Test-And-Set, Lock

Peterson’s Solution for a 2-process case

```java
int turn;
boolean flag[2];
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = FALSE;
    remainder section
} while (TRUE);
```

Peterson’s Solution
Process 0

```java
int turn;
boolean flag[2];
do {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = FALSE;
    remainder section
} while (TRUE);
```

Process 1

```java
int turn;
boolean flag[2];
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = FALSE;
    remainder section
} while (TRUE);
```

Limitation to Peterson’s Solution

- Strict order of execution
- Variable updates (turn and flag) could still be problematic

Where Are the Sources of the Problem?

The root cause of the problem is that we are unable to control which part of the code can be executed in parallel, which part can only be executed in sequence.

For example, the instructions that update the value of a shared variable should only be allowed to execute in sequence.
Using Locks

hardware or software

do {
   acquire lock
   critical section
   release lock
   remainder section
} while (TRUE);

Assumption: the operations acquire_lock and release_lock are atomic

Synchronization Hardware

• Many systems provide hardware support for critical section code.
• Uniprocessors (could disable interrupts):
  – Currently running code would execute without preemption.
  – Generally too inefficient on multiprocessor systems.
  – Operating systems using this not broadly scalable.
• Modern machines provide special atomic hardware instructions:
  – Test memory word and set value.
  – Swap the contents of two memory words.

Lock with TestAndSet

boolean lock = FALSE;
do {
   while (TestAndSet(&lock));
critical section
   lock = FALSE;
   remainder section
} while (TRUE);

The process which wants to get into CR attempts to set lock = FALSE (unlock) if the lock was TRUE, then TestAndSet() returns TRUE, the requesting process will be busy waiting, until the lock becomes FALSE before entering CR.

TestAndSet

boolean TestAndSet(boolean *target) {
   boolean ret_val = *target;
   *target = TRUE;
   return ret_val;
}

When Multiple Processes Do the Same …

Only one will get through the while loop, i.e., the lock == False

Lock with Swap

boolean lock = FALSE; // try to unlock
boolean key;
do {
   key = TRUE;
   while (key == TRUE) Swap(&lock, &key);
   //now key == FALSE, lock == TRUE
   critical section
   lock = FALSE;
   remainder section
} while (TRUE);
Swap

void Swap (boolean *a, boolean *b) {
    // swap() is an atomic operation
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

How Are We Meeting The Requirements?

Do the solutions above provide:

1. Mutual exclusion?
2. Progress?
3. Bounded waiting?