CSE 323: Operating System Design

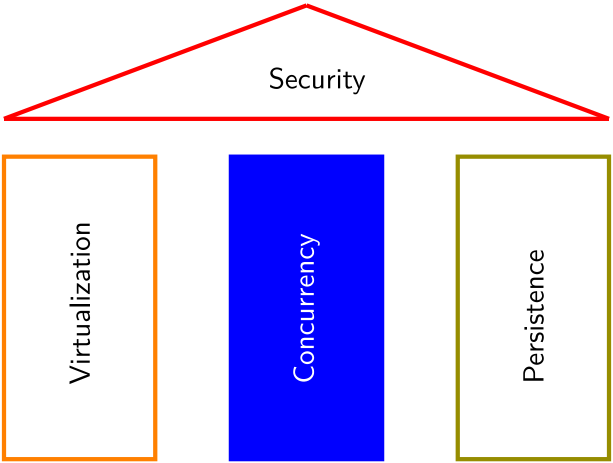
Locking

Salman Shamil

North South University (NSU)

Fall 2025

Original slides by Mathias Payer and Sanidhya Kashyap [EPFL]


Security
Virtualization
Concurrency
Persistence

Lecture Topics

- 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 in OSTEP.

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

```c
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.

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

This pseudocode in **c** demonstrates the basic idea of an atomic exchange instruction (`xchg` on `x86` or `ldstub` on SPARC).

Test-and-set Spin Lock

```
int lock1; // 0 -> lock is available, 1 -> lock is held

void acquire(int *lock) {
  while (TestAndSet(lock, 1) == 1); /* spin */
}

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

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

This time we guarantee that the thread that changes lock from 0 to 1 gets to execute its critical section.

Compare-and-swap Spin Lock

```c
int CompareAndSwap(int *ptr, int expt, int new) {
  int actual = *ptr;
  if (actual == expt) {
    *ptr = new;
  }
  return actual;
}
```

Compare-and-swap Spin Lock

```c
int CompareAndSwap(int *ptr, int expt, int new) {
  int actual = *ptr;
  if (actual == expt) {
    *ptr = new;
  }
  return actual;
}
```

- Returns the actual value (before the potential update), indicating whether it succeeded or not.
- More powerful than test-and-set [blind *vs* conditional update]

Compare-and-swap Spin Lock

```
int CompareAndSwap(int *ptr, int expt, int new) {
  int actual = *ptr;
  if (actual == expt) {
    *ptr = new;
  }
  return actual;
}
```

- Returns the actual value (before the potential update), indicating whether it succeeded or not.
- More powerful than test-and-set [blind *vs* conditional update]

```
void acquire_cas(int *lock) {
  while (CompareAndSwap(lock, 0, 1) == 1); /* spin */
}
```

Ticket Lock with Fetch-And-Add

- Neither test-and-set nor compare-and-swap guarantees progress.
  - A thread may spin forever.

Ticket Lock with Fetch-And-Add

- Neither test-and-set nor compare-and-swap guarantees progress.
  - A thread may spin forever.
- Another hardware primitive `Fetch-And-Add`

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

Ticket Lock with Fetch-And-Add

- Neither test-and-set nor compare-and-swap guarantees progress.
  - A thread may spin forever.
- Another hardware primitive `Fetch-And-Add`

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

- `Fetch-And-Add` can be used to build Ticket Lock, where a thread once queued, will eventually acquire the lock.

```
typedef struct __lock_t {
  int ticket;
  int turn;
} lock_t;
```

Ticket Lock with Fetch-And-Add

```c
void lock_init(lock_t *lock) {
  lock->ticket = 0;
  lock->turn = 0;
}

void lock(lock_t *lock) {
  // get my ticket
  int myturn = FetchAndAdd(&lock->ticket);
  while (lock->turn != myturn) {
    ; // spin until it's my turn
  }
}

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

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 (TestAndSet(l, 1) == 1) {
    yield();
  }
}
```

A Better Way: 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
- OS Support: `park()` and `unpark(threadID)`