CS330: Operating Systems

Locks

Recap: Synchronization and locking

- Locking is necessary when multiple contexts access shared resources
- Example: Multiple threads, multiple OS execution contexts
- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme
- Contending threads should not starve for the lock (infinitely)

Recap: Synchronization and locking

- Locking is necessary when multiple contexts access shared resources
- Example: Multiple threads, multiple OS execution contexts
- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme
- Contending threads should not starve for the lock

Agenda: Spinlocks, Semaphore and mutex (waiting locks)

Spinlock: Buggy attempt

```
lock_t *L; // Initial value = 0
lock(L)
{
   while(*L);
   *L = 1;
}
unlock(L)
{
   *L = 0;
}
```

- Does this implementation work?

Spinlock: Buggy attempt

```
lock_t *L; // Initial value = 0
lock(L)
{
   while(*L);
   *L = 1;
}
unlock(L)
{
   *L = 0;
}
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?

Spinlock: Buggy attempt

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.      while(*L);
5.      *L = 1;
6.  }
7.  unlock(L)
8.  {
9.      *L = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?
  - Single core: Context switch between line #4 and line #5
  - Multicore: Two cores exiting the while loop by reading lock = 0

Spinlock: Buggy attempt

```
1.  lock_t *L; // Initial value = 0
2.  lock(L)
3.  {
4.     while(*L);
5.     *L = 1;
6.  }
7.  unlock(L)
8.  {
9.     *L = 0;
10. }
```

- Does this implementation work?
- No, it does not ensure *mutual exclusion*
- Why?
  - Single core: Context switch between line #4 and line #5
  - Multicore: Two cores exiting the while loop by reading lock = 0
- Core issue: Compare and swap has to happen atomically!

Spinlock using atomic exchange

```
lock_t *L; // Initial value = 0
lock(L)
{
   while(atomic_xchg(L, 1));
}
unlock(L)
{
   *L = 0;
}
```

- Atomic exchange: exchange the value of memory and register atomically
- atomic_xchg (int *PTR, int val) returns the value at PTR before exchange
- Ensures mutual exclusion if "val" is stored on a register
- No fairness guarantees

Spinlock using XCHG on X86

```
lock(lock_t *L )
{
   asm volatile(
   “mov $1, %%rax; ”
   “try: xchg %%rax, (%%rdi); ”
   “cmp $0, %%rax;”
    “jne try; ”
     : : : “memory” );
}
unlock(int *L ) { *L = 0;}
```

- XCHG *R*, *M* ⇒ Exchange value of register *R* and value at memory address *M*
- *RDI* register contains the lock argument
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

Spinlock using compare and swap

```
lock_t *L; // Initial value = 0
lock(L)
{
   while( CAS(L, 0, 1) );
}
unlock(L)
{
   *L = 0;
}
```

- Atomic compare and swap: perform the condition check and swap atomically
- CAS (int **PTR*, int *cmpval*, int *newval*) sets the value of *PTR* to *newval* if *cmpval* is equal to value at *PTR*. Returns 0 on successful exchange
- No fairness guarantees!

CAS on X86: cmpxchg

cmpxchg source[Reg] destination [Mem/Reg]
Implicit registers : rax and flags

```
if rax == [destination]
then
        flags[ZF] = 1
        [destination] = source
else
        flags[ZF] = 0
        rax = [destination]
```

- “cmpxchg” is not atomic in X86, should be used with a “lock” prefix

Spinlock using CMPXCHG on X86

```
lock(lock_t *L )
{
asm volatile(
    "mov $1, %%rcx;"
    "try: xor %%rax, %%rax;"
    "lock cmpxchg %%rcx, (%%rdi);"
    "jnz try; "
    : : : "rcx",  "rax", "memory");
}
unlock(lock_t *L ) { *L = 0;}
```

- Value of RAX (=0) is compared against value at address in register RDI and exchanged with RCX (=1), if they are equal
- Exercise: Visualize a context switch between any two instructions and analyse the correctness

Load Linked (LL) and Store conditional (SC)

- LoadLinked (R, M)
  - Like a normal load, it loads R with value of M
  - Additionally, the hardware keeps track of future stores to M
- StoreConditional (R, M)
  - Stores the value of R to M if no stores happened to M after the execution of LL instruction (after execution, R = 1)
  - Otherwise, store is not performed (after execution R=0)
- Supported in RISC architectures like mips, risc-v etc.

Spinlock using LL and LC

```
lock_t *L;    //initial value = 0
lock( lock_t *L)
{
 while(LoadLinked(L) ||
       !StoreConditional(L, 1));
}
unlock( lock_t *L) { *L = 0;}
```

```
try:  LL   R1,  (R2);  //R2 = lock address
      BNEQZ  R1,  try;
      ADDUI  R1,  R0, #1;   //R1 = 1
      SC R1, (R2)
      BEQZ R1, try
```

- Efficient as the hardware avoids memory traffic for unsuccessful lock acquire attempts
- Context switch between LL and SC results in SC to fail

Spinlocks: reducing wasted cycles

- Spinning for locks can introduce significant CPU overheads and increase energy consumption
- How to reduce spinning in spinlocks?

Spinlocks: reducing wasted cycles

- Spinning for locks can introduce significant CPU overheads and increase energy consumption
- How to reduce spinning in spinlocks?
- Strategy: Back-off after every failure, exponential back-off used mostly

```
lock( lock_t *L) {
        u64 backoff = 0;
        while(LoadLinked(L) || !StoreConditional(L, 1)){
                if(backoff < 63)   ++backoff;
                        pause(1 << backoff);   // Hint to processor
}
```

Fairness in spinlocks

- Spinlock implementations discussed so far are not fair,
  - no bounded waiting
- To ensure fairness, some notion of ordering is required
- What if the threads are granted the lock in the order of their arrival to the lock contention loop?
  - A single lock variable may not be sufficient
  - Example solution: Ticket spinlocks

Atomic fetch and add (xadd on X86)

xadd R, M

TmpReg T = R + [M]

R = [M]

[M] = T

- Example: M = 100; RAX = 200
- After executing "lock xadd %RAX, M", value of RAX = 100, M = 300
- Require lock prefix to be atomic

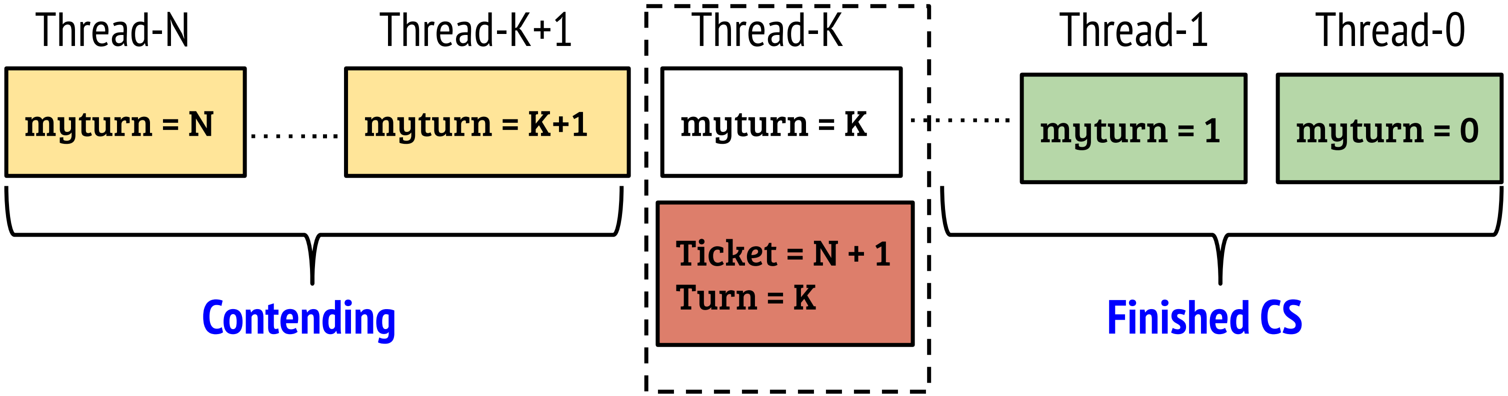
Ticket spinlocks (OSTEP Fig. 28.7)

```
struct lock_t{
        long ticket;
        long turn;
};
void init_lock (struct lock_t *L){
   L → ticket = 0;  L → turn = 0;
}
void unlock(struct lock_t *L){
      L → turn++;
}
```

```
void lock(struct lock_t *L){
  long myturn = xadd(&L → ticket, 1);
  while(myturn != L → turn)
        pause(myturn - L → turn);
}
```

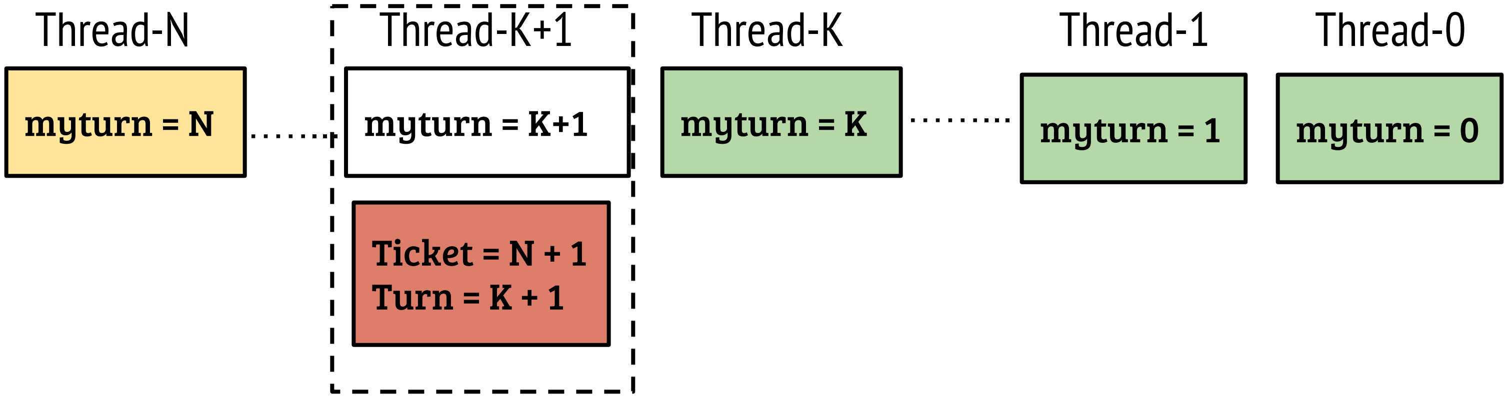
- Example: Order of arrival: T1 T2 T3
- T1 (in CS) : myturn = 0, L = {1, 0}
- T2: myturn = 1, L = {2, 0}
- T3: myturn = 2, L = {3,0}
- T1 unlocks, L = {3, 1}. T2 enters CS

Ticket spinlock



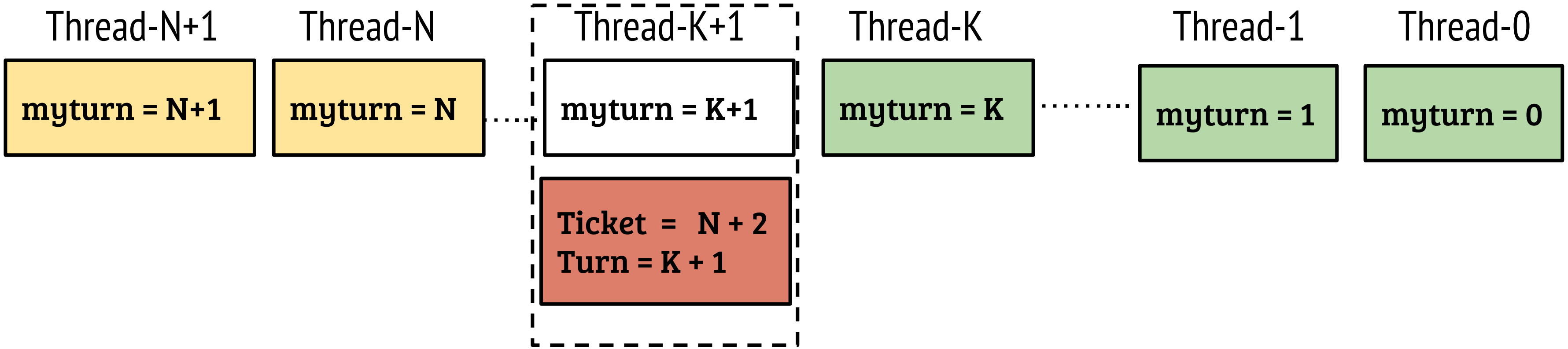
- Local variable “myturn” is equivalent to the order of arrival
- If a thread is in CS ⇒ Local Turn must be same as “Turn”
- Threads waiting = Ticket - Turn -1

Ticket spinlock



- Value of turn incremented on lock release
- Thread which arrived just after the current thread enters the CS
- When a new thread arrives, it gets the lock after the other threads ahead of the new thread acquire and release the lock

Ticket spinlock



- Ticket spinlock guarantees bounded waiting
- If *N* threads are contending for the lock and execution of the CS consumes *T* cycles, then bound = N * T (assuming negligible context switch overhead)

Ticket spinlock (with yield)

```
void lock(struct lock_t *L){
  long myturn = xadd(&L → ticket, 1);
  while(myturn != L → turn)
        sched_yield( );
}
```

- Why spin if the thread's turn is yet to come?
- Yield the CPU and allow the thread with ticket (or other non contending threads)
- Further optimization
  - Allow the thread with "myturn" value one more than "L⟶ turn" to continue spinning

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

```
struct BST{
        struct node *root;
        rwlock_t *lock;
};
```

```
struct node{
        item_t item;
        struct node *left;
        struct node* right;
};
```

```
void insert(BST *t, item_t item);
void lookup(BST *t, item_t item);
```

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

```
struct BST{
        struct node *root;
        rwlock_t *lock;
};
```

```
struct node{
        item_t item;
        struct node *left;
        struct node* right;
};
```

```
void insert(BST *t, item_t item);
void lookup(BST *t, item_t item);
```

- If multiple threads call lookup( ), they may traverse the tree in parallel

Implementation of read-write locks

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

```
init_lock(rwlock_t *rL)
{
   init_lock(&rL → read_lock);
   init_lock(&rL → write_lock);
   rL → num_readers = 0;
}
```

Implementation of read-write locks (writers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

```
init_lock(rwlock_t *rL)
{
   init_lock(&rL → read_lock);
   init_lock(&rL → write_lock);
   rL → num_readers = 0;
}
```

```
void write_lock(rwlock_t *rL)
{
    lock(&rL → write_lock);
}
```

```
void write_unlock(rwlock_t *rL)
{
    unlock(&rL → write_lock);
}
```

- Write lock behavior is same as the typical lock, only one thread allowed to acquire the lock

Implementation of read-write locks (readers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

```
void read_lock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers++;
   if(rL → num_readers == 1)
      lock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

```
void read_unlock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers--;
   if(rL → num_readers == 0)
      unlock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

Implementation of read-write locks (readers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

- The first reader acquires the write lock prevents writers to acquire lock
- The last reader releases the write lock to allow writers

```
void read_lock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers++;
   if(rL → num_readers == 1)
       lock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

```
void read_unlock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers--;
   if(rL → num_readers == 0)
       unlock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

Software lock: Buggy #1

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    while(flag[id ^ 1]); // ^ → XOR
     flag[id] = 1;
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- We have seen that this solution does not work, Why?

Software lock: Buggy #1

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    while(flag[id ^ 1]); // ^ → XOR
     flag[id] = 1;
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- We have seen that this solution does not work, Why?
- Both threads can acquire the lock as "while condition check" and "setting the flag" is non-atomic

Software lock: Buggy #2

```
int flag[2] = {0,0};
void lock (int id)   /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ^ 1]); // ^ → XOR
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Does this solution work?

Software lock: Buggy #2

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ^ 1]); // ^ → XOR
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Does this solution work?
- No, as this can lead to a deadlock (flag[0] = flag[1] = 1) In other words the "progress" requirement is not met
- Progress: If no one has acquired the lock and there are contending threads, one of the threads must acquire the lock within a finite time

Software lock: Buggy #3

```
int turn = 0;
void lock (int id)   /*id = 0 or 1 */
{
   while(turn == id ^ 1));
}
void unlock (int id)
{
    turn = id ^ 1;
}
```

- Assuming T_0 invokes lock( ) first, does the solution provide mutual exclusion?

Software lock: Buggy #3

```
int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   while(turn == id ^ 1));
}
void unlock (int id)
{
    turn = id ^ 1;
}
```

- Assuming T_0 invokes lock( ) first, does the solution provide mutual exclusion?
- Yes it does, but there is another issue with this solution - two threads must request the lock in an alternate manner
- Progress requirement is not met
  - Argument: one of the threads stuck in an infinite loop (in non-CS code)

Peterson's solution

```
int flag[2] = {0,0};   int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Homework: Prove that mutual exclusion is guaranteed
- What about fairness?

Peterson's solution

```
int flag[2] = {0,0};   int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Homework: Prove that mutual exclusion is guaranteed
- What about fairness?
- The lock is fair because if two threads are contending, they acquire the lock in an alternate manner
- Extending the solution to N threads is possible