Chapter 5: Process Synchronization
while (true) {
    while (count == BUFFER_SIZE);
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

while (true) {
    while (count == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
}

Producer & Consumer
Race Condition

- `count++` could be implemented as
  
  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- `count--` could be implemented as
  
  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```

- Consider this execution interleaving with “count = 5” initially:
  
  S0: producer execute `register1 = count` {register1 = 5}
  S1: producer execute `register1 = register1 + 1` {register1 = 6}
  S2: consumer execute `register2 = count` {register2 = 5}
  S3: consumer execute `register2 = register2 - 1` {register2 = 4}
  S4: producer execute `count = register1` {count = 6}
  S5: consumer execute `count = register2` {count = 4}
Solution to Critical-Section Problem

1. Mutual Exclusion - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted:
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the $N$ processes.
Peterson’s Solution

- Two process solution
- The two processes 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` implies that process $P_i$ is ready!
Algorithm for Process $P_i$

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = FALSE;
    remainder section
} while (TRUE);
```
Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
    - Either test memory word and set value
    - Or swap contents of two memory words
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
TestAndndSet Instruction

Definition:

```c
boolean TestAndSet (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```
Solution using TestAndSet

- Shared boolean variable lock, initialized to false.
- Solution:

```c
    do {
        while ( TestAndSet (&lock ) )
            ; // do nothing

        // critical section

        lock = FALSE;

        // remainder section

    } while (TRUE);
```
Swap Instruction

- Definition:

```c
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```c
        do {
            key = TRUE;
            while ( key == TRUE)  
                Swap (&lock, &key );

            // critical section

            lock = FALSE;

            // remainder section

        } while (TRUE);
```