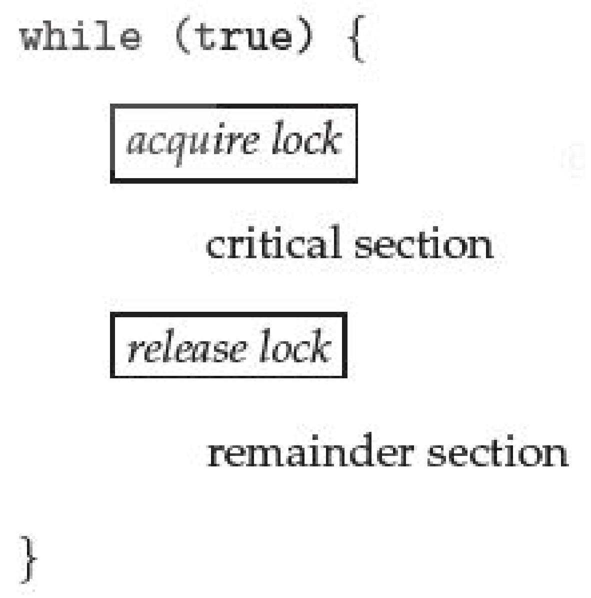
H/W solutions

Locks

- Critical sections must be protected with locks to guarantee the property of mutual exclusion.
- Using a simple lock variable and manipulating it will not work, because the race condition will now occur when updating the lock.
- Better alternative is to use atomic instructions
- Threads/Processes that want to access a critical section must try to acquire the lock, and proceed to the critical section only when the lock has been acquired.

Synchronization via Locks



```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

- Lock(x)

```
              var x: shared integer;
      Lock(x) : begin
             var y: integer;
                      y = x;
             while y =1 do y = x; // wait until gate is open
             x =1;
             end;
```

- Unlock(x)

```
             Unlock(x)
             x = 0;
```

If requested lock is held by other threads/processes

Two Options:

1. the thread/process could wait busily, constantly polling to check if the lock is available.
2. the thread/process could be made to give up its CPU, go to sleep (i.e., block), and be scheduled again when the lock is available.

The former way of locking is usually referred to as a spinlock, while the latter is called a regular lock or a mutex.

HARDWARE SOLUTIONS

- To provide a generalized solution to the critical section problem, some sort of lock must be set to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting the critical section.
- The hardware required to support critical sections must have, indivisible instructions or atomic operations.
- These operations are guaranteed to operate as a single instruction without interruption .

Synchronization Hardware

Test and modify the content of a word atomically

```
var x: shared integer;
   Test-and-set (x) :
         begin
                  var y: integer;
                  y = x;
                  if y = 0 then x = 1
         end;
```

Usage:

Lock

```
    var x: shared integer
lock(x):
    begin
        var y: integer;
        repeat y = test-and-set (x)
        until y = 0;
    end;
```

Unlock

```
    x = 0;
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Process P_i

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
}
```

Synchronization Hardware

- Atomically swap two variables.

```
Swap (var x,y: boolean);
        var temp: boolean;
         begin
               temp = x;
               x =y;
               y = temp;
          end
```

Usage

- Lock

```
p = true;
repeat swap (S,P) until p = false;
```

- Unlock

```
s = false
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

```
boolean lock;
```

- Process P_i

```
do {
    key = true;
    while (key == true)
                Swap(lock, key);
        critical section
    lock = false;
        remainder section
}
```

Classical Problems of Synchronization

Producer Consumer Bounded Buffer Problem

Two classes of processes

- Producers, which produce times and insert them into a buffer.
- Consumers, which remove items and consume them.

Issues

- Overflow Condition: if the producer encounters a full buffer? Block it.
- Underflow Condition : if the consumer encounters an empty buffer? Block it.

Reader/Writers Problem

Two classes of processes.
Readers, which can work concurrently.
Writers, which need exclusive access.

Rules

1. Must prevent 2 writers from being concurrent.
2. Must prevent a reader and a writer from being concurrent.
3. Must permit readers to be concurrent when no writer is active.
4. Perhaps want fairness (i.e., freedom from starvation).

Variants

Writer-priority readers/writers.
Reader-priority readers/writers.
Reader Writer Alternates

Dining Philosophers Problem

A classical problem. Some number of philosophers spend time thinking and eating. Being poor, they must share a total of five chopsticks.

Rules:

One chopstick between each philosopher

Philosopher first picks up one (if it is available) then the second (if available)

Only puts down chopsticks after eating for a while

Deadlock can occur, e.g., if all philosophers simultaneously pick up chopstick to his/her left.

Solutions:

Allow only n-1 (if we have n chopsticks) at table

Philosopher only picks up one chopstick if he/she can pick up both.

Odd philosopher first picks up left, even philosopher first picks up right.