Synchronization (Cont.)

- 3 requirements for critical section: mutual exclusion, forward process, bounded waiting
- Classical implementation is a lock: lock(L) → use/update data → unlock(L)

▶ How would you implement a lock? Main problem is how to check & set atomically

? ~~[illegible]~~ Example w/ 2 threads w/ ids 0 and 1:

```
lock(x) {
    while (turn ≠ tid);
    x = 1;
}

unlock(x) {
    x = 0;
    turn = 1 - tid;
}
```

← NOTE: x is really not necessary here

- This provides **mutual exclusion** – turn is either 0 or 1, so only one thread gets in.

NOTE: we assume lock & unlock are used correctly (no unlock if not holding lock).

- But there is no ~~bounded waiting~~ **forward progress** – say turn = 0 initially, thread 0 gets it, releases it, now turn = 1 and thread 0 can't get the lock again ever, unless thread 1 gets it and releases it.

- **BETTER LOCKING** (THIS ACTUALLY WORKS):

```
flag[0] = flag[1] = 0;
turn = 0;
```

↑ shared state

← for 2 threads

```
lock(x) {
    int other = 1 - tid;
    flag[tid] = 1;
    turn = other;
    while (flag[other]
        && turn == other);
}

unlock(x) {
    flag[tid] = 0;
}
```

▶ **Let's make this easier** (assume **1 CPU**)...

```
Lock(x);    ⟶ disable interrupts
CS ...      ⟵ no interrupts (also syscalls) here
unlock(x);  ⟶ enable interrupts
```

WORKS BECAUSE: this provides mutual exclusion, forward progress, and bounded waiting

▶ With **Few** CPUs this will not work ~~[illegible]~~

- Disable interrupts just says "do not interrupt me on this CPU" – it does not impact stuff that is running on other CPUs.
- Solution is to support special instructions in the computer architecture:

I.e. hardware

Example: **TEST_AND_SET** instruction:

→ If test is not true then set to be true + return old value

ATOMICALLY ACROSS ALL CPUS!!

In hardware, this is usually done by locking the bus, so only 1 CPU can access the data for the test, then performing the check/set (not exactly cheap!).

COMSW4118-9-2 NOTE: we use busy waiting / spinning 10/3/2017

- Now, we can implement as: where TEST_AND_SET(x) sets x to 1 if it is 0 and returns its old value.

SPINNING →
```
lock(x) {
    while (TESTANDSET(x));
}
```

```
unlock(x) {
    x = 0;   ← assume assignment is atomic
}
```

- COMPARE_AND_SWAP() in textbook is similar in nature and also requires hardware support.
- We use busy wait / spinning to wait for the lock above. If we have 4 cores and one thread has the lock on one and three threads are waiting for it on the other 3 cores (spinning in the while) we waste a lot of cycles.
- With multiple locks, you may have a deadlock (Ex: ABBA deadlock) or the dining philosophers problem

▶ DeadLock conditions:

- Mutual exclusion - when one holds the lock, others can't get it
- Hold and wait - one has a lock and is waiting for another
- No preemption - one can't be forced to release lock it has
- Circular wait - like $A \rightarrow B \rightarrow C$

● Rules to **avoid deadlocks:**

- Do not grab multiple locks
- If you must, grab them in the same order

Back to disabling interrupts - disabling them for the duration of the whole critical section is overkill and not necessary...

- say we disable them only for the duration of acquiring the lock... AND USE A SEMAPHORE?

▶ Semaphore:

P(S): ← down
```
while (S<=0);
S--;
```

V(S): ← UP
```
S++;
```

Then we can implement lock/unlock as:

```
Lock(x) {
    disable interrupts;
    while (S<=0);
    S--;
    enable interrupts
}
```

Now disable/enable is only when getting/releasing lock

```
Unlock(x) {
    disable interrupts;
    S++;
    enable interrupts;
}
```

BUT: This does not work well in practice: in a single processor, a thread can get a lock, then another thread would block forever waiting for the lock w/ interrupts disabled!