Algorithm for Process p_1

```
while (true) {
  flag[i] = true
  turn = j
  while (flag[j] && turn == j) {
    // critical section
    flag[i] = false
    //remainder section
  }
}
```

Peterson's Solution

Synchronization Hardware

- Many sys provide HW support for implementing the critical section code.
- Uniprocessors
  - Code would execute wi/out preemption
  - Too inefficient on multi-processor sys
    - OSs using this not scalable
- Hardware support
  - Memory barriers
  - Hardware instructions
  - Atomic variables

Hardware	**SW + HW**
Disable interrupts	mutex lock
Atomic instructions	semaphonre
-- test_and_set	monitor
-- compare_and_swap	

Users cannot block interrupts - must be kernel

No I/O inside critical section code - don't block anything. Can have a ContextSwitch though
Atomic: There may be a ContextSwitch, there may not be.

Memory Barriers

- An instruction that forces any change in memory to be propagated to all other processes
- Memory model: mem guarantees a comp. arch. makes to application programs
  - Strongly ordered: all mem mods of one processors is immediately visible to all other processes
  - Weakly ordered: all mem mods of one process may not be immediately visible to all other processes
- Could add a mem barrier to the following to ensure thread 1 outputs 100:
  t1 now does:

```
while(!flag):
  mem_barrier()
print x
```

t2 now does:

```
x = 100
mem_barrier()
flag = true
```

Hardware Instructions

Special HW instructions that *test-and-modify* the content of a word, or *swap* the contents of two words

- **Test-and-set** instruction
- **Compare and swap** instruction

test_and_set instruction

```
bool test_and_set(boolean* target) {
  bool rv = *target
  *target = true
  return rv
}
```

- Executed atomically
- Returns the og value of the passed param
- Set the new val of passed param to true

compare_and_swap

```
int compare_and_swap(int* value, int expected, int new_val) {
  int temp = *value

  if (*value == expected)
    *value = new_value
  return temp
}
```

- exec atomicaaly
- Return the og val of the passes param `val`
- Set the var `val` the val of the passed param `new_val`

Solution using test_and_swap

- Shared boolean variable `lock` , intialized to `false`
- Solution:

```
do {
  while test_and_set(&lock)
    // do nothing
      // crictical section
    lock = false
      // remainder section
} while true
```

Critical section problem:

- Satisfies Mutual exclusion problem
- Satisfied Progress problem
- Does NOT satisfy Bounded waiting problem
  - One process can grab the same lock over and over again
  - Process can starve.
  - Will never enter critical section because it can never lock
  - BUT, there's no deadlock or livelock

Solution using compare_and_swap

Shared int `lock` initialized to 0

```
while true:
  while compare_and_swap(&lock, 0, 1) != 0:
    //do nothing

  // critical section
  lock = 0
  //remainder section
```

Bounded waiting mutual-exclusion with compare_and_swap

```
while true:
  waiting[i] = true
  key = 1
  while waiting[i] and key == 1:
    key = compare_and_swap(&lock, 0, 1)
  waiting[i] = false

  //critical section

  j = (i+1) % n
  while (j != i) and !waiting[j]:
    j = (j + 1) % n

  if j == i:
    lock = 0
  else:
    waiting[j] = false

  //remainder section
```

- Solves critical section problem
- Satisfies bounded waiting problem
- Has to wait (at most) for n-1 processes to grab the lock. Checks processes in a round-robin way to grant lock.

Atomic Variables

- Typically, instructions such as `compare_and_swap` are used as building blocks for other synchronization tools

- One tool is an atomic variable that provides atomic (uninteruptible) updates on basic data types such as ints and bools
- ex: the `increment()` operation on the atomic var `sequence` ensures `sequence` is incremented without interruption

  `increment(&sequence)`

Consumer

```
while true:
  while counter == 0
    // do nothing
  next_consumed = buffer[out]
  out = (out + 1) % BUFFER_SIZE
  counter--
  //consume the item in next consumed
```

Race Conditions

- counter++

  could be: `reg1 = count; reg1 = reg1 + 1; count = reg1`
- counter--

  could be: `reg2 = count; reg2 = reg2 - 1; count = reg2`
- Consider:

  s0 producer exec reg1 = counter

  s1 producer exec reg1
- The `increment()` function can be implemented as:

```
void increment(atomic_int* v):
  int temp
  do:
    temp = *v
  while temp != compare_and_swap(v, temp, temp+1)
```

Mutex Locks

- Previous solutions are complicated and generally inaccessbile to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is Mutex Lock
- Protect a C.S by first `acquire()` a lock, then `release()` the lock

  - bool var indicating if lock is available or not
- Call to `acquire()` and `releae()` must be atomic
  - Usually implemeted via a hardware atomic instructions such as compare_and_swap
- But this solution requires **Busy waiting**
  - This lock is therefore called a **spinlock**

Definitions

```
acquire()
  while !available
    // busy wait
  available = false
```

```
release()
  available = true
```

these two funcs must be implemeted atomicaaly. Both test_and_set and compare_and_swap can be used to implement these functions.

Semaphore

- Synchronization tool that provides more sophisticated ways (than mutex locks) for process to sync their activities
- Semaphore S - integer var
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()`
    - (originally called P() and V())
- Definition of `wait()` and `signal()` operations

```
wait(S)
  while S <= 0
    // busy wait
  S--
```

```
signal(S):
  S++
```

Midterm - moved from Oct 22nd to Oct 29th