Image Source: Gemini

CSL 301

OPERATING SYSTEMS

Lecture 22

Thread-Locks

Instructor

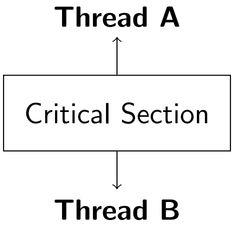
Dr. Dhiman Saha

The Problem of Concurrency

- Concurrent programs have multiple threads of execution.
- These threads often need to access shared resources (e.g., variables, data structures).
- Without proper synchronization, this can lead to race conditions, where the final result depends on the unpredictable timing of thread execution.

The Critical Section

A **critical section** is a piece of code that accesses a shared resource.



We need to ensure that only one thread can be inside the critical section at any given time. This is called **mutual exclusion**.

The Basic Lock

A lock is a simple synchronization primitive that provides mutual exclusion.

- A thread **acquires** the lock before entering a critical section.
- The thread **releases** the lock after leaving the critical section.

```
lock_t mutex;

lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Pthread Mutexes

The POSIX standard provides a lock implementation called a **mutex** (short for mutual exclusion).

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);
balance = balance + 1;
pthread_mutex_unlock(&lock);
```

How to Build a Lock

To build a lock, we need support from the hardware and the operating system.

- **Hardware Support:** Atomic instructions that can read and modify a memory location in a single, uninterruptible step.
- **OS Support:** Mechanisms for putting threads to sleep and waking them up.

Evaluating Locks

We can evaluate a lock implementation based on the following criteria:

- **Mutual Exclusion:** Does the lock prevent multiple threads from entering a critical section?
- **Fairness:** Does every thread that wants to acquire the lock eventually get it?
- **Performance:** What is the overhead of using the lock?

Controlling Interrupts

An early attempt at synchronization on single-processor systems was to disable interrupts.

```
void lock() {
  DisableInterrupts();
}

void unlock() {
  EnableInterrupts();
}
```

Problems:

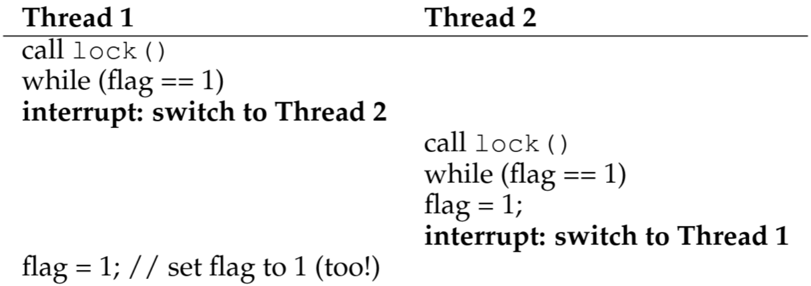
- Doesn't work on multiprocessor systems.
- Gives too much power to user threads.
- Can lead to lost interrupts.

A Failed Attempt: Just Using Loads/Stores

Let's try to build a lock using a simple flag variable.

```
typedef struct _lock_t { int flag; } lock_t;
void init(lock_t *mutex) {
 // 0 -> lock is available, 1 -> held
 mutex->flag = 0;
 }
void lock(lock_t *mutex) {
  while (mutex->flag == 1)
    ;
  mutex->flag = 1;
}
void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

Thread Trace - No Mutual Exclusion

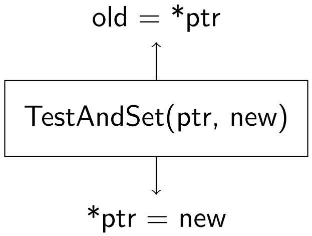


Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

- This approach fails because it's not atomic.
- Two threads can read the flag as 0 and then both set it to 1, violating mutual exclusion.

The Test-and-Set Instruction

The **test-and-set** instruction is an atomic operation that reads the value of a memory location and sets it to a new value in a single step.



Building a Spin Lock with Test-and-Set

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

void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

This is called a **spin lock** because the thread "spins" in a loop while waiting for the lock to be released.

The Problem with Spinning

Spinning can be very inefficient, especially on a single-processor system.

- The spinning thread wastes CPU cycles that could be used by another thread.
- If the thread holding the lock is preempted, the spinning thread will spin for the entire time slice.

Avoiding Spinning: Yielding

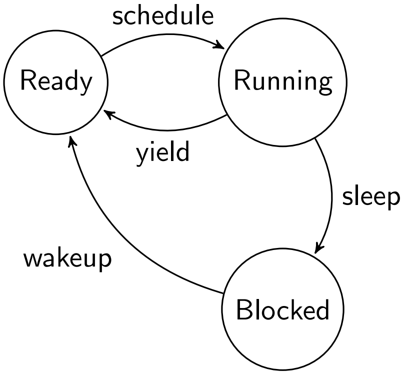
Instead of spinning, a thread can **yield** the CPU to another thread.

```
void lock() {
  while (TestAndSet(&flag, 1) == 1)
    yield(); // give up the CPU
}
```

This is better than spinning, but it's still not ideal. The yielding thread might be scheduled again before the lock is released.

Sleeping Instead of Spinning

A better approach is to put the waiting thread to **sleep** and wake it up when the lock is released.



This requires OS support (e.g., ‘park‘ and ‘unpark‘ in Solaris, ‘futex‘ in Linux).

Ticket Locks

A **ticket lock** is a fair lock that uses a ticket and turn variable.

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

void lock(lock_t *lock) {
  int myturn = FetchAndAdd(&lock->ticket);
  while (lock->turn != myturn)
    ;
}

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

This ensures that threads acquire the lock in the order they requested it.

Two-Phase Locks

A **two-phase lock** is a hybrid approach that combines spinning and sleeping.

- **Phase 1:** The lock spins for a short period of time.
- **Phase 2:** If the lock is not acquired, the thread is put to sleep.

This can be a good strategy if the lock is expected to be held for a short time.

Conclusion

- Locks are essential for writing correct concurrent programs.
- Building an efficient lock requires a combination of hardware and OS support.
- There is a trade-off between spinning and sleeping, and the best approach depends on the specific workload.