Lecture 8:
Concurrency Control:
Advanced Mutual Exclusion

Xin Liu

xl24j@fsu.edu

COP 4610 Operating Systems

Outline

- Fast/Slow Paths
- Mutex Locks
- Futex Locks

Recap: Spinlocks

- A spinlock is a simple lock where a thread constantly checks for lock availability.
- Threads on other processors are spinning idly while only one thread is in the critical section.

- Hardware instructions ensure atomic key exchange.
- Imagine a single key to a critical section. The first thread to acquire the key can enter.

Is it a spinlock?

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
long x = 0;

void *Tsum(void *arg) {
   for (int i = 0; i < N; i++) {
      asm volatile("lock addq $1, %0": "+m"(x));
   }
   return NULL;
}

int main() {
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, Tsum, NULL);      pthread_create(&thread2, NULL, Tsum, NULL);
   pthread_join(thread1, NULL);                     pthread_join(thread2, NULL);
   printf("x = %ld\n", x);
   return 0;
}
```

Is it a spinlock? (Cont.)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
long x = 0;

void *Tsum(void *arg) {
   for (int i = 0; i < N; i++) {
      asm volatile("lock addq $1, %0": "+m"(x));
   }
   return NULL;
}

int main() {
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, Tsum, NULL);        pthread_create(&thread2, NULL, Tsum, NULL);
   pthread_join(thread1, NULL);                       pthread_join(thread2, NULL);
   printf("x = %ld\n", x);
   return 0;
}
```

Not a spinlock: There is no behavior of repeatedly checking and waiting for the lock to be released.
It is an atomic operation: The lock prefix instruction is used to implement a thread-safe atomic addition operation.

Let’s create a spinlock

```
void *Txpp(void *arg) {
  for (int i = 0; i < N; i++) {
    acquire_spin_lock(&lock);
    x++;
    release_spin_lock(&lock);
  }
  return NULL;
}
```

x++: Load -> Exec -> Store

```
int main() {
  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL, Txpp, NULL);      pthread_create(&thread2, NULL, Txpp, NULL);

  pthread_join(thread1, NULL);                     pthread_join(thread2, NULL);

  printf("x = %ld\n", x);
  return 0;
}
```

Let’s create a spinlock (Cont.)

```
int xchg(int *addr, int newval) {
   int result;
   asm volatile (
      "lock xchg %0, %1"
      : "+m" (*addr), "=a" (result)
      : "1" (newval)
      : "cc"
   );
   return result;
}
```

Example :

- Assume:
  - The initial value of *addr is 5, meaning *addr = 5.
  - The new value newval is 10.
- After the xchg function is executed:
  - The value at *addr becomes 10, which is the value of newval.
  - The function returns 5, which was the original value of *addr before the swap.

Let’s create a spinlock (Cont.)

```
// Acquire the lock using xchg (spinlock)
void acquire_lock(int *lock) {
// Please complete the code
}

// Release the lock
void release_lock(int *lock) {
// Please complete the code
}
```

Recap: Drawbacks of Spinlocks

- Threads on other processors are spinning idly while only one thread is in the critical section.
  - The more processors competing for the lock, the lower the efficiency.
  - Example:

https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_6_spin_scalability.c

Dio Brando's Stand, The World, is like a thread that always acquires the spinlock, pausing all other threads and controlling the lock (time).

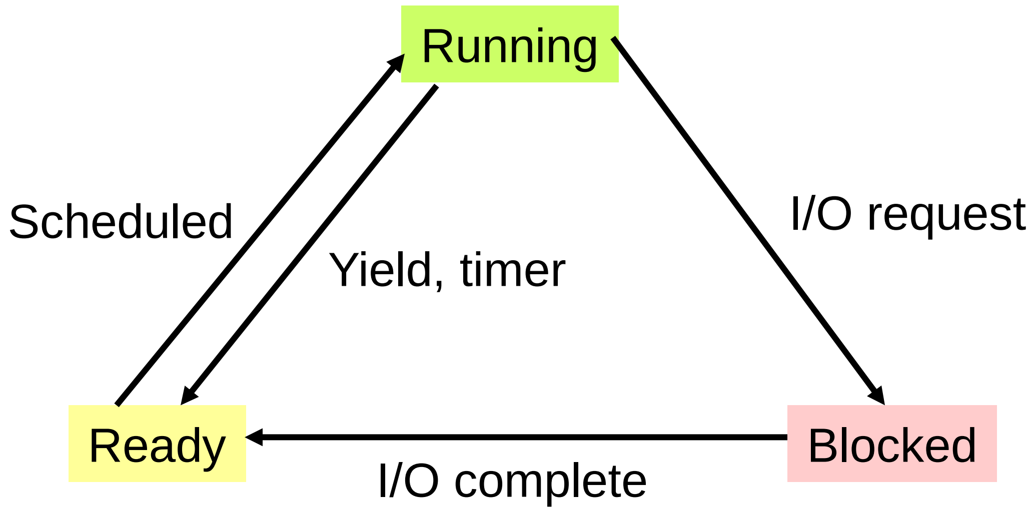
Recap: Drawbacks of Spinlocks (Cont.)

- The thread holding the spinlock might be switched out by the operating system.
  - The OS is unaware of the thread's activity (but why can't it be?).

Thread States and Spinlock Behavior

- The thread holding the spinlock might be switched out by the operating system.
  - The OS is unaware of the thread's activity (but why can't it be?).
  - Each thread can be in one of the three states

1. ***Running***: has the CPU
2. ***Blocked***: waiting for I/O or another thread
3. ***Ready to run***: on the ready list, waiting for the CPU



Thread States and Spinlock Behavior (Cont.)

- A thread waiting for a spinlock is busy-waiting (spinning), constantly checking the lock status.
- From the OS perspective, this thread is **Running** (actively using CPU) even though it's not doing useful work.
- If the thread's time slice runs out, it is moved to the **Ready to Run** state and will be rescheduled.

- The OS doesn't know the thread is busy-waiting for the lock. It treats the thread like any other process that's actively using the CPU.

- This leads to 100% resource waste…

Recap: Use Cases for Spinlocks

- **Low Competition:**
  - The critical section is rarely “contended”.
  - For example, suppose only two threads occasionally access the same global variable for a very short time. A spinlock can quickly complete the lock acquisition and release, with almost no chance of both threads competing for the lock simultaneously.

- **Short Time:**
  - Short critical section: uses just a few instructions to access shared resources.
  - During this time, a lock is held to prevent data races, but since the operation is so quick, the lock is only needed for a short duration.

- **No Switch-Out:**
  - The operating system does not perform a context switch on that thread.
  - The operating system can disable interrupts and preemption, ensuring that the lock holder can release the lock in a very short time.

Mutex

- Problem:
  - When a thread is waiting for a lock, it wastes CPU cycles by waiting idly.
  - Why not let other threads use the CPU instead of busy waiting?

- Solution:
  - What is the key to the solution?

Mutex

- Problem:
  - When a thread is waiting for a lock, it wastes CPU cycles by waiting idly.
  - Why not let other threads use the CPU instead of busy waiting?

- Solution:
  - When a lock is unavailable, the thread is blocked.
  - When the lock is available, the thread is awakened, saving CPU time.

Mutex

- Problem:
  - When a thread is waiting for a lock, it wastes CPU cycles by waiting idly.
  - Why not let other threads use the CPU instead of busy waiting?

- Solution:
  - When a lock is unavailable, the thread is blocked.
  - When the lock is available, the thread is awakened, saving CPU time.

- However, blocking a thread cannot be directly handled by user-space C code, which only performs computations.

- We need to use **system calls** to interact with the operating system.

Understanding Mutex with a Library Key Analogy

Operating System = Library Desk Manager
Critical Section = Study Room
Lock = Key to Study Room

1. The first person (Thread 1) arrives and requests the key:

   - Thread 1 makes a system call to request the lock (key). If the lock is available, the operating system (the desk manager) immediately gives the key to Thread 1.

   - The system call returns immediately, indicating that Thread 1 successfully acquired the lock (key) and can enter the critical section (study room) to perform its task.

   - Illustration: *lk = 🔒 (lock acquired, system call completed).

Understanding Mutex with an Analogy (Cont.)

2. The second person (Thread 2) arrives but the lock is occupied:

- Thread 2 makes a system call to request the lock, but since Thread 1 already holds the lock (the key is taken), Thread 2 cannot enter.
- The operating system puts Thread 2 in a wait queue, and the system call does not return immediately. Thread 2 is **blocked**, waiting for the lock to be released.
- Illustration: Thread 2 is added to the wait queue, waiting for the lock to be released.

Understanding Mutex with an Analogy (Cont.)

3. The first person (Thread 1) finishes and releases the lock:

- When Thread 1 finishes its task, it makes a system call to release the lock (return the key).
- The operating system checks the wait queue and sees that Thread 2 is waiting for the lock.
- The operating system hands the lock to Thread 2, and the system call returns, **waking up** Thread 2, which now acquires the lock and continues its task.

Understanding Mutex with an Analogy (Cont.)

4. If no one is waiting:

- If there are no other threads waiting for the lock, the operating system marks the lock as available.
- Illustration: *lk = ✅ (lock released).

5. We still need spinlocks
   - When multiple threads request a mutex via system calls, the OS must manage access to the lock.
   - During lock allocation or release, spinlocks are used to ensure these operations are atomic, preventing multiple threads from changing the lock’s state simultaneously.
   - Spinlocks protect this critical section for a brief moment in the kernel, and are released immediately after the lock’s state is updated, minimizing CPU usage.
     - A classic spinlock use case: Short Critical Section

Mutex Implementation

- syscall-Based Implementation
  - Direct interaction with the kernel.
  - Uses system calls to request and release locks.
    - Example:

```
syscall(SYSCALL_lock, &lk);
syscall(SYSCALL_unlock, &lk);
```

  - Advantages:
    - Provides low-level control over locking mechanisms.
    - Useful in OS development or scenarios requiring custom lock behavior.
  - Disadvantages:
    - More complex and involves direct kernel interaction.
    - Less portable between different systems.

Mutex Implementation (Cont.)

- pthread-Based Implementation
  - High-level abstraction with POSIX threads.
  - Uses POSIX thread library for lock management.
    - Example:

```
pthread_mutex_lock(&lk);
pthread_mutex_unlock(&lk);
```

  https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_6_spin_scalability.c

  - Advantages:
    - Simplifies locking operations with built-in abstractions.
    - Portable across various platforms that support POSIX threads.
    - More suitable for user-space applications.
  - Disadvantages:
    - Less control over the underlying system calls.
    - Slightly more overhead due to the abstraction layer.

Fast / Slow Path

- In computer systems, **fast path** and **slow path** represent two ways of handling tasks:

  - **Fast Path**: The most efficient execution route, typically handling common cases with minimal overhead.

  - **Slow Path**: A less efficient, resource-intensive route for handling rare or complex cases.

Some Analysis on Mutual Exclusion

- Spinlock (threads directly share the locked variable)
  - Faster fast path
    - xchg succeeds → Immediately enters the critical section, with minimal overhead.
  - Slower slow path
    - xchg fails → Wastes CPU cycles by spinning in a loop, waiting.

- Mutex (accesses locked via system calls)
  - Faster slow path
    - When locking fails, the thread does not occupy the CPU (it goes to sleep).
  - Slower fast path
    - Even if locking succeeds, entering and exiting the kernel (syscall) adds overhead.

Futex: Fast Userspace muTexes

- Why choose when you can have both?
  - Fast path: A single atomic instruction; if locking succeeds, it returns immediately.
  - Slow path: If locking fails, the thread calls a system call to sleep.

- Futex = Spin + Mutex

- Common performance optimization technique:
  - Focus on the average (frequent) case, not the worst case.
  - So, ….

The Mutex in POSIX Threads Library (pthread_mutex) is Futex!

Let's take a look

- Example:
  https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_6_spin_scalability.c

- Monitoring system calls
  - strace –fc ./your_program

Takeaways

- Q: How do we achieve mutual exclusion on multiprocessor systems?
  - Don’t fear race conditions, solve them with software (Peterson’s algorithm).
  - If software isn’t enough, use hardware (spinlocks).
  - If userspace isn’t enough, rely on the kernel (mutexes).
  - Identify the assumptions you're relying on, and break them when necessary.
- Fast/slow paths: An important technique for performance optimization.