Synchronization between processors

- Required on all levels of multi-threaded programming
  - Lock/unlock
  - Mutual exclusion
  - Barrier synchronization

- Key hardware capability:
  - Uninterruptable instruction capable of automatically retrieving or changing a value
Thread synchronization

- Reading and writing a shared variable between two threads
- Timing between two threads will differ in every iteration
- If you need a specific value for thread B of the variable you need to synchronize access to the shared variable

Mutex locks in POSIX threads

- Mutex: Mutual exclusion
  - a lock is used before accessing a shared resource and released after the access
  - mutex lock represented by a mutex variable
  - while mutex lock is set, other threads that try to access the lock will be denied
  - if more than one thread wait for the lock, all of them will be made runnable, but only one thread will get the lock
- All threads have to use mutex locks for accessing the variable, else no guarantee on correctness
Mutex locks (II)

- **mutex**: mutex variable to be initialized/destroyed counter part
  - once initialized, a mutex variable can be used for an unlimited number of lock/unlock operations
- **attr**: attributes for the mutex

```c
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Mutex locks (III)

- **pthread_mutex_lock**: acquire lock for the mutex.
  - If mutex is already blocked by another thread, wait until the mutex is unlocked
- **pthread_mutex_trylock**: acquire lock for the mutex.
  - If mutex is already blocked by another thread, do not wait but return EBUSY to indicated failure

```c
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```
Thread synchronization revisited

- Example: Force thread B to read value of shared variable after write₂

Simple Example (IIIa)

```c
#include <pthread.h>
int value=0;              // shared variable
pthread_mutex_t mymutex;  // mutex variable

int main ( int argc, char **argv )
{
    int threadid, ret;

    pthread_mutex_init (&mymutex,NULL); // Initialize mutex
    pthread_mutex_lock (&mymutex);      // Acquire mutex lock

    // main thread spawns another thread
    ret = pthread_create (&threadid, NULL, tfunc, NULL);
    if ( ret != 0 ) printf("Error creating a thread\n");
    value = 1;                          // write 1
    value ++;                           // write 2
    pthread_mutex_unlock (&mymutex);    // Release lock

    pthread_join( threadid, &val);     // wait for other thread
    pthread_mutex_destroy (&mymutex);   // destroy mutex

    return (0);
}
```
Simple Example (IIIb)

```c
void *tfunc (void *arg){
    int localvalue;
    pthread_mutex_lock (&mymutex); // wait for lock
    localvalue = value;           // read shared variable
    pthread_mutex_unlock (&mymutex);
    do_work(localvalue);
    pthread_exit ((void *) 1);
    return NULL;
}
```

Mutex locks (IV)

- A thread will deadlock itself if it tries to lock the same mutex twice
- If more than one mutex is used a deadlock can occur if one thread holds lock1 and waits for lock2 and the other thread holds lock2 and waits for lock1
  - Order for accessing mutexes has to be identical in all code paths
  - e.g. need to hold lock1 in order to be allows to hold lock2
Synchronization

- Lock/unlock operations on the hardware level, e.g.
  - Lock returning 1 if lock is free/available
  - Lock returning 0 if lock is unavailable
- Implementation using *atomic exchange*
  - Process sets the value of a register/memory location to the required operation
  - Setting the value must not be interrupted in order to avoid race conditions
  - Access by multiple processes/threads will be resolved by write serialization

Synchronization (II)

- Other synchronization primitives:
  - Test-and-set
  - Fetch-and-increment
- Problems with all three algorithms:
  - Require a read and write operation in a single, uninterruptable sequence
  - Hardware can not allow any operations between the read and the write operation
  - Complicates cache coherence
  - Must not deadlock
Load linked/store conditional

- Pair of instructions where the second instruction returns a value indicating, whether the pair of instructions was executed as if the instructions were atomic
- Special pair of load and store operations
  - Load linked (LL)
  - Store conditional (SC): returns 1 if successful, 0 otherwise
- Store conditional returns an error if
  - Contents of memory location specified by LL changed before calling SC
  - Processor executes a context switch

Load linked/store conditional (II)

- Assembler code sequence to atomically exchange the contents of register R4 and the memory location specified by R1

```assembly
try:
  MOV R3, R4
  LL R2, 0(R1)
  SC R3, 0(R1)
  BEQZ R3, try
  MOV R4, R2
```
Load linked/store conditional (III)

- Implementing fetch-and-increment using load linked and conditional store

```assembly
try:  LL    R2, 0(R1)
     DADDUI R3, R2, #1
     SC     R3, 0(R1)
     BEQZ   R3, try
```

- Implementation of LL/SC by using a special Link Register, which contains the address of the operation

Spin locks

- A lock that a processor continuously tries to acquire, spinning around in a loop until it succeeds.
- Trivial implementation

```assembly
lockit:  DADDUI R2, R0, #1
         EXCH R2, 0(R1) !atomic exchange
         BNEZ R2, lockit
```

- Since the EXCH operation includes a read and a modify operation
  - Value will be loaded into the cache
    - Good if only one processor tries to access the lock
    - Bad if multiple processors in an SMP try to get the lock (cache coherence)
  - EXCH includes a write attempt, which will lead to a write-miss for SMPs
Spin locks (II)

- For cache coherent SMPs, slight modification of the loop required

lockit:  
  LD R2, 0(R1) !load the lock  
  BNEZ R2, lockit !lock available?  
  DADDUI R2, R0, #1 !load locked value  
  EXCH R2, 0(R1) !atomic exchange  
  BNEZ R2, lockit !EXCH successful?

Spin locks (III)

- ...or using LL/SC

lockit:  
  LL R2, 0(R1) !load the lock  
  BNEZ R2, lockit !lock available?  
  DADDUI R2, R0, #1 !load locked value  
  SC R2, 0(R1) !atomic exchange  
  BNEZ R2, lockit !SC successful?