Chapter 28: Locks

Overview

- **Main Idea**: Locks enforce mutual exclusion in critical sections using hardware/OS support, preventing race conditions.
- **Key Terms**: Spinlock, Test-and-Set, Compare-and-Swap, Ticket Lock, Yield, Priority Inversion.

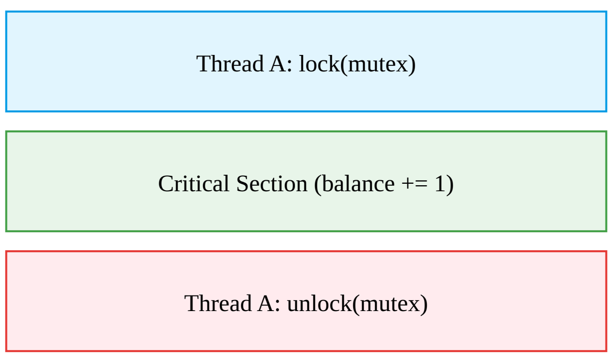
---

1. Lock Basics

Purpose

- **Atomicity**: Ensure critical sections (e.g., `balance = balance + 1`) execute as if they were a single instruction.
- **Mutual Exclusion**: Only one thread can hold a lock at a time.

```
lock_t mutex;
lock(&mutex);
balance = balance + 1;  // Critical section
unlock(&mutex);
```



---

2. Evaluating Locks

Criterion	Description
Correctness	Does the lock prevent multiple threads from entering the critical section?
Fairness	Do threads get a fair chance to acquire the lock? (No starvation)

Criterion	Description
Performance	Overhead of lock acquisition/release under contention/no-contention scenarios.

---

3. Hardware Support for Locks

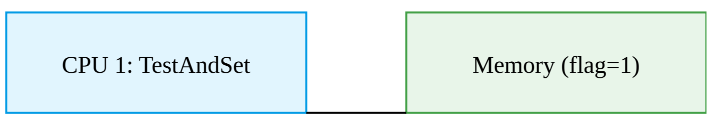
3.1 Test-and-Set (Atomic Exchange)

```c
int TestAndSet(int *ptr, int new) {
    int old = *ptr;
    *ptr = new;
    return old;
}

// Spinlock implementation
typedef struct __lock_t { int flag; } lock_t;

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // Spin
}
```

Problem: Spins waste CPU cycles.



Atomic Read-Modify-Write

3.2 Compare-and-Swap (CAS)

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

```c
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // Spin
}
```

Advantage: More versatile (used in lock-free algorithms).

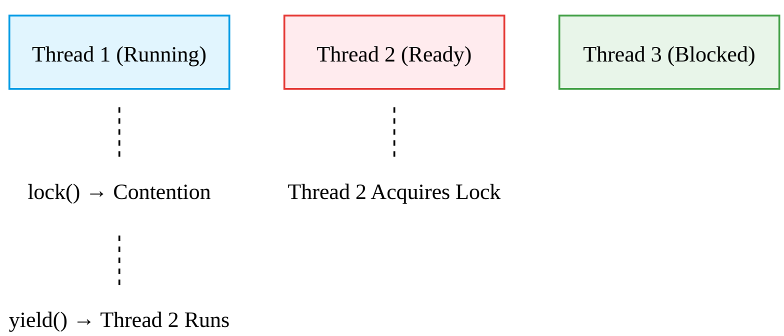
---

4. Avoiding Spinning: OS Support

4.1 Yield on Contention

```c
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        yield();  // Give up CPU
}
```

Problem: Still inefficient with many threads (O(N) context switches).



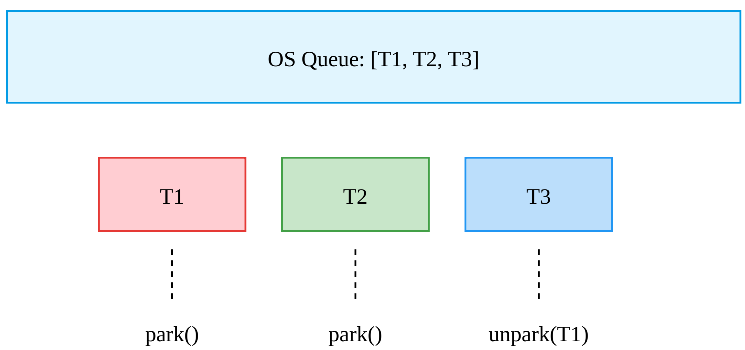
4.2 Queue-Based Locks (Linux `futex`, Solaris `park/unpark`)

```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock(lock_t *lock) {
    while (TestAndSet(&lock->guard, 1) == 1)
        ; // Acquire guard spinlock
    if (lock->flag == 0) {
```

```
        lock->flag = 1;  // Lock acquired
        lock->guard = 0;
    } else {
        queue_add(lock->q, gettid());
        lock->guard = 0;
        park();  // OS puts thread to sleep
    }
}
```

Advantage: No spinning, fair wakeup.



---

5. Advanced Lock Types

5.1 Ticket Lock (Fairness)

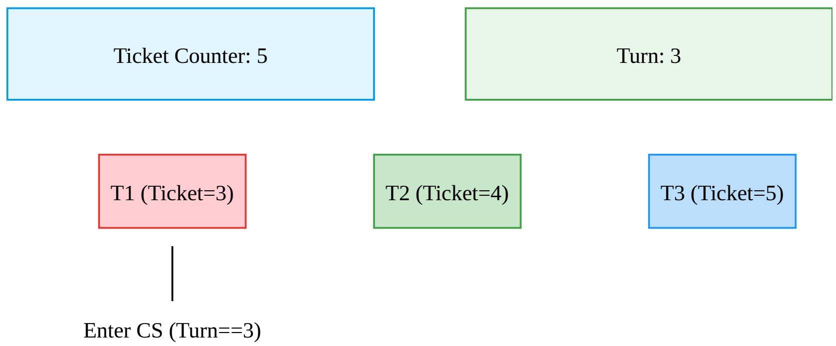
```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

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;
}
```

Guarantees: FIFO order, no starvation.



5.2 Two-Phase Locking (Hybrid Approach)

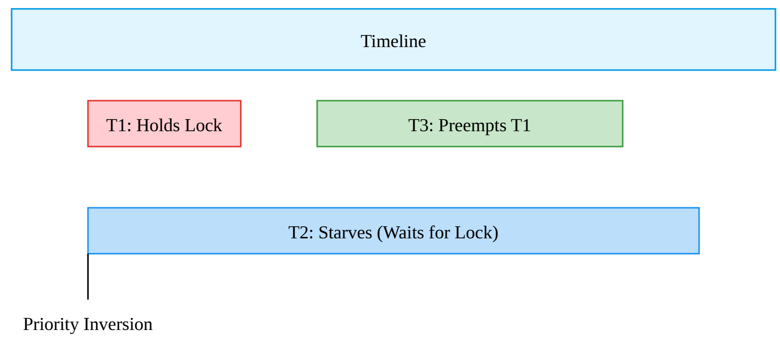
1. **Spin Phase**: Short spin hoping lock is released soon.
2. **Sleep Phase**: If spin fails, sleep via OS support.

---

6. Common Pitfalls

Priority Inversion

- **Scenario**: High-priority thread (T2) waits for low-priority thread (T1) holding a lock, but medium-priority thread (T3) preempts T1.
- **Solution**: Priority inheritance (temporarily boost T1's priority).



---

7. Summary of Lock Implementations

Lock Type	Mechanism	Pros	Cons
Test-and-Set	Hardware atomic instruction	Simple	Spins, wastes CPU
Ticket Lock	Fetch-and-Add	Fair, no starvation	Spins
Queue-Based Lock	OS-supported sleep/wakeup	No spinning, fair	Complex, syscall overhead
Two-Phase Lock	Hybrid spin + sleep	Balances spin/sleep tradeoffs	Tuning required

---

Homework Insights

1. **`flag.s`**: Simple flag-based lock fails under contention (race condition).
2. **`test-and-set.s`**: Uses `xchg` for atomic swap; spinning efficiency depends on interrupt frequency (`-i` flag).
3. **`Peterson.s`**: Software-only lock for 2 threads (no hardware support).
4. **`yield.s`**: Reduces CPU waste vs. pure spinning.