VL09 Concurrency in Algorithms

31. January

Agenda

- Hashing: CRAM-MD5
- Interprocess Communication
- Concurrency in Algorithms
- Synchronization Primitives
- Advanced Synchronization

Hashing: CRAM-MD5

CRAM-MD5 is a challenge-response authentication mechanism based on the MD5 algorithm.

The CRAM-MD5 protocol involves a single challenge request and response, and is initiated by the server. The authentication looks roughly like this:

1. Challenge: The server sends a random string to the client.
2. Response: The client responds with a string created as follows.
    - The challenge is hashed using a hash function, with a shared secret (typically, the password, or a hash thereof) as the secret key.
    - The result is sent back to the server.
3. Comparison: The server repeats the same process to compute the expected response. If the given response and the expected response match, then authentication was successful.

Client	Server
user would like to log in	

Client	Server
	generate a random challenge string "randomABC" and send it
input → "mypassword123"	
hashed_input = MD5(input) → 9c87baa223f464954940f859bcf2e233	
hashed_input += "randomABC"	
final_hash = MD5(hashed_input)	
send user name and final_hash to the server	
	Server looks up the password hash for the user
	The same computation is performed
	final_hash

Client	Server
	from client and server are compared

Interprocess Communication

Interprocess communication (IPC) refers to capabilities of an operating system that allow running processes to communicate and share their data.

- File system
  - Files and directories
  - Pipes
- Signals
- Sockets
- Shared memory

Concurrency in Algorithms

Book Threading in C#, Joseph Albahari

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by the scheduler. Threads share the same memory space within a process/application.

Motivation

- Keeping a responsive user interface
- Parallel programming
- Server solutions, handling multiple requests
- Speculative execution

Elementary Operations

- Thread creation and abortion, exit-synchronization (owner/parent)
- Thread exit, sleep and wake (intra-thread)

When the application starts it is represented by a single thread. As long as there is only one thread running, there are no concerns about concurrency. Additional threads can be created easily.

Thread Types

- Foreground threads - the application is running as long as there is at least one foreground thread
- Background threads (detached threads) - get aborted with application exit

Synchronization Primitives

The lock statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released.

```
object myNiceLock = new object();

lock (myNiceLock)
{
    // Critical section right
here...

    // the locking is re-entrant
    // that is useful if you call
another method with lock statement
    lock (myNiceLock)
    {
        // so this is ok
    }
}
```

Java counterpart: synchronized

Atomic Operations

Atomic operations are uninterruptible, thread-switching can happen just before or just after them. The programming environment defines which operations are atomic. If unsure, always use synchronization.

Problems

- Missing synchronization → **can lead** to data corruption
- Wrong synchronization → **can lead** to a deadlock

```
uint8_t byte;

byte = 10;      // this is an atomic
operation


// This division is probably non-
atomic consisting of
// multiple instructions. Consider a
different thread accessing
// the variable during this
computation.
byte = another_byte / 13;
```

Sleeping and Signaling: Wait, Pulse, PulseAll

- Wait - put the thread into associative sleep state on a wait queue. Waiting (or sleeping) thread does not consume CPU power.
- Pulse - wake the first sleeping thread
- PulseAll - wake all threads in the queue

Java counterparts: Object.wait, Object.notify, Object.notifyAll

Spinlock

A spinlock is a blocking mechanism which causes a

thread trying to acquire it to simply wait in a loop while repeatedly checking if the condition is fulfilled.

```
// busy waiting for green color

// the variable flGreenColor will be set by a different thread
while(!flGreenColor) { }

// driving car
```

Advanced Synchronization

Advanced synchronization use synchronization primitives with extra logic to achieve more sophisticated and complex principles.

- **Semaphore**
  The semaphore class works similar to the lock statement but lets you set a limit on how many threads have access to the critical section. It's often described as a nightclub where the visitors (threads) stand in a queue outside the nightclub waiting for someone to leave in order to gain entrance.
- **AutoResetEvent**
  Wait / Pulse combination; .NET
  Tollbooth allowing one car to pass and automatically closing before the next one can go through.
- **ManualResetEvent**
  Wait / PulseAll combination; .NET
  Door which needs to be closed manually.

Exercise 0: Rendezvous

Alice and Bob would like to meet each other in the park. Independently, they will come to the park at a random time between 7 and 8 p.m. Whoever comes first, waits for the second person up to 20 minutes. How high is the probability that they meet?

- Write an algorithm that determines the probability
- Determine the analytical solution of the problem