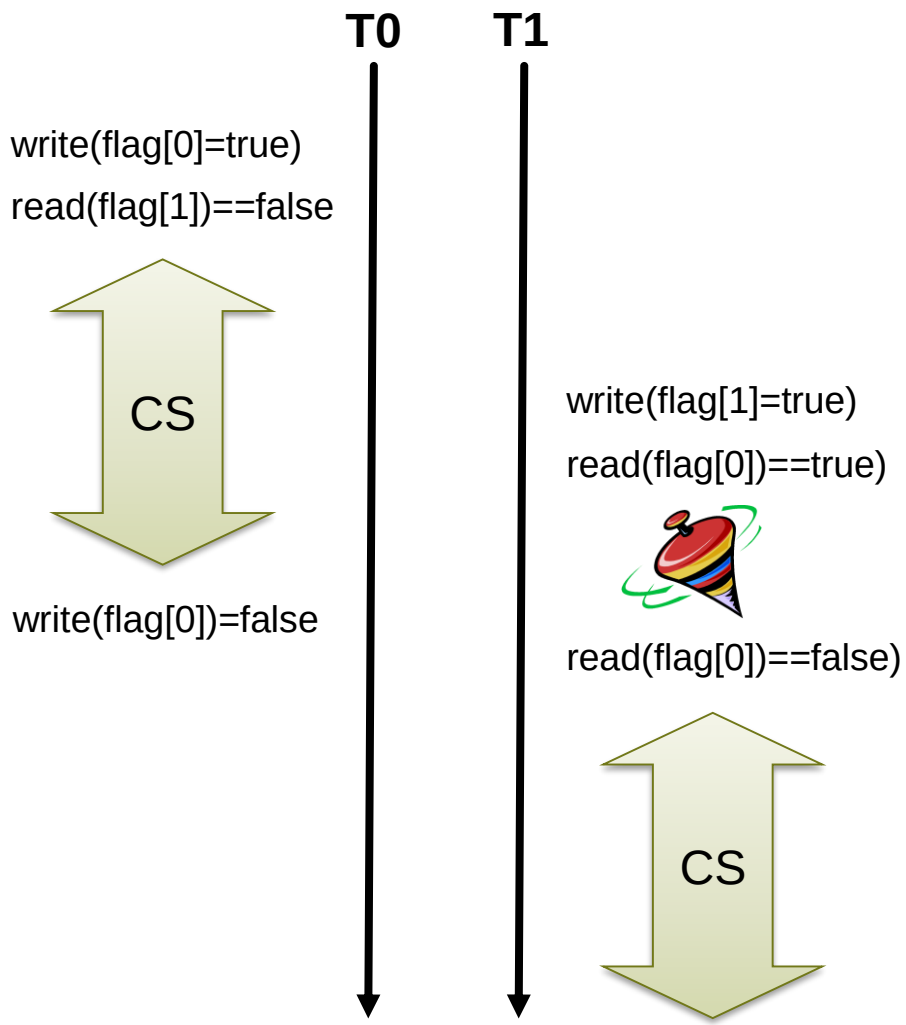
SALVATORE DI GIROLAMO <DIGIROLS@INF.ETHZ.CH>

DPHPC: Locks

Recitation session

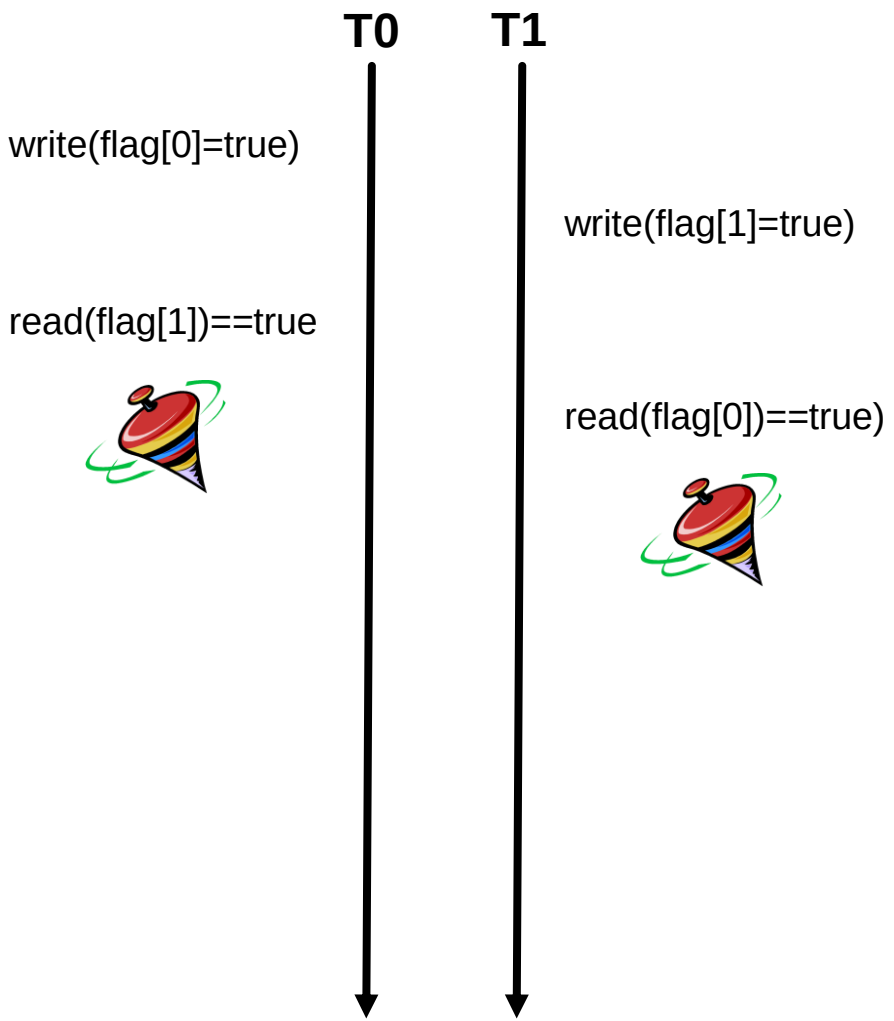
2-threads: LockOne

```
volatile int flag[2];

void lock() {
  int j = 1 - tid;
  flag[tid] = true;
  while (flag[j]) {} // wait
}

void unlock() {
  flag[tid] = false;
}
```



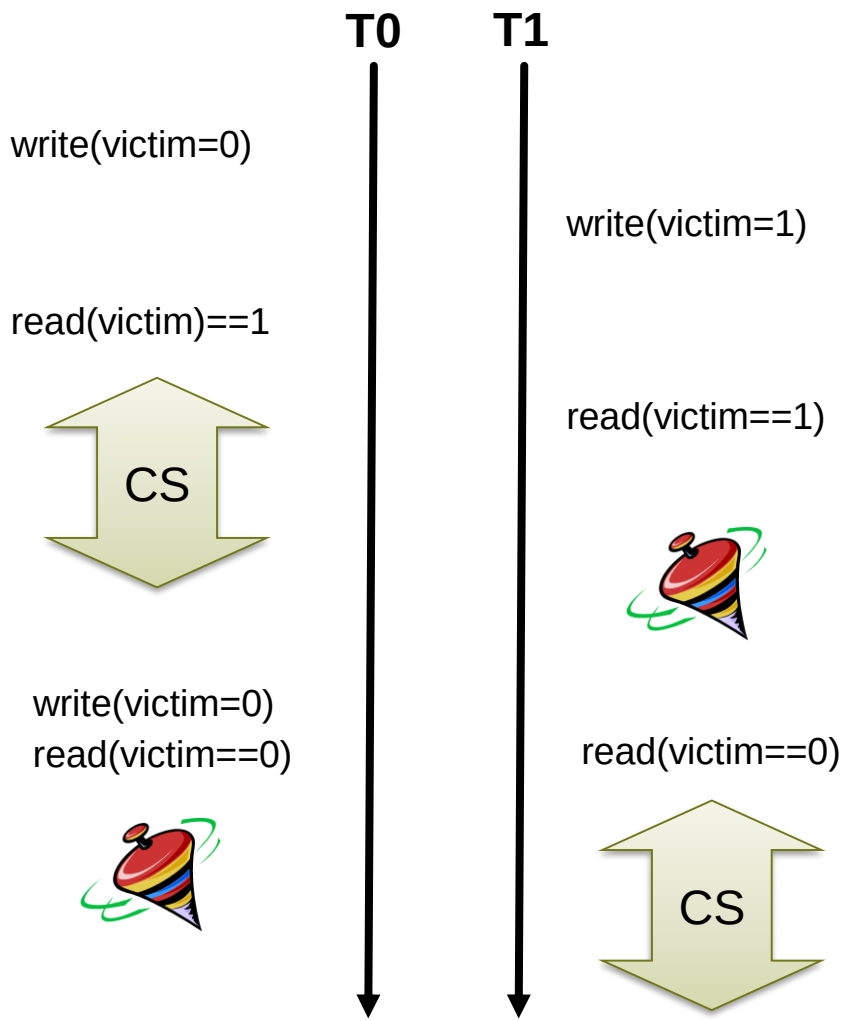
2-threads: LockOne

```
volatile int flag[2];

void lock() {
  int j = 1 - tid;
  flag[tid] = true;
  while (flag[j]) {} // wait
}

void unlock() {
  flag[tid] = false;
}
```



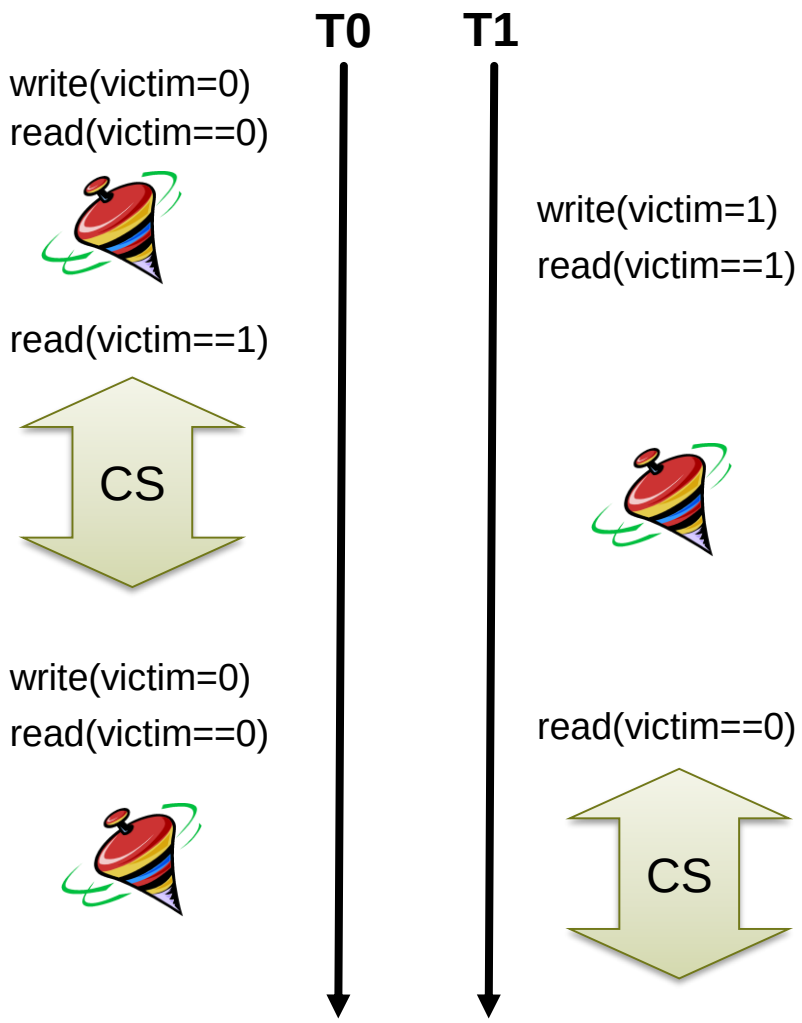
2-threads: LockTwo

```
volatile int victim;

void lock() {
  victim = tid; // grant access
  while (victim == tid) {} // wait
}

void unlock() {}
```



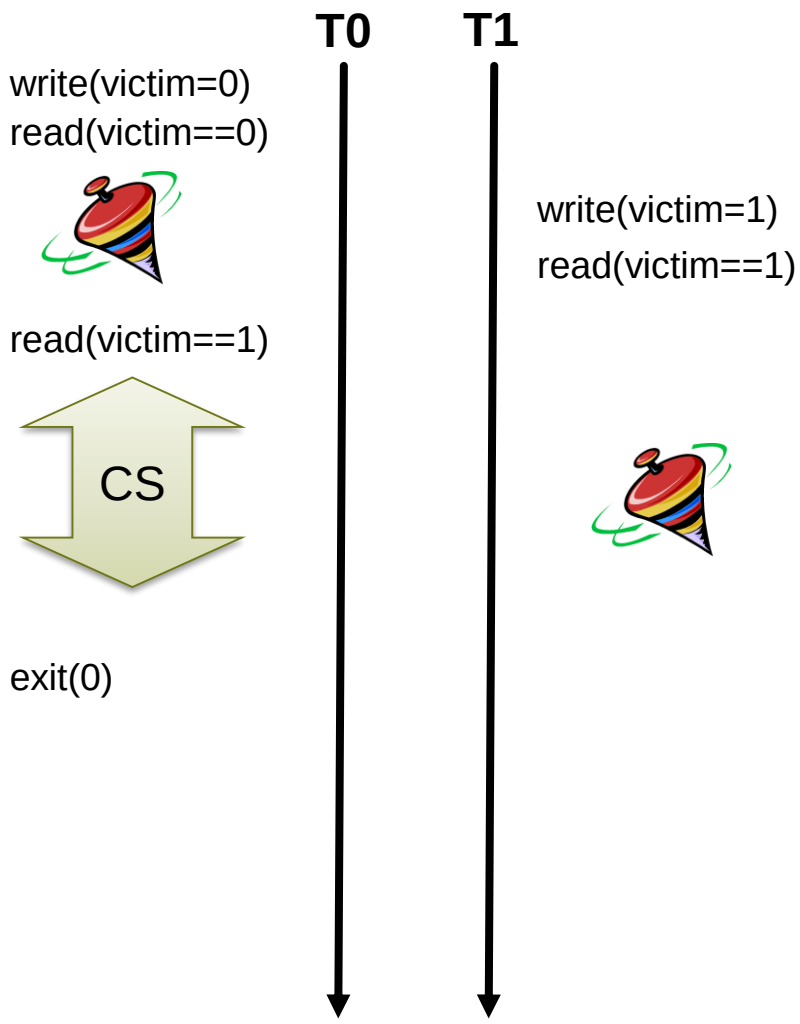
2-threads: LockTwo

```
volatile int victim;

void lock() {
  victim = tid; // grant access
  while (victim == tid) {} // wait
}

void unlock() {}
```



2-threads: LockTwo

```
volatile int victim;

void lock() {
  victim = tid; // grant access
  while (victim == tid) {} // wait
}

void unlock() {}
```

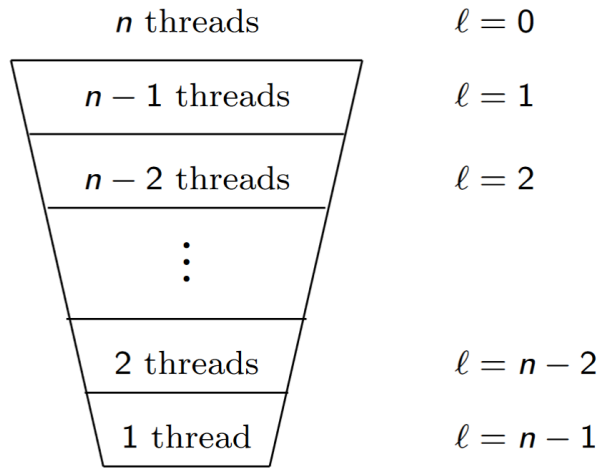


2-threads: Peterson lock

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1;      // I'm interested
  victim = tid;       // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0;  // I'm not interested
}
```

N-threads: Filter Lock

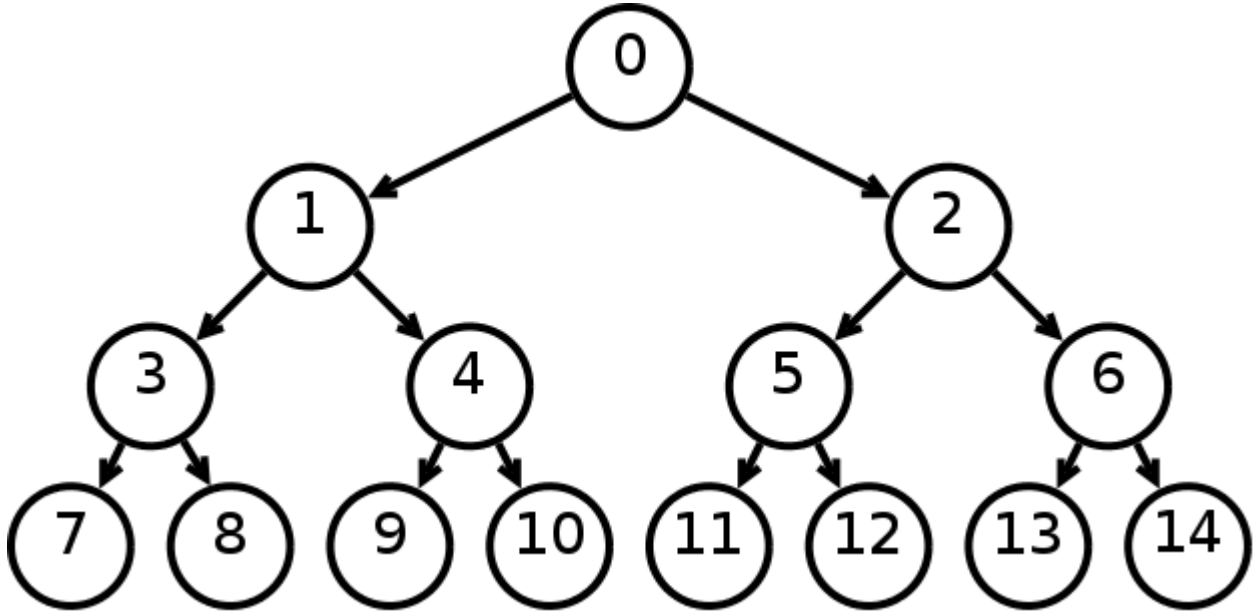
```
volatile int level[n] = {0,0,…,0}; // highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
  for (int i = 1; i < n; i++) { // attempt level i
    level[tid] = i;
    victim[i] = tid;
    // spin while conflicts exist
    while ((∃k != tid) (level[k] >= i && victim[i] == tid )) {};
  }
}

void unlock() {
  level[tid] = 0;
}
```



- **At least one thread trying to enter level L succeeds**
- **If more than one thread is trying to enter level L, then at least one is blocked (waits at that level)**

N-threads: Peterson locks in a binary tree

- **Another way to generalize the Peterson lock to n>=2 threads is to use a binary tree, where each node holds a Peterson lock for two threads.**
  - Threads start at a leaf in the tree, and move one level up when they acquire the lock at a node.
  - A thread that holds the lock of the root can enter its critical section.
  - When a thread exits its critical section, it releases the locks of nodes that it acquired.



N-threads: Bakery lock

```
volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};

void lock() {                                          Doorway
  flag[tid] = 1; // request
  label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
  while ((∃k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}
public void unlock() {
  flag[tid] = 0;
}c
```

- **What happens if two threads execute their doorways concurrently?**
  - **Lexicographical order helps us!**
- **A thread could see a set of labels that never existested in memory at the same time**

Flaky Lock

Programmers at the Flaky Computer Corporation designed the protocol shown below to achieve n-thread mutual exclusion. **Does this protocol satisfy mutual exclusion? Is it starvation-free? Is it deadlock-free?**

```
1 class Flaky implements Lock {
2     private int turn ;
3     private boolean busy = false ;
4     public void lock () {
5         int me = ThreadID .get ();
6         do {
7             do {
8                 turn = me;
9             } while ( busy );
10            busy = true ;
11        } while ( turn != me);
12    }
13    public void unlock () {
14        busy = false ;
15    }
16 }
```

OpenMP - Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- **High level synchronization:**
  - Critical, Atomic, Barrier, Ordered
- **Low level synchronization**
  - Flush, Locks (both simple and nested)

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Barrier

- **Each thread waits until all threads arrive.**

```
#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier

        B[id] = big_calc2(id, A);
}
```

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Critical

- **Mutual exclusion: Only one thread at a time can enter a critical region**

```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     for(i=id;i<niters;i+=nthrds){
          B =  big_job(i);
#pragma omp critical
          res += consume (B);
     }
}
```

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Atomic

- **Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)**

```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
        X +=  tmp;
}
```

The statement inside the atomic must be one of the following forms:

- x binop= expr
- x++
- ++x
- x—
- --x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Locks in OpenMP

- **Simple Locks**
  - A simple lock is available if not set
  - Routines:

    omp_init_lock/omp_destroy_lock: create/destroy the lock

    omp_set_lock: acquire the lock

    omp_test_lock: test if the lock is available and set it if yes. Doesn't block if already set.

    omp_unset_lock: release the lock

- **Nested Locks**
  - A nested lock is available if it is not set OR is set and the calling thread is equal to the current owner
  - Same functions of the simple lock: omp_*_nest_lock

Note: a lock implies a memory fence of all the thread variables

http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

Locks in OpenMP: An example

```
class data{
 private:
   std::set<int> flags;
 #ifdef _OPENMP
   omp_lock_t lock;
 #endif
 public:
   data() : flags(){
 #ifdef _OPENMP
     omp_init_lock(&lock);
 #endif
   }
   ~data(){
 #ifdef _OPENMP
     omp_destroy_lock(&lock);
 #endif
   }

   bool set_get(int c){
   #ifdef _OPENMP
     omp_set_lock(&lock);
   #endif
     bool found = flags.find(c) != flags.end();
     if(!found) flags.insert(c);
   #ifdef _OPENMP
     omp_unset_lock(&lock);
   #endif
     return found;
   }
 };
```

http://bisqwit.iki.fi/story/howto/openmp/#Locks

Locks in OpenMP: An example

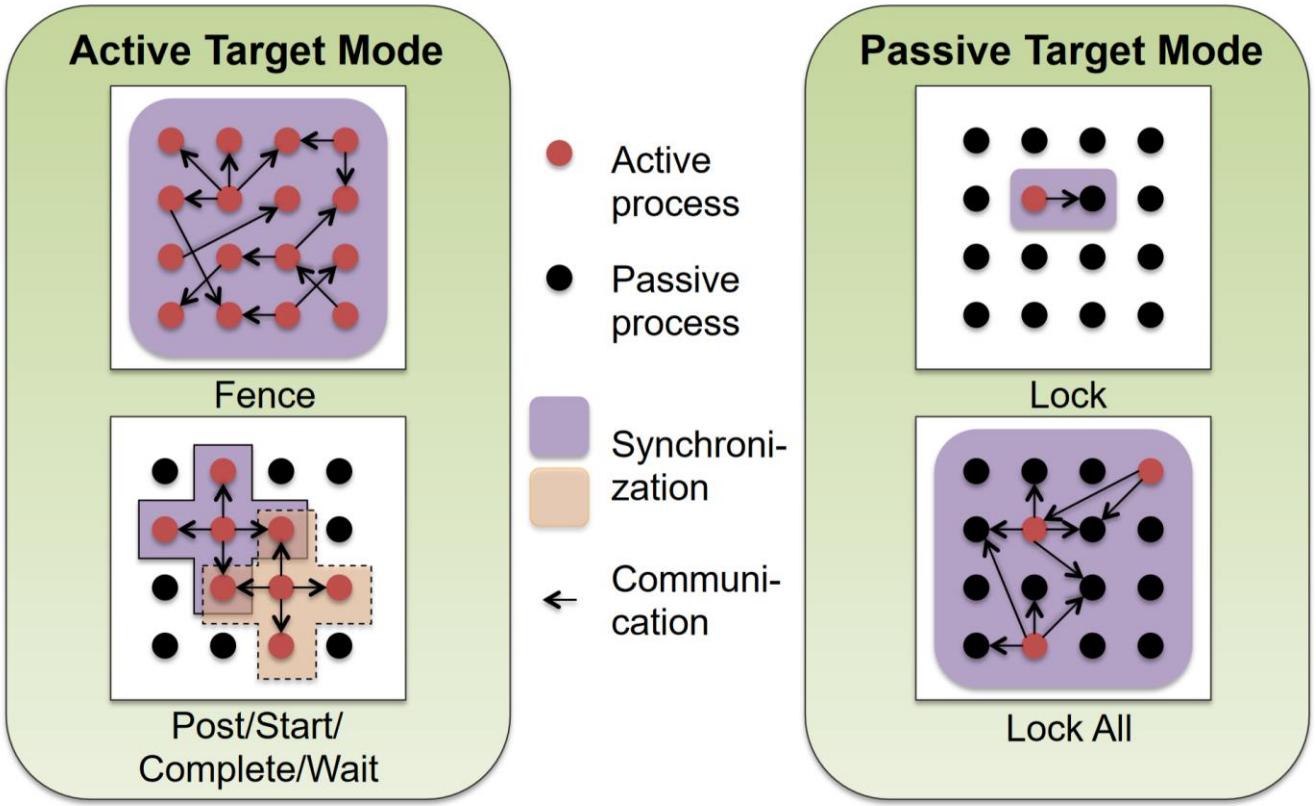
```
#ifdef _OPENMP
struct MutexType{
  MutexType() { omp_init_lock(&lock); }
  ~MutexType() { omp_destroy_lock(&lock); }
  void Lock() { omp_set_lock(&lock); }
  void Unlock() { omp_unset_lock(&lock); }
  MutexType(const MutexType& ) { omp_init_lock(&lock); }
  MutexType& operator= (const MutexType& ) { return *this; }
public:
  omp_lock_t lock;
};
#else
/* A dummy mutex that doesn't actually exclude anything, but as the
struct MutexType{
  void Lock() {}
  void Unlock() {}
};
#endif

/* An exception-safe scoped lock-keeper. */
struct ScopedLock{
  explicit ScopedLock(MutexType& m) : mut(m), locked(true) { mut.Lock(); }
  ~ScopedLock() { Unlock(); }
  void Unlock() { if(!locked) return; locked=false; mut.Unlock(); }
  void LockAgain() { if(locked) return; mut.Lock(); locked=true; }
private:
  MutexType& mut;
  bool locked;
private: // prevent copying the scoped lock.
  void operator=(const ScopedLock&);
  ScopedLock(const ScopedLock&);
};v
```

```
#include <set>

class data
{
private:
  std::set<int> flags;
  MutexType lock;
public:
  bool set_get(int c)
  {
    ScopedLock lck(lock); // locks the mutex

    if(flags.find(c) != flags.end()) return true; // was found
    flags.insert(c);
    return false; // was not found
  } // automatically releases the lock when lck goes out of scope.
};
```

http://bisqwit.iki.fi/story/howto/openmp/#Locks

What about MPI?

- **Active target requires cooperation by all processes in the group of the window object**
  - MPI_Win_fence, MPI_Win_{post/start/complete/wait/test}
  - Good for many but not all RMA applications
- **What if each process may need to independently access data?**
  - Use Passive target synchronization



Passive Target Synchronization

- `MPI_Win_lock(int locktype, int rank, int assert, MPI_Win win)`
- `MPI_Win_unlock(int rank, MPI_Win win)`

- `MPI_Win_flush/flush_local(int rank, MPI_Win win)`

ATTENTION: a LOCK is NOT a LOCK

- **Lock type**
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

- **Flush**: Remotely complete RMA operations to the target process
  - After completion, data can be read by target process or a different process

- **Flush_local**: Locally complete RMA operations to the target process

http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture35.pdf

Lock is not Lock

- **The name "Lock" is unfortunate**
  - Lock is really "begin epoch"
  - Unlock is really "end epoch"

- **An MPI "Lock" does not establish a critical section or mutual exclusion**
  - With "MPI_LOCK_EXCLUSIVE" the RMA operations have exclusive access to the data they access/update during the time that they access the remote window
  - Different RMA operations can interleave (but they're atomically executed at the target)
  - This is very different than a "lock" in the sense of a thread lock

http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture35.pdf