CS330: Operating Systems

Locking

Recap: Strategy to handle race conditions in OS

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

Locking example: pthread mutex

```
pthread_mutex _t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
void *thfunc(void *)
{
  int ctr = 0;
  for(ctr=0; ctr<100000; ++ctr){
    pthread_mutex_lock(&lock);      // One thread acquires lock, others wait
    counter++;                      // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Design issues of locks

```
pthread_mutex _t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
    pthread_mutex_lock(&lock);     // One thread acquires lock, others wait
    counter++;                     // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

- Efficiency of lock and unlock operations
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

Lock ADT

```
lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

```
lock_t *L1, L2;
....
lock(L1)
Critical Section
unlock(L1)
....
lock(L2)
Critical Section
unlock(L2)
....
Lock(L1)
Critical Section
unlock(L2)
```

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations
- Software locks are implemented without assuming any hardware support
  - Not used in practice because of high overheads

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

```
    counter++;                          // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Lock: busy-wait (spinlock) vs. Waiting

T1	T2
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section unlock(L)

Lock: busy-wait (spinlock) vs. Waiting

T1	T2
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section unlock(L)

- With busy waiting, context switch overheads saved, wasted CPU cycles due to spinning
- Busy waiting is prefered when critical section is small and the context executing the critical section is not rescheduled (e.g., due to I/O wait)

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- 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

```
    pthread_mutex_unlock(&lock); // Release the lock
  }
}
```

Fairness

- Given *N* threads contending for the lock, number of unsuccessful attempts for lock acquisition for all contending threads should be same
- Bounded wait property
  - Given *N* threads contending for the lock, there should be an upper bound on the number of attempts made by a given context to acquire the lock

Design issues of locks

- 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

Spinlock: Buggy attempt

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

- Does this implementation work?

Spinlock: Buggy attempt

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

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

Spinlock: Buggy attempt

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

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

- 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;”
   “loop: xchg %%rax, (%%rdi);”
   “cmp $0, %%rax;”
    “jne loop;”
     : : : “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

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

- 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

```
1.  if rax == [destination]
2.  then
3.      flags[ZF] = 1
4.      [destination] = source
5.  else
6.      flags[ZF] = 0
7.      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;”
    “loop: xor %%rax, %%rax;”
    “lock cmpxchg %%rcx, (%%rdi);”
    “jnz loop;”
    : : : “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 = 0;
lock( lock_t *L)
{
 while(LoadLinked(L) ||
       !StoreConditional(L, 1));
}
unlock( lock_t *L) { *L = 0;}
```

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

- 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

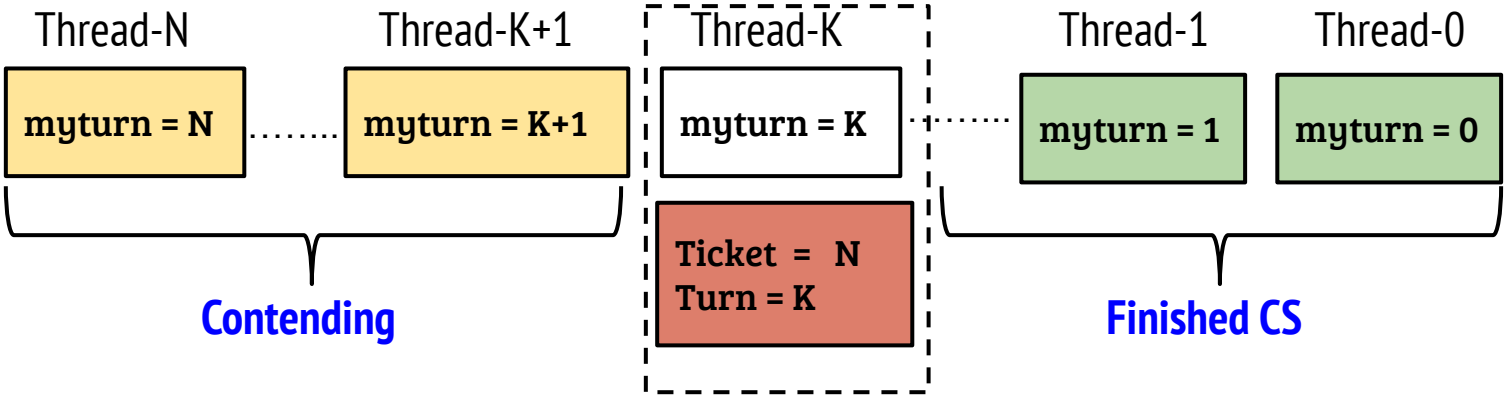
Ticket spinlocks (OSTEP Fig. 28.7)

```
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);
}
```

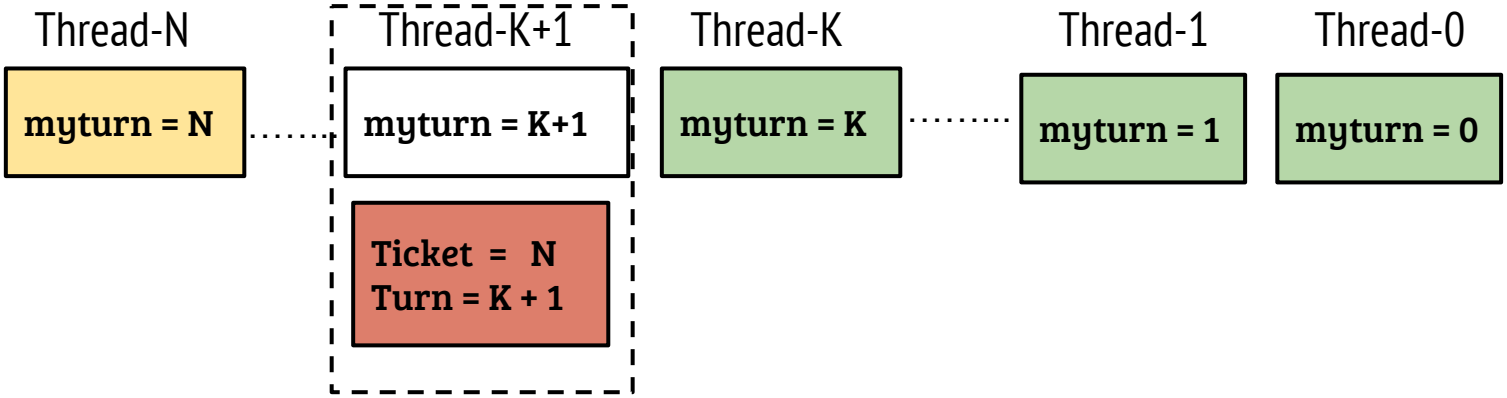
- Example: Order of arrival: T1 T2 T3
- T1 (in CS) : myturn = 0, L = {1, 0}
- T2: myturn = 1, L = {2, 0}
- T3: myturn = 2, L = {3,0}
- T1 unlocks, L = {3, 1}. T2 enters CS

Ticket spinlock



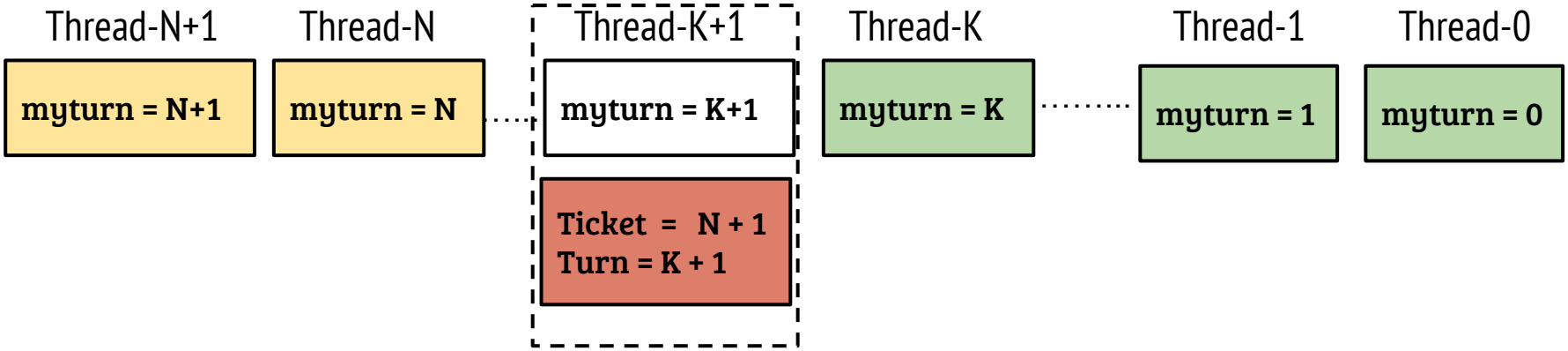
- Local variable “myturn” is equivalent to the order of arrival
- If a thread is in CS ⇒ Local Turn must be same as “Turn”
- Threads waiting = Ticket - Turn

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 less than "L⟶ turn" to continue spining