Locks

- Lock is either taken or free (binary)
- Initially free
- Only one thread can hold the lock at the same time
- Needs to be explicitly released

Naive Lock Implementation

- Does it work?
- Is it a good idea?
- Thread is no longer interruptible, could loop forever and starve other threads
- Critical sections could be long, cannot wait too long to handle interrupts
- This solution defeats the scheduler

Towards Mutual Exclusion

- Key observations:
  - On a uniprocessor, an operation is atomic if no context switch occurs in the middle of the operation
  - Context switches occur because of:
    - Internal events: system calls and exceptions
    - External events: interrupts
  - Mutual exclusion can be achieved by preventing context switch

- Prevention of context switch
  - Eliminate internal events:
    - easy (under program control)
  - Eliminate external events:
    - disable interrupts
    - Hardware delays the processing of interrupts until interrupts are enabled
Better Implementation

```c
typedef uintptr_t lock_t;

void lock_acquire(lock_t *lock) {
    disable_interrupts();
    while (*lock) {
        enable_interrupts();
        disable_interrupts();
    }
    *lock = 1;
    enable_interrupts();
}

void lock_release(lock_t *lock) {
    disable_interrupts();
    *lock = 0;
    enable_interrupts();
}
```

- Works on a uniprocessor system
- What happens on a multiprocessor system?

Atomicity

- We can reduce the problem of mutual exclusion to an atomic update of a single memory location
- Atomic: all or nothing
- How?
  - Modern CPUs: Hardware support!
  - Read-Modify-Write
  - Interaction with cache coherence protocol

Atomic Operations

- **test_and_set**
  - Set a variable and return its old value
  - `old_value = test_and_set(var, new_value)`
  - xchg on x86
  - swp on ARM

- **compare_and_swap**
  - Set a variable only if its current value equals some assumed value
  - `old_value = compare_and_swap(var, assumed_value, new_value)`
  - cmpxchg on x86

- **Load-link/Store-Conditional**
  - Load a variable into register for modification, safe PC of the instruction
  - Track modifications
  - Later: if variable was concurrently modified abort store and jump back to saved PC
  - lwarz, stawz on PowerPC
  - ll/sc on MIPS
  - ldrex, strex on newer ARM (>ARMv6)
Lock with Test and Set

```c
void lock_acquire(lock_t *lock) {
    while (test_and_set(lock)) {
    }
}

void lock_release(lock_t *lock) {
    lock=0;
}
```

**x86:**

```c
_asm__volatile__({
    " lock \
    " xchg %0,%1 \
    : "q"(_res), "m"("Lock")
    : "0"(_res));

return (int) _res;
}
```

**ARM:**

```c
>= ARMv6
ldrex
strex
```

Before:

```c
swp
```

Minimizing Busy Waiting

```c
void lock_acquire(lock_t *lock) {
    while (test_and_set(lock->guard)) {
        if (lock->value) {
            // put TCB on wait queue for lock;
            // set guard = 0 and call switch();
        } else {
            lock->value = 1; lock->guard = 0;
        }
    }
}

void lock_release(lock_t *lock) {
    while (test_and_set(lock->guard)) {
        if (wait_queue_not_empty()) {
            // move a waiting thread to ready queue;
        } else {
            value = 0;
        }
        lock->guard = lock->value = 0;
    }
```

Mutex

```c
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock (&mutex);
pthread_mutex_unlock (&mutex);
```
**Problem: Lock Contention**

- Tasks are contending for the same lock
  - Tradeoff: spin lock is better when wait time is expected to be small and contention is low; mutex better for contended case and when wait time is expected to be longer
- What happens with contended spin locks?
  - Many tasks are performing test_and_set-like operations on a lock variable
  - High traffic on the memory bus
  - Cache lines are bouncing between CPUs
  - Not fair
  - Even more a problem with NUMA

**Lamport’s Bakery Algorithm**

```c
void enter(int i)
{
    choosing[i] = 1;
    ticket[i] = max(ticket[0], …, ticket[N-1])+1;
    choosing[i] = 0;
    for(int j = 0; j < N; j++)
    {
        while(choosing[j]) continue;
        while(ticket[j] && (ticket[j] < ticket[i])) continue;
    }
}
void exit(int i)
{
    ticket[i] = 0;
}
```

**Ticket Lock**

- Idea: split lock value into two pieces:
  - Next, Owner
- Do a compare and swap to get current lock value and increment Next.
  - If (old) Next was == Owner, we have the lock
  - Else, spin.
- Releasing the lock = incrementing Owner.

**Problem: Priority Inversion**

- A wants L, blocks
- High wants L, blocks
- Medium
- Low
- A is blocked by B ∧ C is blocked by A → C is blocked by B
Problem: Priority Inversion

- **Ingredients:**
  - Three tasks: A, B, C
  - Priorities: P(A)<P(B)<P(C)
  - A lock for which A and C contend

- **Problem:**
  - A gets scheduled and acquires lock L
  - C gets scheduled, wants lock L and blocks
  - B gets scheduled, preempts A
  - B runs for a long time
  - A is blocked by B, C is blocked by A → C is blocked by B
  - Priority inversion!

Priority Inversion - Solutions

- **Non Preemptive Protocol**
  - Task which succeeds in acquiring the lock gets highest priority

- **Priority Inheritance Protocol**
  - Task with lock temporarily inherits the priority of the highest waiting task

- **Random Boosting (Windows)**
  - Task with lock get their priority randomly boosted until it releases the lock

Aside: Locking Without Cache Coherence

- **Message Passing Buffer**
  - How to synchronize on it?
  - Single lock bit per core

Producer/Consumer Queue

- **Producer = Thread, Consumer = Thread**
- **Shared Queue**
  - enqueue(item), item = dequeue()
- **Concurrent + Shared = needs protection**
- **What happens to producer if queue is full?**
- **What happens to consumer if queue is empty?**
  
We would like to have some form of event notification.
Condition Variables

- The basic idea:
  ```
  acquire_lock(&lock);
  if(is_full(queue)) {
    wait_for_space(queue);
  }
  add_item(queue, item);
  release_lock(&lock);
  ```

- Wait sends thread to sleep until the producer wakes it up
- What happens to the lock?
  - Must be released right before wait
  - Must be reacquired right after wait

- ** POSIX Condition Variable**
  - Wait for some event to happen
  - Simple inter-thread communication schema
  - Integrated with mutex
    ```
    pthread_mutex_t mutex;
    pthread_cond_t cv;
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cv, &mutex);
    ...do stuff...
    pthread_mutex_unlock(&mutex);
    ```

Producer/Consumer with pthread_cond

- Wait, something is broken...
- Condition needs to be checked from a loop
- Who wakes up after a signal?

- ** Other thread:**
  ```
  pthread_mutex_lock(&mutex);
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&mutex);
  ```
Producer/Consumer with pthread_cond V2.0

- Possible solution: broadcast

```c
pthread_mutex_t lock;
pthread_cond_t cv;

Producer
  pthread_mutex_lock(&lock);
  while (is_full(queue)) {
    pthread_cond_wait(&cv, &lock);
  }
  enqueue(queue, item);
  pthread_cond_broadcast(&cv);
  pthread_mutex_unlock(&lock);

Consumer
  pthread_mutex_lock(&lock);
  while (is_empty(queue)) {
    pthread_cond_wait(&cv, &lock);
  }
  dequeue(queue, &item);
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
```

- Awakes “thundering herd”...

Monitors

- Introduced as programming language constructs
  - Mesa, Java, C#, ...
- OOP style
  - Protection around objects
- Can be turned into a programming convention
  - Use the idea behind monitors in C, C++

- What are monitors?
  - Separate concern of mutual exclusion and conditional synchronization
    - Lock
      - Mutual Exclusion, monitor entry and exit
    - Zero or more condition variables
      - Cooperates with the lock (like POSIX pthread_cond)

Producer/Consumer with pthread_cond V3.0

```c
pthread_mutex_t lock;
pthread_cond_t not_full; pthread_cond_t not_empty;

Producer
  pthread_mutex_lock(&lock);
  while (is_full(queue)) {
    pthread_cond_wait(&not_full, &lock);
  }
  enqueue(queue, item);
  pthread_cond_signal(&not_empty);
  pthread_mutex_unlock(&lock);

Consumer
  pthread_mutex_lock(&lock);
  while (is_empty(queue)) {
    pthread_cond_wait(&not_empty, &lock);
  }
  item = queue.dequeue();
  pthread_cond_signal(&not_full);
  pthread_mutex_unlock(&lock);
```

Producer/Consumer with Monitor in Java

```java
class Queue {
  ...
}
Queue queue = new Queue();

Producer
  synchronized(queue) {
    while (queue.size() == MAX) {
      queue.wait();
    }
    queue.enqueue(item);
    queue.notifyAll();
  }

Consumer
  synchronized(queue) {
    while (queue.size() == 0) {
      queue.wait();
    }
    item = queue.dequeue();
    queue.notifyAll();
  }
```
Semantics of Signal

- **Hansen Style**
  - Signaling thread keeps the lock
  - Waiting thread waits for the lock
  - Signal is only a hint
  - Condition might have changed

- **Hoare Style**
  - Signaling thread gives up the lock
  - Waiting thread gets the lock
  - Signal is atomic
  - Condition cannot change before waiting thread is resumed

Producer/Consumer Queue

- Producer = Thread, Consumer = Thread
- We used two events:
  - Queue Full
  - Queue Empty

- Problem: condition variables keep no state.
  - Compare to signals...
  - What if we had state?

Semaphores

- Edsger Dijkstra: Semaphore
- Non-negative integer variable with two atomic operations
  - `P()`
    - 
    - Otherwise “wait” until `sem > 0` and then decrement
  - `V()`
    - `sem < 0` and then decrement
    - Increment `sem by 1`
    - Wake up a thread waiting in `P()`

- Can be used for mutual exclusion (binary semaphore) and conditional synchronization
System V Semaphores

- `int semget(key_t key, int nsems, int semflg);`
  - Creates a semaphore with up to 25 values
- `int semctl(int semid, int semnum, int cmd, union semun arg);`
  - Manipulates the state of the semaphore, e.g., GETVAL, SETVAL

Properties
- Can increment/decrement by any value
- Behave similar to a file, can have permissions
- System-wide

POSIX Semaphores

- `sem_t sem;`
  - Define a semaphore
- `sem_init(&sem, pshared, value);`
  - Initialize a semaphore with a value range
- `sem_post(&sem);`
  - Increments and wakes up any thread blocked waiting on the semaphore
- `sem_wait(&sem);`
  - If semaphore value is negative, blocks. Else, decrement.
- `sem_getvalue(&sem, &value);`
  - Get the current value
- `sem_destroy(sem_t*sem);`
  - Destroy the semaphore. Only succeeds if no thread is blocked.

Semaphores Summary

- Semaphores combine mutual exclusion and waiting for condition
  - Separation of concern?
  - Proper use can be difficult in practice

- System V Semaphores are rarely used
  - Invented as an IPC mechanism
  - Unintuitive API, difficult to use
  - Heavyweight

- POSIX Semaphores are more lightweight

Futex

- Kernel-level locking is fair but expensive
  - Kernel gives out an opaque handle to a kernel-level entity
  - System call required for every operation

- User-level locking is faster but often unfair
  - Why?

- What if a lock is available most of the time?
  - Cost of checking

  - Idea: export a kernel-level data structure into user space
    - Combine the advantages of user-level with those of kernel-level locking
    - Make checking cheap
Futex

- Integer accessible to both kernel and user space
  - Three states:
    - 0: unlocked
    - 1: locked
    - 2: locked and contended
  - A kernel-level wait queue

- Lock based on Futex
  - Start as spinlock
    - for loop: compare_and_swap(v, 0, 1)
  - If not successful, turn into wait lock
    - set v to 2 and add itself to wait queue

Performance Aspects of Concurrency

- Sequential

- Parallel

- Speedup
  - \( S = \frac{T_1}{T_P} \)

Amdahl's Law

- \( S(n) = \frac{1}{(1-P)+\frac{P}{n}} \)

Lock-Free Data Structures

- Locking entire sections of the code is problematic
  - Hinders parallelism
  - Priority inversion
  - Preemption

- Designing lock-free algorithms is hard
  - Idea: design lock-free data structures
Example: Lock-Free Linked List

- CAS for changing pointers
- Delete: logical delete, then physical delete