CS310 Operating Systems

Lecture 21 : Locks using Hardware atomic Instructions: Test-and-Set, Compare-and-swap

Ravi Mittal

IIT Goa

Acknowledgements !

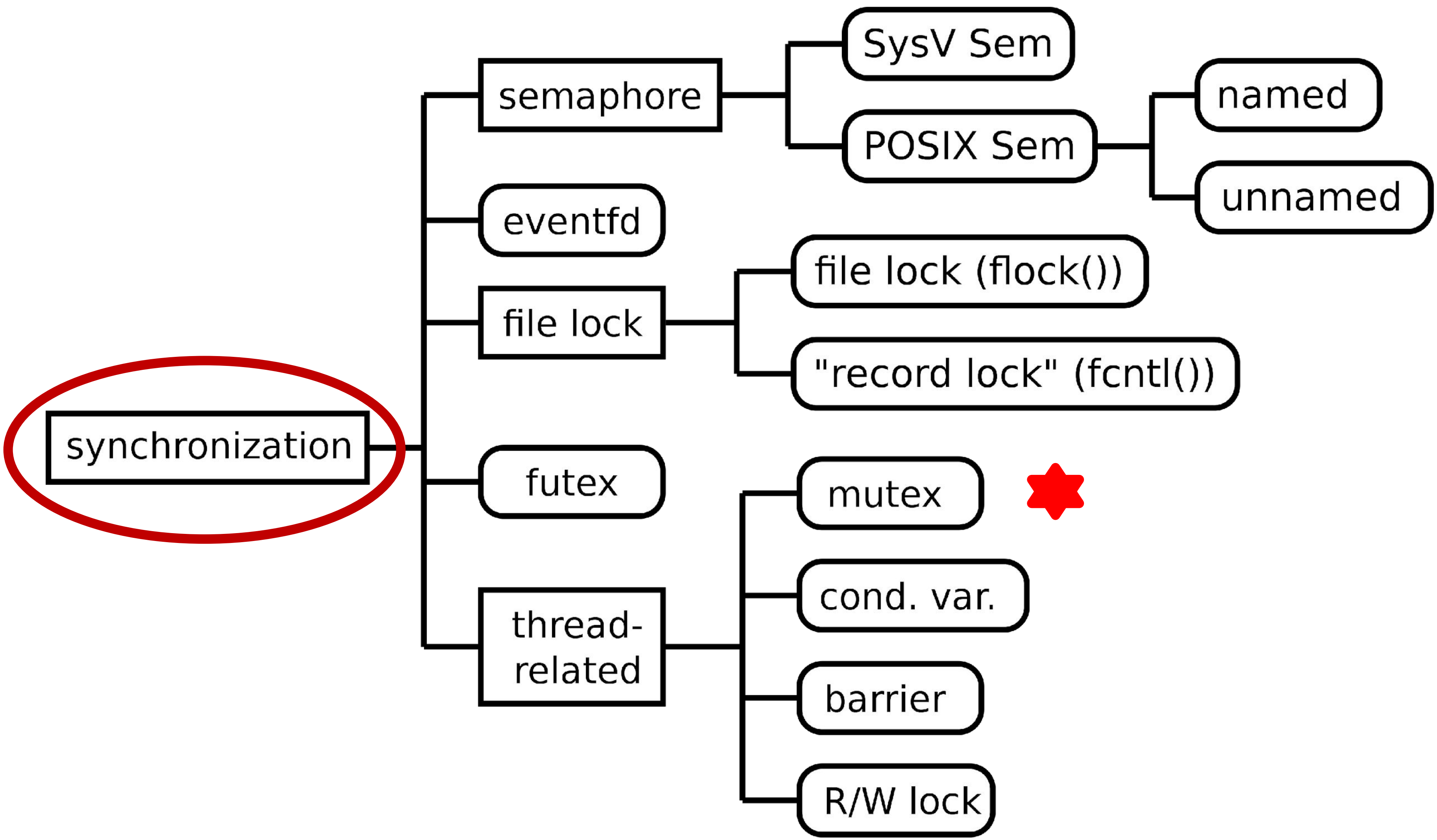
- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Book: Operating System: Three Easy Pieces, by Remzi H Arpaci-Dusseau, Andrea C Arpaci-Dusseau, Chapter 28 Locks
    - https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf

Reading

- Book: Operating System: Three Easy Pieces, by Remzi H Arpaci-Dusseau, Andrea C Arpaci-Dusseau
  - Chapter 31, https://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf

- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II, Chapter 5.8

Needs for Synchronization



We will start with High level primitives

Programs	Shared Programs
Higher-lev el API	Locks Semaphores Monitors Others
Hardware	Disable Ints Test&Set Compare&Swap, others

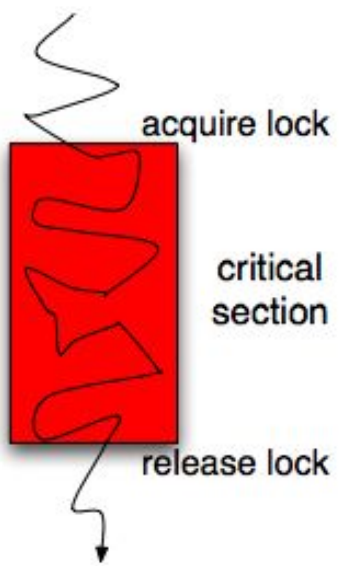
In the class we will study

- Implementations of locks
- Lock implementation by disabling interrupts
- Test-and-Set Atomic Instruction
- Compare-and-swap
- Sleeping Mutex
- Using Queues to improve on spinning

Last Class: Lock Introduction

What if we want to generalize beyond increments?

- Best mainstream solution: **Locks**
  - Implements **mutual exclusion**
    - You can't have it if I have it, I can't have it if you have i



when lock = 0, set lock = 1, continue

lock = 0

Recall: Lock

- Consider an update of shared variable
  - `balance = balance + 1;`
- This statement forms critical section. We need to protect it with a special lock variable (mutex in the example below)

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

- Note that lock is a variable that holds the state of the lock at any instant in time
- All threads accessing a critical section share a lock
- Only one thread becomes successful in holding the lock – current owner of the lock
  - Thus the thread is in critical section

From book: Operating Systems: Three Easy Pieces

Recall: Lock

- `Lock()` and `unlock()` are routines
- The thread that acquires the lock enters the critical section
- Then, if another thread calls the lock on the same lock variable, it will not return
  - So the thread is prevented from entering critical section
  - Function `lock()` doesn't return means it doesn't come out of lock() function
- Note that the owner of lock calls unlock
- Thread entities are created by the programmer but scheduled by the OS
  - Locks give some control to the programmer

- Pthreads library in Linux provides such locks

pthread lock calls

- The name that the POSIX library uses for a lock is a **mutex**

- `pthread_mutex_init()` - get a mutex
- `pthread_mutex_lock()` - lock a mutex (acquire it), block until available
- `pthread_mutex_trylock()` - try to lock a mutex (acquire it), do not block if unavailable
- `pthread_mutex_unlock()` - unlock a mutex (release it)
- `pthread_mutex_destroy()` - destroy a mutex (remove it)

Lock Implementation questions

- How can we build an efficient lock?
- What hardware support is required?
- What OS support is required?

Building a lock

- Goals of lock implementation
  - Mutual Exclusion
  - Fairness
  - Performance
    - Low time overhead
    - Acquiring, Releasing and waiting for a lock should not consume too many resources
- Implementations of locks are needed for both user-space programs and kernel code
- Implementing locks needs support from hardware and OS

Lock implementation by disabling interrupts

- The earliest solution: mutual exclusion by disabling interrupts
- Solution was invented for a single processor system

```
void lock() {
      DisableInterrupts();
}
void unlock() {
      EnableInterrupts();
}
```

- Simple approach
- Is it good enough? No.
- Disabling interrupts is a privileged instruction (for turning interrupts on and off) and program can misuse it (e.g., run forever)
- This approach will not work on multiprocessor systems, since another thread on another core can enter critical section

From book: Operating Systems: Three Easy Pieces

Lock implementation by disabling interrupts

- Turning off interrupts for extended period of time can lead to other interrupts becoming lost - this is a serious problem to handle
- It is dangerous to give a user process a power to disable interrupt
  - What happens if a user process disable interrupts and goes into infinite loop?
- This approach is inefficient
- This approach of interrupt masking is used by kernel to guarantee atomicity when accessing its own data structures
  - No trust issue inside the OS

Does this lock implementation work?

```c
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

- First thread that enters the critical section calls `lock()`
  - Tests if flag is 1; as the flag is 0, it enters the critical section and setup flag = 1
    - Thread 1 now holds the flag
  - When it finishes it calls `unlock()`
- Another thread calls `lock()` when thread 1 is in critical section
- It spin-waits in the while loop until flag becomes 0

Does this lock implementation work? No

- The solution is incorrect. Consider the code interleaving

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

- Thread 1 spins and finds out lock flag is 0;
- Thread 1 is interrupted before it sets flag = 1
- Thread 2 also finds flag = 0 and sets flag = 1 and enters critical section
- Thread 2 is interrupted
- Thread 1 now sets flag =1 and enters critical section
- Also, Spin-waiting wastes processor’s time --> expensive

From book: Operating Systems: Three Easy Pieces

Atomic Read-Modify-Write Instructions

- Hardware instructions that allows us to test and set or compare and swap, operations atomically

- Test_and_Set
- Compare_and_Swap
- Load-Linked and Store-Conditional
- Fetch-And-Add

Will study

- We can build locks with these instruction

Test-and-Set: Hardware Atomic Instruction

- It’s hard to ensure atomicity only in software
- Modern architectures provide hardware atomic instructions
  - Test-and-Set
- It enables to test the old value (which is what is returned) while simultaneously setting the memory location to a new value
  - Function returns the old value

Note that operation (of reading old value (of lock variable) and setting the new value) is atomic

This is used to implement lock

Test-and-Set: Hardware Atomic Instruction

```
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
}
```

- It returns the old value pointed to by the `old_ ptr`, and simultaneously updates said value to new

From book: Operating Systems: Three Easy Pieces

Test-and-Set: Hardware Atomic Instruction

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- Case 1: Flag is 0 initially. Thread 1 calls `lock()`.
  - Calls `TestAndSet(flag, 1)`
  - Routine returns old value of flag i.e. 0 and then atomically sets flag to 1; thread enters critical section
  - When the thread is finished it sets the flag to 0
- Case 2: Thread 1 is holding the lock. The second thread calls `lock()` and then calls `TestAndSet()`. `TestAndSet` returns the old value of flag i.e. 1 and simultaneously setting it to 1
  - Thread 2 spin-locks

From book: Operating Systems: Three Easy Pieces

Evaluating Spin Locks

- Correctness – Yes
  - It provides mutual exclusion
- Fairness
  - Spin-locks don’t provide any fairness guarantees
  - An unlucky thread may spin forever under contention
    - Starvation
- Performance
  - Single CPU case
    - Poor performance. If the thread holding lock is preempted, all other threads that are competing for the lock will spin wait in their time slot

Spin lock with `compare-and-swap`

- Another hardware primitive – some systems provide it
- Idea: To test whether the value at the address specified by `ptr` is equal to expected
    - if so, update the memory location pointed to by `ptr` with the new value
    - If not, do nothing
    - In both cases, return the original value
        - Thus the code calling `compare-and-swap` come to know whether it succeeded or not
- Similar to `test-and-set` instruction

Spin lock with `compare-and-swap`

- `compare-and-swap` Instruction

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

- Spinlock using compare-and-swap

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

Simple Approach: Just Yield (sleeping mutex)

- What to do when a context switch occurs in a critical section?
  - Other threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again
- Instead of spinning for a lock, a contending thread simply gives up the CPU and check back later
  - – `yield()` moves thread from running to ready state

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

From book: Operating Systems: Three Easy Pieces

Spin lock and Sleeping Mutex

- If there are many threads (say 100) they contend for lock repeatedly
- If a thread holding lock is preempted, other 99 threads will each call `lock()` and find it held
  - Sleeping Mutex: These threads will yield as soon it finds that lock is held
  - Spin lock: These threads will waste entire time slot spin locking
- Context switch can be substantial expensive
- Starvation problem still exist

Using Queues – to improve on spinning

- Too much is left to chance
- if the scheduler makes a bad choice, it picks up a thread
  - that is spin-waiting for the lock (1st approach)
  - that yields the CPU immediately (2nd approach)
- A lot of waste and possible starvation

- Can we have more control over which thread is scheduled next ?
- Keep a queue to track which threads are waiting to acquire the lock – used in Solaris OS
  - Schedule other thread that doesn't want to use lock()
- Many such ideas exist

How should locks be used?

- Thread-safe data structures
  - A lock should be acquired before accessing any variable or data structure that is shared between multiple threads
- Coarse-grained Locking
  - One big lock for all shared data
  - Easy to manage
- Fine-grained locking
  - Separate locks
  - More parallelism
  - Difficult to manage
- Note that OS only provides locks, correct locking discipline is left to users

Lecture Summary

- Spin-lock based approaches
    - Test-and-Set: Atomic -> hence lock is atomic
    - Compare-and-Swap: Atomic -> hence lock is atomic
  - Spin-lock based approaches are inefficient
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
    - Unbounded waiting
- Locks inside the OS are always spinlocks
- Sleeping Mutex Approach
  - Used in most user space lock implementations
- Priority Inversion problem with original Martian rover