CSE120
Principles of Operating Systems

Yuanyuan (YY) Zhou
Synchronization
Inter-Process Communication (IPC)

- Communication
  - Pass information to each other
- Mutual exclusion & Synchronization
  - Keep each other’s hair
  - Proper sequencing

- The last one also applies to threads
- Any real life example for synchronization?
  - Two people talking at the same time, can you hear them clearly?
Before we start...

Thread 1

```c
void foo() {
    x++;
}
```

Thread 2

```c
void bar() {
    x--;
}
```

- x is a **global** variable and initially x=0;
- After thread 1 and 2 execute, what is the value of x?
  - The value could be 0, -1, 1
  - Want to know why?
    - Really pay attention, bad time to sleep
- Does the above case happen only on a multicore machine? Will it happen on a uniprocessor machine?
- What if x is a **local** variable claimed inside the function foo or bar, respectively?
A simple game

- **Two volunteers (each step is an “instruction”)**
  - Producer program: produce item per iteration
    - Step1: increment the counter on the board
    - Step2: put one item on the table
  - Consumer program:
    - Step1: read the counter LOUD
    - Step2a: if the counter is zero, go back to step1
    - Step2b: if the counter is nonzero, take one item from the table
    - Step3: decrement counter on the board
- **Rule**
  - only one should “execute” on the CPU at any time.
  - The other sits on the ready queue (no I/O in this game)
- **You are the OS**
  - You decide who should “execute”, who should be switched off
- **Can you get them into “trouble” before the items runs out?**
Silly Producer-Consumer Game

- Stop Producer before step2 and let Consumer go.
- What happens?

  - Producer: produce 1 item per iteration
    - Step1: increment the counter
    - Step2: put one item on the table
  - Consumer:
    - Step1: check the counter to see if it is zero
    - Step2a: if the counter is zero, go back to step1
    - Step2b: if the counter is nonzero, take one item from the table
    - Step3: decrement the counter
Data Races

- Reason: data sharing
- What are shared in the game?
  - Share the counter
  - Share the item
Shared Resources

- The problem is that two concurrent threads (or processes) accessed a shared resource (account) without any synchronization
  - Known as a race condition (memorize this buzzword)

- We need mechanisms to control access to these shared resources in the face of concurrency
  - So we can reason about how the program will operate

- Shared data structure
  - Buffers, queues, lists, hash tables, etc.
Can you give me some real world examples?

- What are shared in real world and require some synchronization?

- Can two teachers share the same projector and classroom at the same time?
  - What about your phone line (do you remember those walkie-talkie)
  - …
When Are Resources Shared?

- **Local variables are not shared** (private)
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2

- **Global variables and static objects are shared**
  - Stored in the static data segment, accessible by any thread

- **Dynamic objects and other heap objects are shared**
  - Allocated from heap with malloc/free or new/delete

- **Accesses to shared data need to be synchronized**
Why synchronize?

- Why accesses to shared data need to be synchronized?
  - Interleaving by an access from another thread to the same shared data between two subsequent accesses

```
Read X
```
```
Write X
```
```
Write X
```
Synchronization is like traffic signals

- Each thread is like a car----it can make progress independently with its own speed

- http://www.youtube.com/watch?v=nocS1Z4gcDU
How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of words
  - Some architectures don't even give you that!

- We'll assume that a context switch can occur at any time or the threads can run in different processors at the same time

- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever
Spooling Example: Problem-Free Interleaving

Thread 1
int next_free;

1. next_free = in;
2. Stores F1 into next_free;
3. in = next_free + 1

Thread 2
int next_free;

4. next_free = in
5. Stores F2 into next_free;
6. in = next_free + 1

Does this code always work?
Spooling Example: Races

Thread 1
int next_free;

1 next_free = in;

2 Stores F1 into next_free;

3 in = next_free + 1

Thread 2
int next_free;

4 next_free = in

/* value: 7 */

5 Stores F2 into next_free;

6 in = next_free + 1
Classic Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

  ```java
  withdraw (account, amount) {
      balance = get_balance(account);
      balance = balance – amount;
      put_balance(account, balance);
      return balance;
  }
  ```

- Now suppose that you and your significant other share a bank account with a balance of $1000.

- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.
Example Continued

- We’ll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

```c
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What’s the problem with this implementation?
  - Think about potential interleaving of these two threads
Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

  ```
  balance = get_balance(account);
  balance = balance – amount;
  balance = get_balance(account);
  balance = balance – amount;
  put_balance(account, balance);
  ```

- What is the balance of the account now?
- Is the bank happy with our implementation?
- What if this is not withdraw, but deposit?
Mutual Exclusion

- We want to use mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks

- Code that uses mutual exclusion to synchronize its execution is called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing your bathroom with housemates

- What requirements would you place on a critical section?
Critical Region (Critical Section)

Process {
    while (true) {
        ENTER CRITICAL SECTION
        Access shared variables; // Critical Section;
        LEAVE CRITICAL SECTION
        Do other work
    }
}
Critical Region Requirement

- **Mutual Exclusion**
  - No other process must execute within the critical section while a process is in it.

- **Progress**
  - If no process is waiting in its critical section and several processes are trying to get into their critical section, then entry to the critical section cannot be postponed indefinitely.

- **Bounded Wait**
  - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section.

- **No assumption**
  - No assumption may be made about speeds or number of CPUs.
Critical Regions (2)

Mutual exclusion using critical regions

A enters critical region
A leaves critical region
B attempts to enter critical region
B enters critical region
B leaves critical region

T₁ T₂ T₃ T₄

Time
Mechanisms For Building Critical Sections

- Atomic read/write
  - Can it be done?

- Locks
  - Primitive, minimal semantics, used to build others

- Semaphores
  - Basic, easy to get the hang of, but hard to program with

- Monitors
  - High-level, requires language support, operations implicit

- Messages
  - Simple model of communication and synchronization based on atomic transfer of data across a channel
  - Direct application to distributed systems
  - Messages for synchronization are straightforward (once we see how the others work)
Mutual Exclusion with Atomic Read/Writes: First Try

This is called alternation
(1) Does it satisfy mutual exclusion?
   • If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2 (why?)
   • (turn == 1) ≡ (turn != 2)

(2) Does it work?
No. It violates progress: the blue thread might be far away from the critical section, but the yellow thread still cannot enter the critical section
Mutual Exclusion with Atomic Read/Writes: Second Try

bool try1 = false, try2 = false;

Thread 1

...  
try1 = true;  
while (try2) ;  
    critical section  
try1 = false;  
...  

Thread 2

...  
try2 = true;  
while (try1) ;  
    critical section  
try2 = false;  
...  

Has thread 2 executed “try2=true?”. If not, I am safe. If yes, let’s see…

• Does it work?
Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

Thread 1

try1 = true;
turn = 2;
while (try2 && turn != 1) {
    critical section
    try1 = false;
}
...

Thread 2

try2 = true;
turn = 1;
while (try1 && turn != 2) {
    critical section
    try2 = false;
}
...

Did I execute "turn=2" before thread 2 executed "turn=1"?
Has thread 2 executed "try2=true?". If not, I am safe. If yes, let's see…

• Does it work?
• Try all possible interleavings
A lock is an object in memory providing two operations:
- acquire(): before entering the critical section
- release(): after leaving a critical section
- Some systems may refer this pair as Lock() / Unlock()

Threads pair calls to acquire() and release():
- Between acquire()/release(), the thread holds the lock
- acquire() does not return until any previous holder releases
- What can happen if the calls are not paired?

Locks can spin (a spinlock) or block (a mutex)
Using Locks

withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}

- What happens when blue tries to acquire the lock?
- Why is the “return” outside the critical section? What if not?
- What happens when a third thread calls acquire?
How do we implement locks?

Here is one attempt:

```c
struct lock {
    int held = 0;
};
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

This is called a \textit{spinlock} because a thread spins waiting for the lock to be released.

Does this work?
Implementing Locks (2)

No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition.
Implementing Locks (3)

- The problem is that the implementation of locks has critical sections, too
  - How do we stop the recursion?

- The implementation of acquire/release must be atomic
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes “all or nothing”

- How do we make them atomic?

- Need help from hardware
  - Atomic instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)
Atomic Instructions: Test-And-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to TRUE
  - Return the old value
- Hardware executes it **atomically**!

```c
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- When executing test-and-set on “flag”
  - What is **value of flag** afterwards if it was initially False? True?
  - What is the **return result** if flag was initially False? True?
Using Test-And-Set

Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
};
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it work?
- Does it work on multiprocessors?
Other Similar Hardware Instruction

- **Swap** = TSL

```c
void Swap (char* x, *y);
\ All done atomically
{
    char temp = *x;
    *x = *y;
    *y = temp
}
```
Problems with Spinlocks

- The problem with spinlocks is that they are wasteful
  - If a thread is spinning on a lock, it occupies the CPU, and the other thread holding the lock is not executing to get out of the critical section and release the lock

- Solution:
  - If cannot get the lock, call `thread_yield` to give up the CPU!

- Solution 2: sleep and wakeup
  - When blocked, go to sleep
  - Wakeup when it is OK to retry entering the critical section
Higher-Level Synchronization

- Spinlocks 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
Summary

1. Why we need synchronizations
   - Races to shared data from concurrent threads

2. Critical sections

3. Simple algorithms to implement critical sections

4. Locks

5. Lock implementations
   - Spin-lock using atomic instructions
   - Spin lock using disable/enable interrupt

6. Next lecture
   - Semaphores
   - Monitor
   - Communication
   - Some classic problems