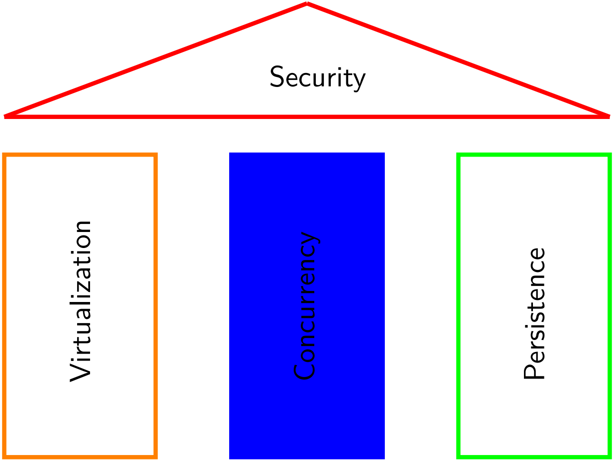
CS323 Operating Systems

Locking

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021

Concurrency



Topics covered in this lecture

- Review of threading and mutual exclusion
- Abstraction: locks to protect shared data structures
- Mechanism: interrupt-based locks
- Mechanism: atomic hardware locks
- Busy waiting (spin locks) versus wait queues

This slide deck covers chapters 28, 29, 30 in OSTEP.

Threads: Executions context

- Threads are independent execution context
- Similar to processes

Threads: Executions context

- Threads are independent execution context
- Similar to processes
- Except that they share the same address space

Threads: Executions context

- Threads are independent execution context
- Similar to processes
- Except that they share the same address space
- Why do we need threads?

Threads: Executions context

- Threads are independent execution context
- Similar to processes
- Except that they share the same address space
- Why do we need threads?
- CPUs run very fast, they might get blocked for fetching data
- Multiples of CPUs are available that can do a job in parallel

For parallelism and concurrency

- **Parallelism**: multiple threads (or processes) working on a single task using multiple CPU cores
- **Concurrency**: tasks can start, run, and complete in overlapping time periods, e.g., through time multiplexing by interleaving their executions, or through parallelism when they are executed at the same time

Note that processes can share information through partially overlapping address spaces or by communicating (future lectures).

Race conditions

```c
int cnt = 0;
void *incer(void *arg) {
  printf("%s starts\n", (char*)arg);
  for (int i=0; i < 1000000; ++i) {
    cnt = cnt + 1;
  }
  return NULL;
}
int main(int argc, char *argv[]) {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, incer, "T1");
  pthread_create(&t2, NULL, incer, "T2");
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Counter: %d (expected: %d)\n", cnt, 1000000*2);
  return 0;
}
```

Race conditions: what is happening?

```
$ ./21-race
T1 starts
T2 starts
T1 is done
T2 is done
Counter: 1150897 (expected: 2000000)
$
```

Race conditions: what is happening?

```
$ ./21-race
T1 starts
T2 starts
T1 is done
T2 is done
Counter: 1150897 (expected: 2000000)
$
```

Assembly of `incer`:

```
mov     0x601044,%eax ; load value
add     $0x1,%eax     ; increment
mov     %eax,0x601044 ; store value
```

Race conditions: what is happening?

```
$ ./21-race
T1 starts
T2 starts
T1 is done
T2 is done
Counter: 1150897 (expected: 2000000)
$
```

Assembly of `incer`:

```
mov     0x601044,%eax ; load value
add     $0x1,%eax     ; increment
mov     %eax,0x601044 ; store value
```

Both threads load the same value, increment, and write back. The addition of one thread is lost!

Race conditions

- Concurrent execution leads to race conditions
  - Access to shared data must be mediated
- **Critical section:** part of code that accesses shared data
- **Mutual exclusion:** only one process is allowed to execute critical section at any point in time
- **Atomicity:** critical section executes as an uninterruptible block

A **mechanism** to achieve atomicity is through locking.

Locks: basic idea

- Lock variable protects critical section
- All threads competing for *critical section* share a lock
- Only one thread succeeds at acquiring the lock (at a time)
- Other threads must wait until lock is released

```
lock_t mutex;
...
lock(&mutex);
cnt = cnt + 1;
unlock(&mutex);
```

Locks: basic idea

- Requirements: mutual exclusion, fairness, and performance
  - **Mutual exclusion**: only one thread in critical section
  - **Fairness**: all threads should eventually get the lock
  - **Performance**: low overhead for acquiring/releasing lock
- Lock implementation requires hardware support
  - ... and OS support for performance

Lock operations

- `void lock(lock_t *lck)`: acquires the lock, current thread owns the lock when function returns
- `void unlock(lock_t *lck)`: releases the lock

Lock operations

- `void lock(lock_t *lck)`: acquires the lock, current thread owns the lock when function returns
- `void unlock(lock_t *lck)`: releases the lock

Note that we assume that the application *correctly* uses locks for *each* access to the critical section.

Interrupting locks

- Turn off interrupts when executing critical sections
  - Neither hardware nor timer can interrupt execution
  - Prevent scheduler from switching to another thread
  - Code between interrupts executes atomically

```
void acquire(lock_t *l) {
  disable_interrupts();
}

void release(lock_t *l) {
  enable_interrupts();
}
```

Interrupting locks (disadvantages)

- No support for locking multiple locks
- Only works on uniprocessors (no support for locking across cores in multicore system)
- Process may keep lock for arbitrary length
- Hardware interrupts may get lost (hardware only stores information that interrupt X happened, not how many times it happened)

Interrupting locks (perspective)

- Interrupt-based locks are extremely simple
- Work well for low-complexity code

Interrupting locks (perspective)

- Interrupt-based locks are extremely simple
- Work well for low-complexity code

- Implementing locks through interrupts is great for MCUs

(Faulty) spin lock

- Use a shared variable to synchronize access to critical section

```
bool lock1 = false;

void acquire(bool *lock) {
  while (*lock); /* spin until we grab the lock */
  *lock = true;
}

void release(bool *lock) {
  *lock = false
}
```

(Faulty) spin lock

- Use a shared variable to synchronize access to critical section

```
bool lock1 = false;

void acquire(bool *lock) {
  while (*lock); /* spin until we grab the lock */
  *lock = true;
}

void release(bool *lock) {
  *lock = false
}
```

Bug: both threads can grab the lock if thread is preempted before setting the lock but after the `while` loop completes.

Required hardware support

Locking requires an atomic *test-and-set* instruction.

```
int tas(int *addr, int val) {
  int old = *addr;
  *addr = val;
  return old;
}
```

Required hardware support

Locking requires an atomic *test-and-set* instruction.

```
int tas(int *addr, int val) {
  int old = *addr;
  *addr = val;
  return old;
}
```

```
int tas(int *addr, int val) {
  int old;
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (old) :
               "1" (val) : "cc");
  return old;
}
```

Required hardware support

- Hardware support is required for (i) an instruction that updates memory location and returns old value and (ii) executes the instruction atomically.
- Directly encoding inline assembly is error prone, use intrinsics instead:

```
type __sync_lock_test_and_set(type *ptr, type val);
```

Test-and-set spin lock

```
int lock1;

void acquire(int *l) {
  while (__sync_lock_test_and_set(l, 1) == 1); /* spin */
}

void release(int *l) {
  *l = 0;
}

acquire(&lock1);
critical_section();
release(&lock1);
```

Compare-and-swap spin lock

```
bool cas(T *ptr, T expt, T new) {
  if (*ptr == expt) {
    *ptr = new;
    return true;
  }
  return false;
}
```

The function compares the value at `*ptr` and if it is equal to `expt` then the value is overwritten with `new`. The function returns `true` if the swap happened.

Compare-and-swap spin lock

```
__sync_bool_compare_and_swap(T *ptr, T expt, T new);
```

How would you implement the lock `acquire` operation?

Compare-and-swap spin lock

```
__sync_bool_compare_and_swap(T *ptr, T expt, T new);
```

How would you implement the lock acquire operation?

```
void acquire_cas(bool *lck) {
  while (__sync_bool_compare_and_swap(lck, false, true)
        == false);
}
```

Spin lock: reduce spinning

- A simple way to reduce the cost of spinning is to `yield()` whenever lock acquisition fails
  - This is no longer a "strict" spin lock as we give up control to the scheduler every loop iteration

```
void acquire(bool *lck) {
  while (__sync_lock_test_and_set(l, 1) == 1) {
    yield();
  }
}
```

Lock requirements: spin locks

- **Correctness:** mutual exclusion, progress, and, bounded
  - Mutual exclusion: $\leq$ one thread in critical section at a time
  - Progress (deadlock freedom): one waiting process will proceed
  - Bounded (no starvation): eventually each process will proceed
- Fairness: each thread waits for the same amount of time
- Performance: CPU is not used unecessarily

Lock requirements: spin locks

- **Correctness:** mutual exclusion, progress, and, bounded
  - Mutual exclusion: $\leq$ one thread in critical section at a time
  - Progress (deadlock freedom): one waiting process will proceed
  - Bounded (no starvation): eventually each process will proceed
- Fairness: each thread waits for the same amount of time
- Performance: CPU is not used unecessarily

Spinlocks are unfair (threads race for lock) and hog performance (spinning and burning CPU time)!

Queue lock

- Idea: instead of spinning, put threads on a queue
- Wake up thread(s) when lock is released
  - Wake up all threads to have them race for the lock
  - Selectively wake one thread up for fairness

Queue lock implementation: nptl

```
/* Bit 31 clear means unlocked; bit 31 set means locked.
   Remaining bits encode num. interested threads. */
static inline void mutex_lock(int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex.  (fastpath). */
  if (atomic_bit_test_set(mutex, 31) == 0) return;
  atomic_increment(mutex);
  while (1) {
    if (atomic_bit_test_set(mutex, 31) == 0) {
      atomic_decrement(mutex); return;
    }
    /* We have to wait. Make sure futex is act. locked */
    v = *mutex;
    if (v >= 0) continue;
    futex_wait(mutex, v);
  }
}
```

Queue lock implementation: nptl

```
static inline void mutex_unlock(int *mutex) {
  /* Adding 0x80000000 to the counter results in 0 iff
     there are no other waiting threads (fastpath).  */
  if (atomic_add_zero(mutex, 0x80000000)) return;

  /* There are other threads waiting, wake one up. */
  futex_wake(mutex, 1);
}
```

Do you want to know more? Check out the Linux futex system call.

Comparison spinlock / queue lock

- Spinlock works well when critical section is short and rare and we execute on more than one CPU (i.e., no context switch, likely to acquire lock soon)
- Queue locks work well when critical section is longer or more frequent (i.e., high contention, likelihood that thread must wait)

Comparison spinlock / queue lock

- Spinlock works well when critical section is short and rare and we execute on more than one CPU (i.e., no context switch, likely to acquire lock soon)
- Queue locks work well when critical section is longer or more frequent (i.e., high contention, likelihood that thread must wait)
- Hybrid approach: spin for a while, then yield and enqueue

Lock principles

- Locks protect access to shared data structures
- Shared kernel data structures rely on locks
- Locking strategy: coarse-grained (one lock) versus fine-grained (many locks)
- OS only provides locks, locking strategy is up to programmer

Lock best practices

- When acquiring a lock, recheck assumptions
- Ensure that all shared information is refreshed (and not stale)
- Multiple threads may wake up and race for the lock (i.e., loop if unsuccessful)

Summary

- Locks enforce mutual exclusion for critical section (i.e., an object that can only be owned by a single thread)
- Trade-offs between spinlock and queue lock
  - Time lock is held
  - Contention for lock
  - How many concurrent cores execute
- Locking requires kernel support or atomic instructions
  - **test-and-set** atomically modifies the contents of a memory location, returning its old value
  - **compare-and-swap** atomically compares the contents of a memory location to a given value and, iff they are equal, modifies the contents of that memory location to a given new value.

Don't forget to get your learning feedback through the Moodle quiz!