Bounded waiting, mutex and progress with spinlock using TestAndSet

Module 4
self study material

Operating systems 2020

1DT003, 1DT044 and 1DT096

Spinlock

A **spinlock** is a lock where a task simply waits in a loop ("spins") repeatedly checking until the lock becomes available.

TestAndSet

The **TestAndSet** instruction **atomically** first **sets** the **value** at the target address **to True** and **return** the **old value** stored at the target address.

TestAndSet

Here we use C to define the semantics of the TestAndSet (TAS) instruction. Optimally, the TAS instruction is supported directly by the CPU.

```
boolean TestAndSet (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

TestAndSet set the value at the target address to True and return the old value at address target. The **TestAndSet** instruction must be **atomic**. If two such instructions are executed simultaneously on the same target address (each on a different CPU or core), they will be executed sequentially in some arbitrary order.

Spinlock using TestAndSet

Using **TestAndSet**, once we're allowed to enter the CS, the lock is **atomically** set to true blocking others from entering.
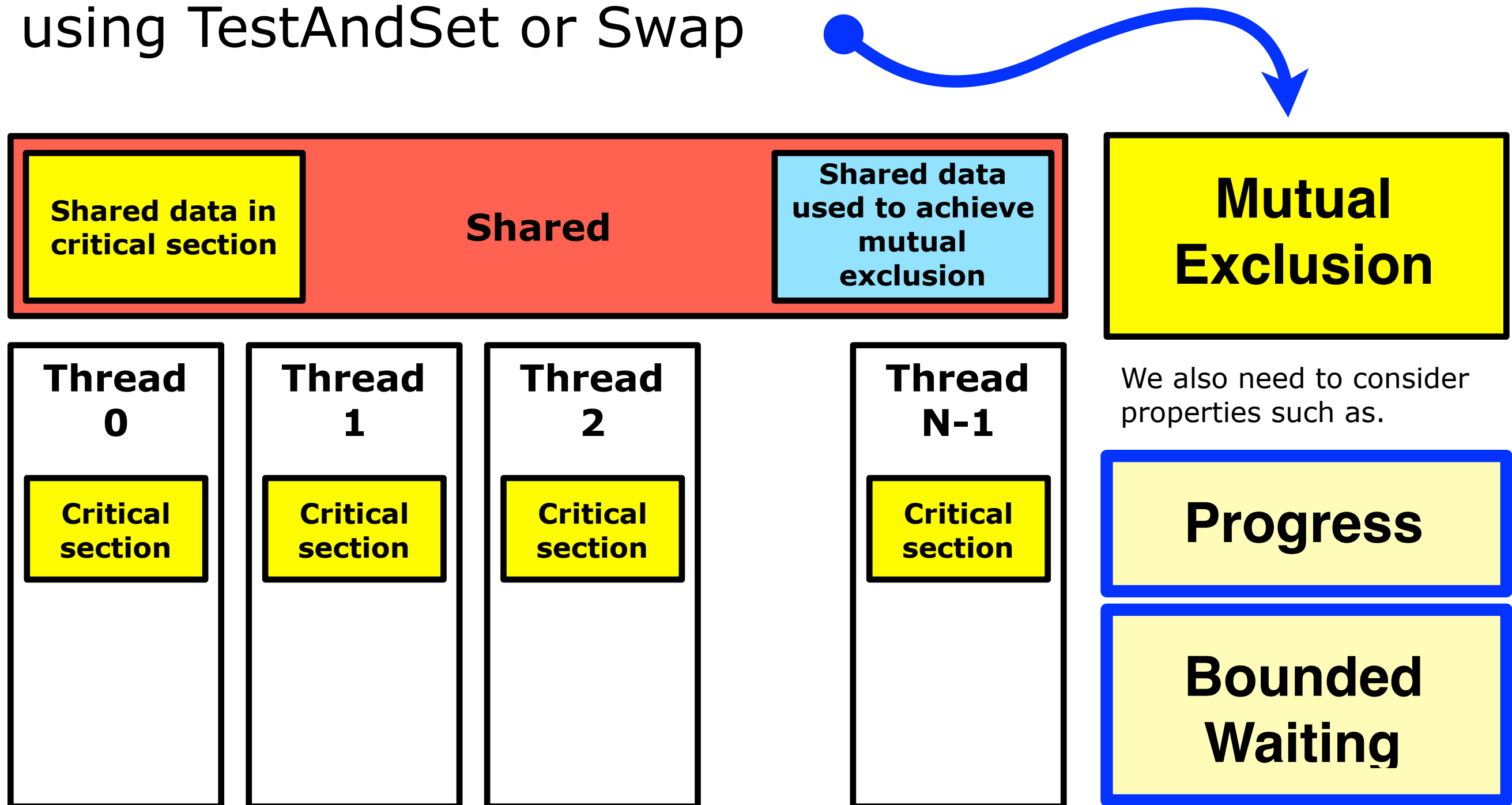
▶ Shared `boolean` variable **lock** initialized to **false**.

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

  /* critical section */

  lock = FALSE;

  /* remainder section */

} while (TRUE);
```

	lock
Before entering the critical section	**FALSE**
Inside the critical section	**TRUE**
After the critical section	**FALSE**

Solution to Critical-Section Problem

Mutual exclusion can be achieved
using TestAndSet or Swap

Shared

Shared data in critical section

Shared data used to achieve mutual exclusion

Mutual Exclusion

We also need to consider properties such as.

Progress

Bounded Waiting

Thread 0	Thread 1	Thread 2	Thread N-1
Critical section	Critical section	Critical section	Critical section

How can progress and bounded waiting be achieved?

Properties of critical sections

Assume that each process/thread executes at a nonzero speed. No assumption concerning relative speed of the N processes. In the following discussion the term **task** is used to mean a concurrent unit of execution such as a process or thread.

Mutual Exclusion

If task T_i is executing in its critical section, then no other tasks can be executing in their critical sections.

Bounded Waiting

A bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

Progress

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

Deadlock

Fairness

Starvation

We will come back to this ...

Implementing bounded waiting and progress using a shared flag array

Similar to the flag array used in Peterson's Solution, we introduce a shared array where each thread indicate with True if they want to enter the critical section.

	bool waiting[N]						
Thread	0	1	2	3	...	N-2	N-1
Value	False	False	True	False	...	False	True

In the above example only threads 2 and N-1 wants to enter the critical section.

Circular scan

When leaving the critical section, a thread scans the array full circle until another thread wanting to enter the critical section is found. Each thread **i** start the circular scan at index **i+1 % N**.

	bool waiting[N]						
Thread	0	1	2	3	...	N-2	N-1
Value	False	False	True	False	...	False	True

Bounded waiting

Worst case, all threads wants to enter the critical section. Any process waiting to enter its critical section will thus do so within **N-1** turns.

Bounded waiting

A bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

- Shared **boolean** `waiting[n]` initialized to **FALSE**.
- Shared **boolean** **lock** initialized to **FALSE**.
- Each thread has a private **boolean** variable **key**.

```
do {
```

Grab lock

```
/* critical section */
```

Release lock

```
} while (TRUE);
```

- Shared **boolean** **waiting[n]** with all elements initialized to **FALSE**.

- Shared **boolean** **lock** initialized to **FALSE**.

- Each thread has a private **boolean** variable **key**.

	bool waiting[N]						
Thread	0	1	2	3	...	N-2	N-1
Value	False	False	True	False	...	False	True

```
waiting[i] = TRUE;
```

```
do {
  waiting[i] = TRUE;
  key = TRUE;
```

Start to grab lock ...

... grab lock continues.

```
/* critical section */
```

Release lock

```
} while (TRUE);
```

- Shared **boolean `waiting[n]`** initialized to **FALSE**.
- Shared **boolean lock** initialized to **FALSE**.
- Each thread has a private **boolean** variable **key**.

```
do {
    waiting[i] = TRUE;
    key = TRUE;

    while (waiting[i] && key)
        key = TestAndSet(&lock);
```

Start to grab lock …

Mutual Exclusion

Pi can enter its critical section only if

  waiting[i] == FALSE || key == FALSE

The value of **key** can become false only if the **TestAndSet** is executed.

The first process to execute

  key = TestAndSet(&lock)

where,

  lock == FALSE

will cause:

  key ← FALSE
  lock ← TRUE

, entering the critical section and forcing all others to wait.

… grab lock continues.

```
    /* critical section */
```

Release lock

```
} while (TRUE);
```

- Shared **boolean** `waiting[n]` initialized to **FALSE**.
- Shared **boolean** **lock** initialized to **FALSE**.
- Each thread has a private **boolean** variable **key**.

No longer waiting, enter the critical section.

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&lock);

  waiting[i] = FALSE;

  /* critical section */


                    Release lock


} while (TRUE);
```

- Shared **boolean** `waiting[n]` initialized to **FALSE**.

- Shared **boolean** **lock** initialized to **FALSE**.

- Each thread has a private **boolean** variable **key**.

Bounded-Waiting

When a process leaves the critical section, it scans the waiting array in the cyclic order

(i+1, i+2, ..., n-1, 0, ..., i-1)

and designates the first process in this ordering that is in the entry section (**waiting[j] == true**) as the next one to enter the critical section.

Any process waiting to enter its critical section will thus do so within n - 1 turns.

```
do {
    waiting[i] = TRUE;
    key = TRUE;

    while (waiting[i] && key)
        key = TestAndSet(&lock);

    waiting[i] = FALSE;

    /* critical section */

    j = (i + 1) % n;

    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;  // No one is waiting
    else
        waiting[j] = FALSE;

    /* remainder section */

} while (TRUE);
```

Mutual exclusion

If task T_i is executing in its critical section, then no other tasks can be executing in their critical sections.

- Shared **boolean** `waiting[n]` initialized to **FALSE**.
- Shared **boolean** **lock** initialized to **FALSE**.
- Each thread has a private **boolean** variable **key**.

Mutual Exclusion

Pi can enter its critical section only if

    waiting[i] == FALSE || key == FALSE

The value of **key** can become false only if the **TestAndSet** is executed.

The first process to execute

        key = TestAndSet(&lock)

where,

        lock == FALSE

will cause:

        key ← FALSE

        lock ← TRUE

, entering the critical section and forcing all others to wait.

Mutual Exclusion

waiting[i] can become **FALSE** only if another process leaves its critical section, only one **waiting[i]** is set to false, maintaining the mutual-exclusion requirement.

```
do {
    waiting[i] = TRUE;
    key = TRUE;

    while (waiting[i] && key)
        key = TestAndSet(&lock);

    waiting[i] = FALSE;

    /* critical section */

    j = (i + 1) % n;

    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;  // No one is waiting
    else
        waiting[j] = FALSE;

    /* remainder section */

} while (TRUE);
```
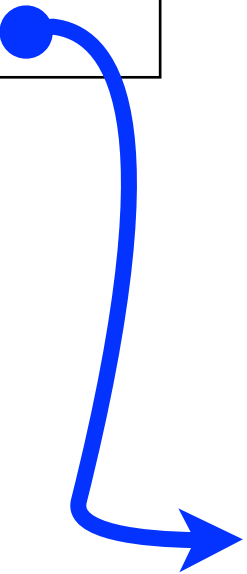
Progress

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

- Shared **boolean waiting[n]** initialized to **FALSE**.
- Shared **boolean lock** initialized to **FALSE**.
- Each thread has a private **boolean** variable **key**.

Progress

When a process leaves the critical section, either **lock** or **waiting[j]** is set to **FALSE**. Both allow a process that is waiting to enter its critical section.

```
do {
  waiting[i] = TRUE;
  key = TRUE;

  while (waiting[i] && key)
    key = TestAndSet(&lock);

  waiting[i] = FALSE;

  /* critical section */

  j = (i + 1) % n;

  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = FALSE;  // No one is waiting
  else
    waiting[j] = FALSE;

  /* remainder section */

} while (TRUE);
```