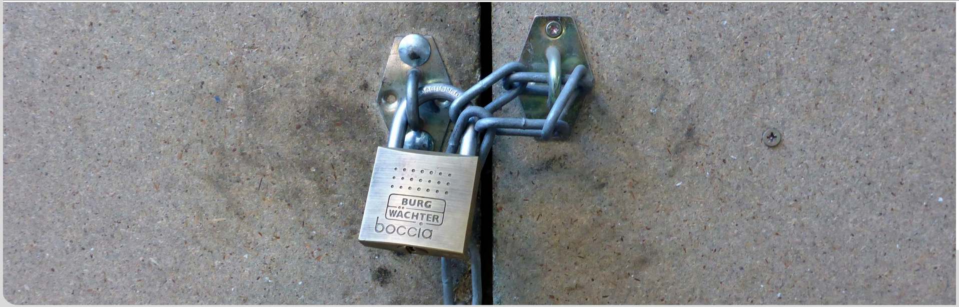
Operating Systems

12. Locks

Prof. Dr. Frank Bellosa | WT 2020/2021

Karlsruhe Institute of Technology (KIT) – ITEC – Operating Systems



Race Conditions (Motivation)

- Assume that we have sequential memory consistency
- Assume the following two code fragments are executed by two threads

Thread 1

```
count++;
```

Thread 2

```
count--;
```

- After both threads finished, **`count`** should still have the same value as before, but they often don't.

Thread 1 instructions

```
mov A count
add A 1
mov count A
```

Thread 2 instructions

```
mov A count
sub A 1
mov count A
```

Race Conditions (Example)

Thread 1 instructions

```
mov A count
add A 1
mov count A
```

Thread 2 instructions

```
mov A count
sub A 1
mov count A
```

Possible execution order (assume we initialized **`count`** to **`0`**)

```
mov A count    ; count = 0
sub A 1        ; decrement register, count still 0
mov A count    ; count = 0
add A 1        ; increment register, count still 0
mov count A    ; write -1 back to count
mov count A    ; write 1 back to count
```

- Both threads have private registers
  → separate **`A`** register exists for every thread (no problem here)
- However: **`count`** is now **`1`** instead of the expected **`0`**
  - We call this problem data race or race condition

What about single-instruction add/subtract?

- x86 allows single instruction **`add count 1`**
  - Race condition possible because of separate memory operations for load and store
- Only interlocked operations solve the problem!
  - Only applicable if there is a single interlocked operation for the problem
  - Interlocked operations are much more expensive than regular operations
    → Compiler will not generate interlocked operation for **`count++`**
- Critical Section (CS)
  - CS is a code sequence that might result in a race condition when executed concurrently
    → protect critical section from concurrent execution

Thread 1

```
enter_critical_section( &CS );
count++;
leave_critical_section( &CS );
```

Thread 2

```
enter_critical_section( &CS );
count--;
leave_critical_section( &CS );
```

Desired Properties for the Solution of the CS-Problem

- Mutual Exclusion

  At most one thread can be in the CS at any time
- Progress

  No thread running outside of the CS may block another thread from getting in
  - If no thread is in the CS, a thread trying to enter will eventually get in
  - If no thread can enter CS → don’t have progress
- Bounded Waiting/Fairness

  Once a thread starts trying to enter the CS, there is a bound on the number of times other threads get in
  - There is no bounded waiting if thread A waits to enter CS while B repeatedly leaves and re-enters CS infinitely
- Performance

  The time overhead added by using the lock for different cases of no/low/high contention

Mutual Exclusion

Disabling Interrupts

- The kernel only switches threads on interrupts
  - Usually on the timer interrupt
- Have per-thread “do not interrupt” (DNI) bit
- On a **single-core system** we can just implement
  - **`enter_critical_section()`** sets DNI bit
  - **`leave_critical_section()`** clears DNI bit
  - With interrupts disabled, the scheduler is never called
    → the thread runs until it reaches **`leave_critical_section()`**

+ Easy and convenient in the kernel

– Only works in single-core systems: disabling interrupts on one CPU does not affect other CPUs

– Only feasible in kernel: user code should not have the privilege to turn off interrupts (infinite loop, hogging the CPU)

Lock Variables

- Define a global variable **`lock`**
  - Only enter CS if **`lock`** is 0 and then set it to 1 when entering
  - Wait for lock to become 0 otherwise (busy waiting)

```
void enter_critical_section( volatile bool *lock )
{
   while( *lock != 0 )
      ; // wait for lock to become 0

   *lock = 1;
}

void leave_critical_section( volatile bool *lock )
{
   *lock = 0;
}
```

- To make the lock variables approach work, we need to **test and set** the lock variable atomically

Locking w. TestAndSet (SPARC ldstub,x86 xchg)

```
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
   }
```

```
typedef struct __lock_t {
    int flag;
} lock_t;
void init(lock_t *lock) {
    lock->flag = 0; // 0: lock is available, 1: lock is held
}
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1) ; // spin-wait (do nothing)
}
void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Locking w. Compare and Swap (x86, SPARC, 68k)

```
int CompareAndSwap(int *addr, int expected, int new)
{
int actual = *addr;
if (actual == expected)
*addr = new;
return actual;
}
```

```
void acquire(lock_t *lock) {
 while(CompareAndSwap(&lock->flag, 0, 1) == 1) ; // spin-wait (do nothing)
}
```

Lock w. Load-linked/Store-conditional(MIPS,ARM)

```
int LoadLinked(int *ptr) {
        return *ptr;
    }
    int StoreConditional(int *ptr, int value) {
        if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
        } else {
        return 0; // failed to update
        }
    }
```

```
    void lock(lock_t *lock) {
        while (1) {
            while (LoadLinked(&lock->flag) == 1) ; // spin until it is zero
            if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
            }
        }
```

Spinlocks with Atomic Store Operations

- Most modern CPUs provide atomic instructions with such semantics
  - Test memory word And Set value (TAS) (e.g., LDSTUB on SPARC V9)
  - Fetch and Add (e.g., XADD on x86)
  - Exchange contents of two memory words (SWAP, XCHG)
  - Compare content of one memory word and set to new value
    - Compare and Swap (e.g., CAS on SPARC V9 and Motorola 68k)
    - Compare and Exchange (e.g., CMPXCHG on x86)
  - Load-Link/Store-Conditional (LL/SC) (e.g., ARM, PowerPC, MIPS)

- Properties of atomic store operations

✓ **Mutual Exclusion** Only one thread can enter CS

✓ **Progress** Only the thread within the CS hinders others to get in

✗ **Bounded Waiting** No upper bound

✗ **Performance** Wasted time while spinnung on the CPU

Ticket Lock with Fetch-and-Add (x86 XADD)

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
typedef struct __lock_t {
    int ticket; //initialized with 0
    int turn;   //initialized with 0
} lock_t;

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn) ; // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Spinlock Limitations

- Spinlocks don't work well if the lock is contended
  - If most of the time there is no thread in the CS when another tries to enter then spinlocks are simple and efficient
  - If the CS is large or many threads try to enter, spinlocks might not be a good choice as many threads actively wait spinning
- Spinlocks don't work well if threads on different cores use the lock
  - The memory address is written at every atomic swap operation
    → memory is kept coherent between cores which is expensive (MESI protocoll)
- Spinlocks can behave unexpectedly when processes are scheduled with static priorities
  → priority inversion
  - Two threads (low and high priority) share a lock and are scheduled with static priorities
  - If the low priority thread holds the lock, the high priority thread has to wait for the low priority thread to release the CPU. However the low priotiy thread will not be dispatched.

Yield() Instead of Spinning

- When you are going to spin, give up the CPU to another thread
  - OS system call moves the caller from the running state to the ready state

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

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

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

  - The cost of a context switch can be substantial
  - Bad performance in case of many threads

Blocking Locks

Sleeping While Waiting

- Busy waiting. . .
  - wastes resources when threads wait for locks
  - stresses the cache coherence protocol when used across cores
  - can cause the priority inversion problem

- Idea for Blocking Locks
  - Threads should sleep on locks if they are occupied
  - Wake up threads one at a time when lock becomes free

Sleeping Instead of Spinning (w/ wakeup race)

```
    typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
        void lock_init(lock_t *m) {
            m->flag = 0;
            m->guard = 0; // guard spin-lock to protect flag AND queue
            queue_init(m->q);
        }
```

```
        void lock(lock_t *m) {
            while (TestAndSet(&m->guard, 1) == 1) ; // acquire guard by spinning
            if (m->flag == 0) {
                m->flag = 1; // lock is acquired
                m->guard = 0;
            } else {
                queue_add(m->q, gettid());
                m->guard = 0;
                park();
            }
        }
```

Sleeping Instead of Spinning

```
void unlock(lock_t *m) {
       while (TestAndSet(&m->guard, 1) == 1) ; // acquire guard by spinning
       if (queue_empty(m->q))
           m->flag = 0; // let go of lock; no one wants it
       else
           unpark(queue_remove(m->q)); // hold flag (for next thread!)
       m->guard = 0;
   }
```

Sleeping Instead of Spinning (w/ wakeup race)

```
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
   void lock_init(lock_t *m) {
       m->flag = 0;
       m->guard = 0; // guard spin-lock to protect flag AND queue
       queue_init(m->q);
   }
```

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1) ; // acquire guard by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

Sleeping Instead of Spinning (w/o wakeup race)

```
    typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
       void lock_init(lock_t *m) {
            m->flag = 0;
           m->guard = 0; // guard lock to protect flag AND queue
           queue_init(m->q);
       }
```

```
       void lock(lock_t *m) {
           while (TestAndSet(&m->guard, 1) == 1) ; // acquire lock by spinning
           if (m->flag == 0) {
               m->flag = 1; // lock is acquired
               m->guard = 0;
           } else {
               queue_add(m->q, gettid());
               setpark(); // new code to notify kernel of upcoming park()
               m->guard = 0;
               park();
           }
       }
```

Spinning vs. Blocking

- Each approach is better under different circumstances
- Uniprocessor
  - Waiting process is scheduled –> Process holding lock isn't
  - Waiting process should always relinquish processor
  - Associate queue of waiters with each lock
- Multiprocessor [DGT13]
  - Waiting process is scheduled –> Process holding lock might be running
  - Spin or block depends on time t, before lock is released
    - Lock released quickly –> Spin-wait
    - Lock released slowly –> Block
    - Quick and slow are relative to context-switch cost, C

When to Spin-Wait?
When to Block?

- If you know the time t, before the lock is released,
  you can determine optimal behavior
- How much CPU time is wasted when spin-waiting? Answer: t
- How much time is wasted when the thread blocks? Answer: C
- What is the best action when t<C? Answer: spin-wait
- What is the best action when t>C? Answer: block

- Optimal solution to the spin-vs-block-problem:
  - Requires knowledge of the future
  - Too much overhead to do any special prediction
    → Hybrid (2-Phase) locking with limited overhead in the worst case

Two-Phase Locks

- There is a userspace and kernel component
- Try to get into the CS with a userspace spinlock (phase 1)
- Leave the CS in userspace if no other thread is waiting, otherwise wakeup a blocked thread
- If the CS is busy use a syscall to put thread to sleep (phase 2)
- The thread is only woken up when the lock becomes free later
- Theory: Bound worst-case performance; ratio of actual/optimal
- When does worst-possible performance occur? $\rightarrow$ Spin for very long time $t \gg C$
- 2-Phase-algorithm: Spin-wait for C then block $\rightarrow$ Factor of 2 of optimal
  - $t < C$: optimal spin-waits for t; we spin-wait t too
  - $t >= C$: optimal blocks immediately (cost of C)
    - we spin C then block (cost of 2C)
    - $\frac{2C}{C}$ $\rightarrow$ 2-competitive algorithm

Non-blocking Synchronization (Outlook) [HS12]

- Design algorithm to avoid critical sections
- Requires: atomic instructions that atomically set state based on a condition
- Can implement many common data structures lockless
  - stacks
  - queues
  - hash tables
- Does not guarantee better “performance”
  - Seldomly faster
  - Almost always more scalable!

More Atomic Operations

- We have already studied the atomic single word Compare And Swap (CAS) instruction in the context of spinlocks
- CAS of limitted use for implementing lockless data structures
  - The more bits a CAS can compare and swap atomically, the easier it is to implement lockfree data structures
- Modern CPUs contains larger versions of CAS called Double-Width Compare And Swap (DWCAS)
  - Intel Architecture: **`CMPXCHG8`** to CAS 64-Bit, **`CMPCHG16B`** for 128-Bit CAS
- Another idea was to extend CPUs to compare and swap multiple locations atomically Double Compare and Swap (DCAS)
  - Cannot be emulated efficiently on systems with atomic ops on single location

Advantages of Non-blocking Synchronization

- Thread-killing immunity
  Any thread forcefully killed in the system won't delay other threads
  - What happens when a thread is killed that currently holds a lock?
- Signal immunity
  CS synchronized with locks shouldn't call functions that might block
  → Could otherwise block with lock held
  - Need to carefully write interrupt handlers that don't allocate memory
  - No such problems with lock-free/wait-free CS
- Priority inversion immunity
  - Priority inversion cannot occur when synchronizing without locks

Transactional Memory (Outlook)

- Idea: CPU support for memory modification transaction
  e.g. Intel TSX with Restricted Transactional Memory (RTM) [YHLR13]
  - Instructions to begin a transaction and to commit
  - Perform modification within transaction only in cache
  - Commit: Write back modified cache lines if the source lines have not been touched by other cores while the transaction was going on
  - Abort: Discards all transactional updates, restores architectural state, and resumes execution.
    - Source lines were touched during transaction
    - Exceeding buffer capacity (depends on cache size and associativity)
    - Instructions that always abort (e.g., syscall)

References I

[DGT13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 33–48, New York, NY, USA, 2013. Association for Computing Machinery.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[YHLR13] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.