Operating Systems

Synchronization

Concurrency Example

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
volatile int counter = 0;
int loops;
void *worker(void *arg) {
        int i;
        for (i = 0; i < loops; i++) {
                counter++;
        }
        return NULL;
}
int
main(int argc, char *argv[])
{
        loops = atoi(argv[1]);
        pthread_t p1, p2;
        printf("Initial value : %d\n", counter);
        Pthread_create(&p1, NULL, worker, NULL);
        Pthread_create(&p2, NULL, worker, NULL);
        Pthread_join(p1, NULL);
        Pthread_join(p2, NULL);
        printf("Final value : %d\n", counter);
        return 0;
}
```

Concurrency Example (Cont.)

- `loops` determines how many times each of the two workers will **increment the shared counter** in a loop.
  - `loops`: 1000.

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

  - `loops`: 100000.

```
prompt> ./thread 100000
Initial value : 0
Final value : 143012 // huh??
prompt> ./thread 100000
Initial value : 0
Final value : 137298 // what the??
```

Why is this happening?

- Increment a shared counter → take three instructions.
  1. Load the value of the counter from memory into register.
  2. Increment it
  3. Store it back into memory

counter = counter + 1

```
105     mov 0x8049a1c, %eax
108     add $0x1, %eax
113     mov %eax, 0x8049a1c
```

Race condition

- Example with two threads
  - counter = counter + 1 (default is 50)
  - We expect the result is 52. However,

```
105    mov 0x8049a1c, %eax
108    add $0x1, %eax
113    mov %eax, 0x8049a1c
```

OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	before critical section		100	0	50
	`mov 0x8049a1c, %eax`		105	50	50
	`add $0x1, %eax`		108	51	50
`interrupt`					
`save T1's state`					
`restore T2's state`			100	0	50
		`mov 0x8049a1c, %eax`	105	50	50
		`add $0x1, %eax`	108	51	50
		`mov %eax, 0x8049a1c`	113	51	51
`interrupt`					
`save T2's state`					
`restore T1's state`			108	51	50
	`mov %eax, 0x8049a1c`		113	$\boxed{51}$	**51**

A few terminologies

- Race condition:
  - the results depend on the timing execution of the code.
  - Result is indeterminate.
- Critical section
  - A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity** for critical sections (**mutual exclusion**)
  - This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

The wish for atomicity

- Ideal approach; make the increment as a single assembly instruction

```
memory-add 0x8049a1c, $0x1
```

- Atomically, in this context, means "as a unit", which sometimes we take as "all or none."
- In general, we do not have such instruction. Instead, we use lock.
- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

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

Critical section

Locks

Locks: The Basic Idea

- Ensure that any **critical section** executes as if it were a single atomic instruction.
  - An example: the canonical update of a shared variable

```
balance = balance + 1;
```

  - Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks: The Basic Idea

- Lock variable holds <u>the state of </u>the lock.
  - **available** (or **unlocked** or **free**)
    - No thread holds the lock.
  - **acquired** (or **locked** or **held**)
    - Exactly one thread holds the lock and presumably is in a critical section.

The semantics of the lock()

- `lock()`
  - **Try to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock.
  - **Enter** the *critical section*.
    - This thread is said to be the owner of the lock.
  - Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

Pthread Locks - mutex

- The name that the POSIX library uses for a lock.
  - Used to provide mutual exclusion between threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

  - We may be using *different locks* to protect *different variables* → Increase concurrency (a more **fine-grained** approach).

Building A Lock

- Efficient locks provided mutual exclusion at low cost.
- Building a lock need some help from the **hardware** and the **OS**.

Evaluating locks – Basic criteria

- **Mutual exclusion**
  - Does the lock work, preventing multiple threads from entering *a critical section*?
- **Fairness**
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)
- **Performance**
  - The time overheads added by using the lock
    - No contention
    - Contention of multiple threads on a single-core CPU
    - Contention multiple threads on multiple CPUs

Controlling Interrupts

- **Disable Interrupts** for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems.

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

  - Problem:
    - Require too much *trust* in applications
      - Greedy (or malicious) program could monopolize the processor.
    - Do not work on multiprocessors.
    - Code that masks or unmasks interrupts be executed *slowly* by modern CPUs.

Why hardware support needed?

- **First attempt**: Using a *flag* denoting whether the lock is held or not.
  - The code below has problems.

```
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)  // TEST the flag
            ;  // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it !
}

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

Why hardware support needed? (Cont.)

- **Problem 1**: No Mutual Exclusion (assume `flag=0` to begin)

Thread1	Thread2
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!)	

- **Problem 2**: Spin-waiting wastes time waiting for another thread.

- So, we need an atomic instruction supported by Hardware!
  - *test-and-set* instruction, also known as *atomic exchange*

Test And Set (Atomic Exchange)

- An atomic instruction to support the creation of simple locks

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;   // fetch old value at ptr
    *ptr = new;       // store 'new' into ptr
    return old;       // return the old value
}
```

- **return**(testing) old value pointed to by the `ptr`.
- *Simultaneously* **update**(setting) said value to `new`.
- This sequence of operations is performed atomically.

A Simple Spin Lock using test-and-set

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

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

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

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

- **Note**: To work correctly on *a single processor*, it requires <u>a preemptive scheduler</u>.

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to entry the critical section.
- **Fairness**: no
  - Spin locks <u>don't provide any fairness</u> guarantees.
  - Indeed, a thread spinning may spin *forever*.
- **Performance**:
  - In the single CPU, performance overheads can be quite *painful*.
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

Fetch-And-Add

- Atomically increment a value while returning the old value at a particular address.

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

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock

- **Ticket lock** can be built with <u>fetch-and add</u>.
  - Ensure progress for all threads. → fairness

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

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

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

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

So Much Spinning

- Hardware-based spin locks are simple and they work.

- In some cases, these solutions can be quite inefficient.
  - Any time a thread gets caught *spinning*, it **wastes an entire time slice** doing nothing but checking a value.

How To Avoid *Spinning*?
We'll need OS Support too!

Using Queues: Sleeping Instead of Spinning

- **Queue** to keep track of which threads are waiting to enter the lock.
- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`.

Using Queues: Sleeping Instead of Spinning

```
 typedef struct __lock_t {
    int flag;    // lock is acquired or not
    int guard;   // to protect the queue
    queue_t *q;
} lock_t;
```

Using Queues: Sleeping Instead of Spinning

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

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
…
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues: Sleeping Instead of Spinning

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
        m->guard = 0;
}
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Locks APIs

- Provide mutual exclusion to a critical section
  - Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - No other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Locks APIs (Cont.)

- All locks must be properly initialized.
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks APIs (Cont.)

- Check errors code when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- These two calls are used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - trylock: return failure if the lock is already held
  - timelock: return after a timeout

Locks APIs (Cont.)

- These two calls are also used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - `trylock`: return failure if the lock is already held
  - `timelock`: return after a timeout or after acquiring the lock