CSCI315 – Operating Systems Design

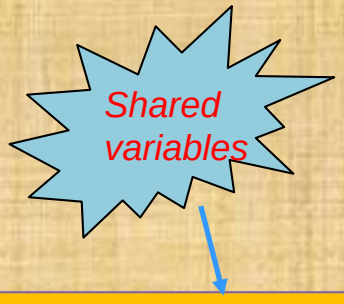
Department of Computer Science
Bucknell University

Synchronization Tools: test_and_set()

Ch 6.4-6.5

This set of notes is based on notes from the textbook authors, as well as L. Felipe Perrone, Joshua Stough, and other instructors.
Xiannong Meng, Fall 2021.

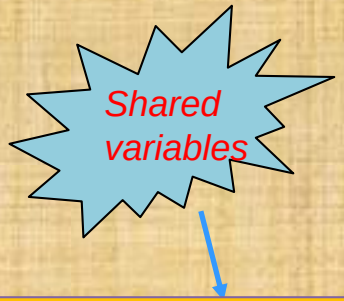
Peterson's Solution
for a 2-process case




```
do {
  flag[i] = TRUE; // i 0 or 1
  turn = j;       // j 0 or 1
  while (flag[j] && turn == j);
        critical section
  flag[i] = FALSE;
        remainder section
} while (TRUE);
```

Peterson's Solution

Process 0



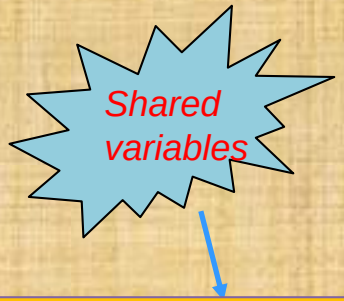

```
int turn;
boolean flag[2];
```

turn: status
flag[2]: intension

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

Peterson's Solution

Process 1




```
int turn;
boolean flag[2];
```

turn: status
flag[2]: intension

```
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.

We'll look at some solutions in this segment.

Using Locks
hardware or software

```
do {
   acquire_lock
   critical section
   release_lock
   remainder section
} while (TRUE);
```

Key: the operations ***acquire_lock*** and ***release_lock*** are atomic, i.e., they either complete or do nothing.

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 (dedicated instructions) :
  - Test memory word and set value.
  - Swap the contents of two memory words.

Lock with test_and_set

```
boolean lock = FALSE; // try to unlock
do {
  while (test_and_set(&lock))
       ; //wait on TRUE
  critical section  // lock is FALSE, our turn
  lock = FALSE;     // release the lock
  remainder section
} while (TRUE);
```

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

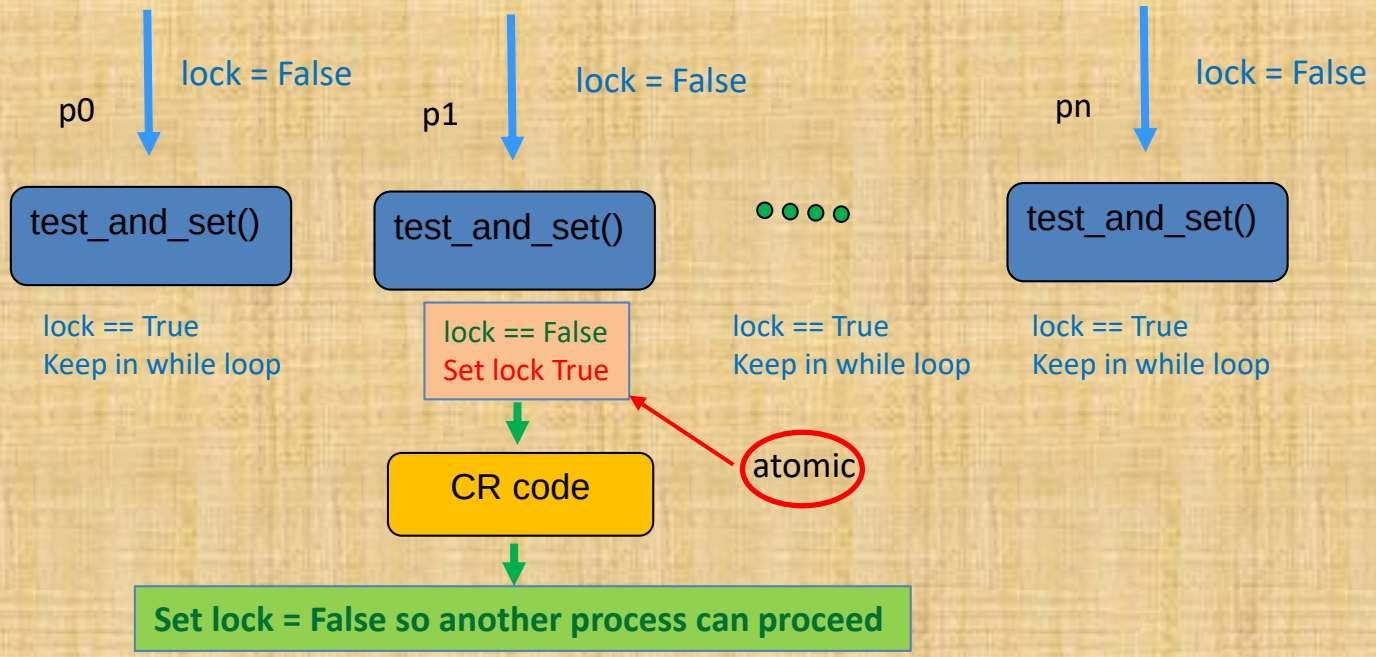
Atomic test_and_set

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

The above operations must be completed without interrupt, thus **atomic**. Only the very first process can get through this by getting a **False** return value. All subsequence processes will see **True** until the process in CR sets it to **False**.

When Multiple Processes Do the Same …

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



Lock with compare_and_swap

```
int lock = 0; // try to unlock
do {
   while(compare_and_swap(&lock,0,1) != 0)
       ;  // wait
   critical section
   lock = 0;  // release the lock
   remainder section
} while (TRUE);
```

Atomic compare_and_swap

```
void compare_and_swap (int *cur_value,
      int expected, int new_value) {
   int temp = *cur_value;  // current lock value
   if (*cur_value == expected) // we can lock
      *cur_value = new_value;
   return temp;
}
```

The above operations must be completed without interrupt, thus **atomic**.

How Are We Meeting The Requirements?

Do the solutions above provide:

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

Try out an example:
http://www.eg.bucknell.edu/~cs315/F2021/meng/code/locks/gnu_locks.c