CS 347M (Operating Systems Minor)

Spring 2022

Lecture 13: Locks

Mythili Vutukuru
CSE, IIT Bombay

Recap: Shared data access in threads

```
load counter → reg
reg = reg + 1
store reg → counter
```

- 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

T1

```
load counter → reg
reg = reg + 1
(context switch, save reg)
```

T2

```
load counter → reg
reg = reg + 1
store reg → counter
```

```
(resume, restore reg)
store reg → 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 (next topic)

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)
}

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

```
int isLocked = 0

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

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

T1

```
while(isLocked==1);
isLocked = 1

CRITICAL SECTION

isLocked = 0
```

T2

```
while(isLocked==1);
while(isLocked==1);
while(isLocked==1);
while(isLocked==1);
 isLocked = 1

CRITICAL SECTION
```

T1

```
while(isLocked==1);
(context switch, PC saved)



(resumes execution)
isLocked = 1
CRITICAL SECTION
```

T2

```
while(isLocked==1);
isLocked = 1
CRITICAL SECTION
```

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
- Such hardware atomic instructions used to implement locks: how?

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
}
```

Another instruction: compare-and-swap

- Another example: compare-and-swap 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
- 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
}
```

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
- 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
- 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

Locking in xv6

- No threads in xv6, no two user programs can access same memory image
  - No need for userspace locks like pthreads mutex
- However, scope for concurrency in xv6 kernel
  - Two processes in kernel mode in different CPU cores can access same kernel data structures like ptable
  - Even in single core, when a process is running in kernel mode, another trap occurs, trap handler can access data that was being accessed by previous kernel code
- Solution: spinlocks used to protect critical sections
  - Limit concurrent access to kernel data structures that can result in race conditions
- xv6 also has a sleeping lock (built on spinlock, not discussed)

Spinlocks in xv6

- Acquiring lock: uses xchg x86 atomic instruction (test and set)
  - Atomically set lock variable to new value and returns previous value
  - If previous value is 0, it means free lock has been acquired, success!
  - If previous value is 1, it means lock is held by someone, continue to spin in a busy while loop till success

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program
                     // that locked the lock.
};
```

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

Disabling interrupts for kernel spinlocks (1)

- When acquiring kernel spinlock, disables interrupts on CPU core: why?
  - What if interrupt and handler requests same lock: deadlock
  - Interrupts disabled only on local core, OK to spin for lock on another core
  - Why disable interrupts before even acquiring lock? (otherwise, vulnerable window after lock acquired and before interrupts disabled)
- Disabling interrupts not needed for userspace locks like pthread mutex
  - Kernel interrupt handlers will not deadlock for userspace locks

Process in kernel mode

Kernel spinlock L acquired
Interrupt, switch to trap handler

Interrupt handler

Spin to acquire L
DEADLOCK

Process in kernel mode

Kernel spinlock L acquired

CRITICAL SECTION

Spinlock released

On another core

Spin to acquire L
Spin
Spin
Spin
Spinlock L acquired

Disabling interrupts for kernel spinlocks (2)

- Function pushcli: disables interrupts on CPU core before spinning for lock
  - Interrupts stay disabled until lock is released
- What if multiple spinlocks are acquired?
  - Interrupts must stay disabled until all locks are released
- Disabling/enabling interrupts:
  - pushcli disables interrupts on first lock acquire, increments count for future locks
  - popcli decrements count, renables interrupts only when all locks released

```
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```