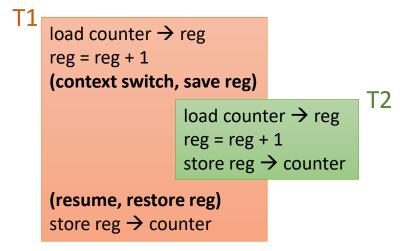
Locking

Mythili Vutukuru CSE, IIT Bombay

Recap: Shared data access in threads

- The C code "counter = counter + 1" is compiled into multiple instructions
 - Load counter variable from memory into register
 - Increment register
 - Store register back into memory of counter variable
- What happens when two threads run this line of code concurrently?
 - Counter is 0 initially
 - T1 loads counter into register, increment reg
 - Context switch, register (value 1) saved
 - T2 runs, loads counter 0 from memory
 - T2 increments register, stores to memory
 - T1 resumes, stores register value to counter
 - Counter value rewritten to 1 again
 - Final counter value is 1, expected value is 2



load counter \rightarrow reg reg = reg + 1 store reg \rightarrow counter

Recap: Race conditions, critical sections

- Incorrect execution of code due to concurrency is called race condition
 - Due to unfortunate timing of context switches, atomicity of data update violated
- Race conditions happen when we have concurrent execution on shared data
 - Threads sharing common data in memory image of user processes
 - Processes in kernel mode sharing OS data structures
- We require mutual exclusion on some parts of user or OS code
 - Concurrent execution by multiple threads/processes should not be permitted
- Parts of program that need to be executed with mutual exclusion for correct operation are called critical sections
 - Present in multi-threaded programs, OS code
- How to access critical sections with mutual exclusion? Using locks

Using locks

- Locks are special variables that provide mutual exclusion
 - Provided by threading libraries
 - Can call lock/acquire and unlock/release functions on a lock
- When a thread T1 acquires a lock, another thread T2 cannot acquire same lock
 - Execution of T2 stops at the lock statement
 - T2 can proceed only after T1 releases the lock
- Acquire lock → critical section → release lock ensures mutual exclusion in critical section

```
int counter;
pthread_mutex_t m;
```

void start_fn() {

for(int i=0; i < 1000; i++) {
 pthread_mutex_lock(&m)
 counter = counter + 1
 pthread_mutex_unlock(&m)</pre>

```
main() {
counter = 0
```

pthread_t t1, t2
pthread_create(&t1,.., start_fn, ..)
pthread_create(&t2, .., start_fn,..)

```
pthread_join(t1, ..)
pthread_join(t2, ..)
```

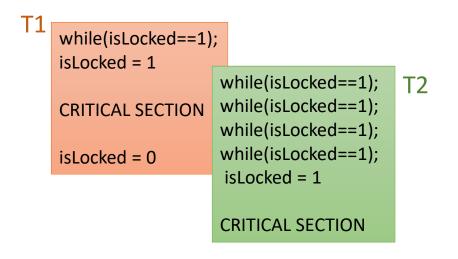
print counter

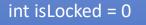
How to implement a lock?

- Goals of a lock implementation
 - Mutual exclusion (obviously!)
 - Fairness: all threads should eventually get the lock, and no thread should starve
 - Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources
- Implementation of locks are needed for both userspace programs (e.g., pthreads library) and kernel code
 - Separate implementations in user libraries and OS

Incorrect lock implementation

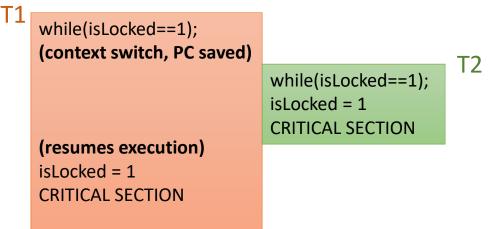
- Example of incorrect lock implementation
 - Use variable isLocked to indicate lock status (0 means lock is free, 1 indicates it is acquired)
 - To acquire lock, a thread waits as long as lock is busy, and then sets it to 1 (acquired)
 - One interleaving of executions (left) works while another (right) may not work





```
void acquire_lock() {
  while(isLocked == 1); //wait
  isLocked = 1
```

```
void release_lock() {
    isLocked = 0
```



Hardware atomic instructions

- Need a way to check a variable and set its value atomically
 - No context switch between checking lock variable and setting it
 - But user programs have no control over context switches
- Solution: use hardware atomic instructions
- Example: test-and-set hardware atomic instruction
 - Two arguments: address of variable and new value to set
 - Writes new value into a variable and returns old value in one single step
 - Entire logic implemented in hardware, runs in one single step

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store 'new' into old_ptr
    return old; // return the old value
}
```

Lock implementation using test-and-set

- Simple lock can be implemented using test-and-set instruction
 - isLocked variable indicates lock status (0=free, 1=acquired)
 - If test-and-set(&isLocked, 1) returns 1, it means lock is not free, wait
 - If test-and-set(&isLocked, 1) returns 0, lock was free and was acquired, done!
- No further race conditions possible with this lock implementation
 - All modern lock implementations based on such hardware instructions
 - Software based locking algorithms do not work well in modern systems

int isLocked = 0

```
void acquire_lock() {
    while(test-and-set(&isLocked, 1) == 1); //wait
    //return, lock is acquired
}
```

```
typedef struct __lock_t {
1
        int flag;
2
    } lock_t;
3
4
    void init(lock_t *lock) {
5
        // 0 indicates that lock is available, 1 that it is held
6
        lock -> flag = 0;
7
    }
8
9
    void lock(lock_t *lock) {
10
        while (TestAndSet(&lock->flag, 1) == 1)
11
             ; // spin-wait (do nothing)
12
13
    }
14
    void unlock(lock_t *lock) {
15
        lock -> flag = 0;
16
    }
17
              Figure 28.3: A Simple Spin Lock Using Test-and-set
```

Image credit: OSTEP

Another instruction: compare-and-swap

- Another example: compare-and-swap (CAS) hardware atomic instruction
 - Three arguments: address of variable, expected old value, new value
 - If variable has expected old value, then write new value and return true; else do not change variable and return false

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
    }
    Figure 28.4: Compare-and-swap
```

Image credit: OSTEP

Lock using CAS

- Lock implementation using compare-and-swap
 - If compare-and-swap(&isLocked, 0, 1) returns false, it means lock is busy, wait
 - If compare-and-swap(&isLocked, 0, 1) returns true, it means old value of lock was 0 and was changed to 1, so lock has been acquired, done!

```
int isLocked = 0
void acquire_lock() {
    while(compare-and-swap(&isLocked, 0, 1) == false); //wait
}
```

Evaluating spinlock implementations

- Correctness: does it lead to mutual exclusion correctly?
- Fairness: are all waiting threads treated fairly? Can we guarantee that every waiting thread will get its turn?
 - The implementations we saw here do not guarantee it
- Performance: overheads of having threads spin for lock
 - Single core system: what happens when thread holding lock is context switched out and other threads that are scheduled continue to spin for lock?
 - Problem less severe in multicore system. Why? (Thread holding lock can finish while other threads are spinning)

Spinlock vs. sleeping mutex

- Simple lock implementation seen here is a spinlock
 - If thread T1 has acquired lock, and thread T2 also wants lock, then T2 will keep spinning in a while loop till lock is free
- Another implementation option: thread can go to sleep (be blocked) while waiting for lock, saving CPU cycles
 - OS blocks waiting thread, context switch to another thread/process
 - Such locks are called (sleeping) mutex
- Threading libraries provide APIs for both spinlocks and sleeping mutex
 - Better to use spinlock if locks are expected to be held for short time, avoid context switch overhead
 - Better to use sleeping mutex if critical sections are long

Guidelines for using locks

- When writing multithreaded programs, careful locking discipline
 - Protect each shared data structure with one lock
 - Locks can be coarse-grained (one big fat lock) or fine-grained (many smaller locks)
 - Any thread wanting to access shared data must acquire corresponding lock before access, release lock after access
- If using third-party libraries in multi-threaded programs, check the documentation to see if if the library is thread-safe
 - Thread-safe implementations work correctly with concurrent access

Guidelines for using locks

- Good practice to acquire locks for both reading and writing data
 - Why locks for reading? We do not want to read incorrect data while another thread is concurrently updating the data
 - Some libraries provide separate locks for reading and writing, allowing multiple threads to concurrently read data if no other thread is writing
- Good practice to minimize use of locks, use only when needed
 - Why? Use of locks serializes thread access, removes gains due to parallelism
 - Example of minimizing lock usage: instead of each thread updating shared global counter, let each thread update a local counter, and periodically update global counter

```
typedef struct __counter_t {
1
        int
2
                          value;
        pthread_mutex_t lock;
3
    } counter_t;
4
5
    void init(counter_t *c) {
6
        c \rightarrow value = 0;
7
        Pthread_mutex_init(&c->lock, NULL);
8
9
   }
10
    void increment(counter_t *c) {
11
        Pthread_mutex_lock(&c->lock);
12
13
        c->value++;
        Pthread_mutex_unlock(&c->lock);
14
15
   }
16
    void decrement(counter_t *c) {
17
        Pthread mutex lock(&c->lock);
18
19
        c->value--;
        Pthread_mutex_unlock(&c->lock);
20
21
    }
22
    int get(counter_t *c) {
23
        Pthread_mutex_lock(&c->lock);
24
25
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
26
        return rc;
27
28
   }
                       Figure 29.2: A Counter With Locks
```

Image credit: OSTEP