## Hardware Primitives

- Many modern operating systems provide special synchronization hardware to provide more powerful **atomic operations**
  - `testAndSet( lock )`  
    - atomically reads the original value of lock and then sets it to true.
  - `Swap( a, b )`  
    - atomically swaps the values
  - `compareAndSwap( a, b )`  
    - atomically swaps the original value of lock and sets it to true when they values are different
  - `fetchAndAdd( x, n )`  
    - atomically reads the original value of x and adds n to it.

## Hardware: `testAndSet()`

```java
boolean testAndSet( boolean lock )
{
    boolean old-lock = lock;
    // initialization
    lock = false;  // shared -- lock is available
    void deposit( int amount )
    {
        // entry to critical section - get the lock
        while( testAndSet( theLock ) == true ) {};
        balance += amount;  // critical section
        // exit critical section - release the lock
        lock = false;
    }
    return old = old-lock;
}
```

- **Atomicity guaranteed - even on multiprocessors**

## Hardware: `compareAndSwap()`

```java
void Swap( boolean &a, boolean &b )  
{
    boolean temp = *a;
    *a = *b;  
    *b = temp;
}
```

- **Atomicity guaranteed - even on multiprocessors**

## Hardware with Bounded Waiting

- Need to create the illusion of a waiting line!
- **Idea:** Dressing Room is the critical section, only one person can be in room at one time, and one waiting line first come first serve outside the dressing room
  - `waiting[n]`: Global shared variable
  - `lock`: Global shared variable
- Entry get a local variable ‘key’ and check via `testAndSet()` if someone is ‘in’ the dressing room
  - `No!`
Hardware with Bounded Waiting

```c
// Initialization
lock = false; // shared -- lock is available
waiting[0..n-1] = (false); // shared -- no one is waiting
void deposit(int amount)
{
    pthread_mutex_lock(&locks[account_tid]);
    balance += amount; // critical section
    pthread_mutex_unlock(&locks[account_tid]);
}

// entry to critical section
waiting[tid] = true; // signal tid is waiting
key = true; // local variable
while( waiting[tid] == true ) and ( key == true )
    key = testAndSet( lock );
waiting[tid] = false; // got lock done waiting
balance += amount; // critical section

// exit critical section - release the lock
j = (j+1) mod n; // j is possibly waiting next in line
while( j != i ) and ( waiting[j] ) == false )
j = (j+1) mod n; // check next if waiting
if( j == i ) // no one is waiting unlock room
    lock = false;
else
    waiting[j] = false // hand over the key to j
}
```

Hardware Solution: Proof Outline

- **Mutual Exclusion:**
  - A thread enters only if it is waiting or if the dressing room is unlocked
    - First thread to execute `testAndSet( lock )` gets the lock all others will wait
    - Waiting becomes false only if the thread with the lock leaves its CS and only one waiting is set to false.

- **Progress:**
  - Since an exiting thread either unlocks the dressing room or hands the ‘lock’ to another thread progress is guaranteed because both allow a waiting thread access to the dressing room

- **Bounded Waiting:**
  - Leaving threads scans the waiting array in cyclic order thus any waiting thread enters the critical section within n-1 turns.

Synchronization Layering

- Build higher-level synchronization primitives in OS
  - Operations that ensure correct ordering of instructions across threads
- Motivation: Build them once and get them right
  - Don’t make users write entry and exit code

Locks

- **Goal:** Provide mutual exclusion (mutex)
- **Three common operations:**
  - **Allocate and Initialize**
    ```c
    pthread_mutex_t mylock;
    mylock = PTHREAD_MUTEX_INITIALIZER;
    ```
  - **Acquire**
    ```c
    Acquire exclusion access to lock; Wait if lock is not available
    pthread_mutex_lock(&mylock);
    ```
  - **Release**
    ```c
    Release exclusive access to lock
    pthread_mutex_unlock(&mylock);
    ```

Lock Examples

- **After lock has been allocated and initialized**

```c
void deposit(int amount)
{
    pthread_mutex_lock(&my_lock);
    balance += amount; // critical section
    pthread_mutex_unlock(&my_lock);
}
```

- **One lock for each bank account:**

```c
void deposit(int account_tid, int amount)
{
    pthread_mutex_lock(&locks[account_tid]);
    balance[account_tid] += amount; // critical section
    pthread_mutex_unlock(&locks[account_tid]);
}
```

- **Disadvantage:** Two threads only

Implementing Locks: Atomic loads and stores

```c
typedef struct lock_s
{bool lock[2] = (false, false);
int turn = 0;
}
void acquire(lock_s *lock)
{lock->lock[tid] = true;
 turn = 1-tid;
while(lock->lock[tid] & !lock->turn == 1-tid)
}
void release(lock_s lock)
{lock->lock[tid] = false;
```
Implementing Locks: Disable/Enable Interrupts

```c
void acquire( lock_s *lock )
    disableInterrupts();
void release( lock_s lock )
    enableInterrupts();
```

- Advantage: Supports mutual exclusion for many threads (prevents context switches)
- Disadvantages:
  - Not supported on multiple processors,
  - Too much power given to a thread
  - May miss or delay important events

Implementing Locks: Hardware Instructions

```c
typedef boolean lock_s;
void acquire( lock_s *lock )
    while ( true == testAndSet( theLock ) ) {} // wait
void release( lock_s lock )
    lock = false;
```

- Advantage: Supported on multiple processors
- Disadvantages:
  - Spinning on a lock may waste CPU cycles
  - The longer the CS the longer the spin
    - Greater chance for lock holder to be interrupted too

Spin Locks and Disabling Interrupts

- Spin locks and disabling interrupts are useful only for very short and simple critical sections
  - Wasteful otherwise
  - These primitives are primitive -- don’t do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section
  - All synchronization requires atomicity
  - So we’ll use our “atomic” locks as primitives to implement them

Semaphores

- Semaphores are another data structure that provides mutual exclusion to critical sections
  - Block waiters, interrupts enabled within CS
  - Described by Dijkstra in the THE system in 1968
- Semaphores have two purposes
  - Mutual Exclusion: Ensure threads don’t access critical section at same time
  - Scheduling constraints: Ensure threads execute in specific order (waiting queue).

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes
- `wait()` tests the semaphore
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- `signal()` opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
  - `signal()` has history
  - This ‘history’ is a counter

Semaphore Operations

- Allocate and Initialize
  - Semaphore contains a non-negative integer value
  - User cannot read or write value directly after initialization
    - `sem_t sem;
      - int semInit( &sem, is_shared, init_value );`
- Wait or Test
  - `P()` for “test” in Dutch (proberen) also `down()`
  - Waits until semaphore is open (`sem>0`) then decrement `sem`
    - `int semWait( &sem );`
- Signal or Increment or Post
  - `V()` for “increment” in Dutch (verhogen) also `up()`, `signal()`
  - Increments value of semaphore, allow another thread to enter
    - `int semPost( &sem );`
Semaphore Implementation

typedef struct {
    int value;
    queue tlist;
} semaphore;

sem_wait( semaphore *S ) // Must be executed atomically
S->value--;
if (S->value < 0)
    add this process to S->tlist;
    block();

sem_signal( semaphore *S ) // Must be executed atomically
S->value++;
if (S->value <= 0)
    remove thread t from S->tlist;
    wakeup(t);

Semaphore Example

What happens when sem is initialized to 2?

Assume three threads call sem_wait( &sem )

- Observations:
  - sem value is negative
    - Number of waiters on queue
  - sem value is positive
    - Number of threads that can be in critical section at the same time

Mutual Exclusion with Semaphores

- Previous example with locks:
  ```c
default void deposit( int amount )
    {
        pthread_mutex_lock( &sem );
        balance += amount; // critical section
        pthread_mutex_unlock( &sem );
    }
  ```

- Example with Semaphore:
  ```c
default void deposit( int amount )
    {
        sem_wait( &sem );
        balance += amount; // critical section
        sem_post( &sem );
    }
  ```

  What value should sem be initialized to?

Dangers with Semaphores

- Deadlock:
  - Two or more threads are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Example:
  - Two threads: Maria and Tucker
  - Two semaphores: semA, and semB both initialized to 1

  ```c
  sem_wait( semA );
  sem_wait( semB );
  sem_post( semA );
  sem_post( semB );
  ```

Binary Semaphore

- Binary semaphore is sufficient for mutex
  - Binary semaphore has boolean value (not integer)
  - bsem_wait(): Waits until value is 1, then sets to 0
  - bsem_signal(): Sets value to 1, waking one waiting process

- General semaphore is also called counting semaphore
Semaphore Verdict

- **Advantage:**
  - Versatile, can be used to solve any synchronization problems
- **Disadvantages:**
  - Prone to bugs
  - Difficult to program: no connection between semaphore and the data being controlled by the semaphore (later will see this better)
- Consider alternatives: Monitors, for example

Classical Problems: Reader Writers

Set of problems where data structures, databases or file systems are read and modified by concurrent threads

- **Idea:**
  - While data structure is updated often necessary to bar other threads from reading
- **Basic Constraints:**
  - Any number of readers can be in CS simultaneously
  - Writers must have exclusive access to CS
- **Some Variations:**
  - **First Reader:** No reader kept waiting unless writer already in CS
  - **Second reader:** Once a writer is ready the writer performs write as soon as possible

First Reader: Entrance/Exit Writer

```c
void enterWriter()
    sem_wait(&mutex)
void exitWriter()
    sem_post(&mutex);
```

- Writer can only enter if it possesses the lock

First Reader: Entrance/Exit Reader

```c
void enterReader()
    sem_wait(&mutex);
    reader++;
    if (reader == 1)
        sem_wait(&roomEmpty); // first on in locks
    sem_post(&mutex);

void exitReader()
    sem_wait(&mutex);
    reader--;
    if (reader == 0)
        sem_post(&roomEmpty); // last unlocks
    sem_post(&mutex);
```

- Only one reader is queued on roomEmpty
- When a reader signals roomEmpty no other readers are in the room

Two Classes of Synchronization Problems

- **Uniform resource usage with simple scheduling constraints**
  - No other variables needed to express relationships
  - Use one semaphore for every constraint
  - Examples: producer/consumer
- **Complex patterns of resource usage**
  - Cannot capture relationships with only semaphores
  - Need extra state variables to record information
  - Use semaphores such that
    - One is for mutual exclusion around state variables
    - One for each class of waiting
- Always try to cast problems into first, easier type
- Today: Two examples using second approach
First Reader: Entrance/Exit Reader

Dining Philosophers: Attempt 1

- Two neighbors can’t use chopstick at same time
- Must test if chopstick is there and grab it atomically
  - Represent each chopstick with a semaphore
  - Grab right chopstick then left chopstick
  - `sem_t_chopstick[5]`: Initialize each to 1

\[
\text{take_chopstick(int i)}; \\
\text{sem_wait( a_chopstick[i] );} \\
\text{sem_wait( a_chopstick[i+1] % 5 );} \\
\text{put_chopstick(int i)}; \\
\text{sem_post( a_chopstick[i] );} \\
\text{sem_post( a_chopstick[i+1] % 5 );}
\]

- Guarantees no two neighbors eats simultaneously
- What happens if all philosophers wants to eat and grabs the left fork?

Dining Philosophers: Attempt 2

- Add a mutex to ensure that a philosopher gets both chopsticks

\[
\text{take_chopstick(int i)}; \\
\text{sem_wait( a_chopstick[i] );} \\
\text{sem_wait( a_chopstick[(i+1) % 5] );} \\
\text{put_chopstick(int i)}; \\
\text{sem_post( a_chopstick[i] );} \\
\text{sem_post( a_chopstick[(i+1) % 5] );}
\]

- Problems?
  - How many philosophers can dine at one time?
  - How many should be able to eat?

Dining Philosophers: Attempt 3

- Grab lower-numbered chopstick first, then higher-numbered

\[
\text{take_chopstick(int i)}; \\
\text{if( i < 4 )} \\
\text{sem_wait( a_chopstick[i] );} \\
\text{sem_wait( a_chopstick[(i+1)] );} \\
\text{else} \\
\text{sem_wait( a_chopstick[5] );} \\
\text{sem_wait( a_chopstick[4] );}
\]

- Problems?
  - Safe: Deadlock? Asymmetry avoids it - it is safe
  - Performance (concurrency?)
    - P3 and P4 grabs chopstick simultaneously - assume P3 wins
    - P3 can now eat but P0 and P2 are not eating event if they don’t share a fork with P3

Dining Philosophers: Dijkstra

- Guarantee two goals:
  - Safety: Ensure nothing bad happens (don’t violate constraints of problem)
  - Liveness: Ensure something good happens when it can (make as much progress as possible)
- Introduce state variable for each philosopher i
  - state[i] = THINKING, HUNGRY, or EATING
- Safety: No two adjacent philosophers eat simultaneously
  - for all i: !(state[i]=EATING & state[(i+1)%5] = EATING)
- Liveness: No philosopher is hungry unless one of his neighbors is eating
  - Not the case that a philosopher is hungry and his neighbors are not eating
  - for all i: !(state[i]=HUNGRY & (state[(i+1)%5]=EATING & state[(i+2)%5]=EATING))
### Monitors

#### Motivation:
- Users can inadvertently misuse locks and semaphores (e.g., never unlock a mutex)

#### Idea:
- Languages construct that control access to shared data
- Synchronization added by compiler, enforced at runtime

#### Monitor encapsulation
- Shared data structures
- Methods
  - that operates on shared data structures
- Synchronization between concurrent method invocations

#### Protects data from unstructured data access
- Guarantees that threads accessing its data through its procedures interact only in legitimate ways

---

### Homework

**Due: 10/31 before class**

1. Prove that Peterson’s lock solution provides for mutual exclusion, no starvation, and bounded waiting?
2. Text book exercise: 6.9
4. Reading questions (p. 217-230):
   a) Describe Priority Inversion
   b) What is an atomic transaction
   c) What is a serializable schedule?
   d) Describe the two-phase locking protocol?