i in peterson's Solution

* The eventual value of turn decides which of the two processes is allowed to enter its critical Section first.

We now prove that this solution is correct. We need to show that: Rob.

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

*. To prove property 1, we note that each P_i enters its critical section only. if either flag[j]= false (or) turn ==i.

*. Also note that, if both processes can be executing in their critical sections at the same time, then the flag[0]==flag[1] = = true.

*. To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] ==true and turn==j; this loop is the only one possible.

*. If P_j is not ready to enter the critical section then flag[j]==false, and P_i can enters its critical section.

* P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

do {

acquire lock

critical section

release lock

remainder section

} while (TRUE);

Solution to the Critical Section problem using locks.

15) * Race condition in operating System *

A race condition is an undesirable situation that occurs when a device or System attempts to perform two or more operations at the same time.

But because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Race conditions are Considered a Common issue for multithreaded applications.

For example;

- The computer crashes or identifies an illegal operation of the program.
- Errors reading the old data.
- Errors writing the new data.

Race condition:

- counter ++ could be implemented as

register 1 = counter

register 1 = register 1 + 1

Counter = register 1.

• counter -- could be implemented as

register 2 = counter
register 2 = register - 1
counter = register.

* Following the execution of these two statements the value of variable counter may be 4, 5, 6 the only correct result is Counter = 5 which is generated correctly if they are executed seperately. Such condition is called race condition.

Problem of race condition:

int shared = 5 → (They are sharing. the variable)

P1
{
Int x=shared;
x++;
Sleep(1); ← Race Condition
Shared = x;
}
∴ (Shared = 6)

P2
{
Int y=Shared;
y--;
Sleep(1);
Shared=y;
}
∴ (Shared = 4)
↓
PCB Content Switching.