L12B: Concurrent Programming

CS1101S: Programming Methodology

Low Kok Lim

November 3, 2021

Readings

- SICP JS, Sec 3.4: "Concurrency: Time is of the Essence"

Outline

- Concurrency
- Applications of Serializers
- Concurrent programming for multiple shared resources
- More synchronization mechanisms
- Classical synchronization problems

Outline

- **Concurrency**
  - Correctness
  - Serialization
- Applications of Serializers
- Concurrent programming for multiple shared resources
- More synchronization mechanisms
- Classical synchronization problems

What is Concurrent Computing?

- **Concurrent computing**
  - A form of computing in which several computations are executed **concurrently** instead of **sequentially** [Wikipedia]
    - **concurrently** — during overlapping time periods
    - **sequentially** — one completing before the next starts

Why Concurrent Computing?

- **Performance**
  - Dividing a complex task into multiple smaller subtasks and executing the subtasks simultaneously on different processors
- **Responsiveness**
  - Providing a responsive user interface even when there are other tasks executing independently
- **Abstraction** and **modularity**
  - Allowing more natural and modular modeling of real-world objects, which are acting concurrently

Concurrency in Source

- Running concurrent threads using [Source §3 Concurrent]
  - Use the primitive function `concurrent_execute`:

    $\texttt{concurrent\_execute(} f_1\texttt{, } f_2\texttt{, ..., } f_n \texttt{)}$

    - Each f_i must be a nullary function that returns `undefined`
    - The function `concurrent_execute` sets up a separate **thread** t_i that executes the body of f_i
    - These threads t_i all run **concurrently** with the main thread

Example 1: Concurrency in Source

```
function sum_1_to_n(n) {
    let sum = 0;
    for (let i = 1; i <= n; i = i + 1) { sum = sum + i; }
    return sum;
}

function thread1() {
    display(get_time() - prog_start_tm, "Thread 1 start time:");
    sum_1_to_n(5000);
    display(get_time() - prog_start_tm, "Thread 1 middle time:");
    sum_1_to_n(5000);
    display(get_time() - prog_start_tm, "Thread 1 end time:");
}

function thread2() {
    display(get_time() - prog_start_tm, "Thread 2 start time:");
    sum_1_to_n(5000);
    display(get_time() - prog_start_tm, "Thread 2 middle time:");
    sum_1_to_n(5000);
    display(get_time() - prog_start_tm, "Thread 2 end time:");
}
// continue next page
```

Show in Playground

Example 1: Concurrency in Source

```
// continue from previous page

let prog_start_tm = 0;

// Sequential run
prog_start_tm = get_time();
display("Sequential run:");
thread1(); thread2();

// Concurrent run
prog_start_tm = get_time();
display("Concurrent run:");
concurrent_execute(thread1, thread2);
```

[Show in Playground]

Example 1: Concurrency in Source

- Sample output:

```
"Sequential run:"
Thread 1 start time: 2
Thread 1 middle time: 88
Thread 1 end time: 159
Thread 2 start time: 159
Thread 2 middle time: 226
Thread 2 end time: 292
"Concurrent run:"
Thread 2 start time: 0
Thread 1 start time: 0
Thread 2 middle time: 157
Thread 1 middle time: 158
Thread 1 end time: 292
Thread 2 end time: 294
```

Show in Playground

- What do you observe?
  - “Concurrent run” took about same amount of time as “Sequential run”
    - **Concurrency** does not imply **parallelism**

Concurrency vs Parallelism

- **Concurrency**
  - When execution of multiple tasks **overlap in time**
  - Can happen on a single core or processor

- **Parallelism**
  - When execution of multiple tasks **occur simultaneously in time**
  - Must happen on multiple cores or processors

Example 2: Interleaving of Threads

- Another example to show the **interleaving** of **events** in concurrent threads

Show in Playground

```
function delay(ms) {
    const start_tm = get_time();
    while (get_time() - start_tm < ms) {
        /* do nothing */
    }
}

function thread_A() {
    delay(20 * math_random());
    display("A1");
    delay(20 * math_random());
    display("A2");
    delay(20 * math_random());
    display("A3");
}
```

```
function thread_B() {
    delay(20 * math_random());
    display("B1");
    delay(20 * math_random());
    display("B2");
    delay(20 * math_random());
    display("B3");
}

// Concurrent run
concurrent_execute(thread_A,
                   thread_B);
```

Example 2: Interleaving of Threads

- Outputs from a few sample runs:

Show in Playground

"A1"	"B1"	"B1"	"B1"
"A2"	"A1"	"B2"	"B2"
"B1"	"A2"	"B3"	"A1"
"A3"	"B2"	"A1"	"A2"
"B2"	"A3"	"A2"	"B3"
"B3"	"B3"	"A3"	"A3"

Example 3: Interleaving of Threads

```
function delay(ms) {
    const start_tm = get_time();
    while (get_time() - start_tm < ms) {
        /* do nothing */
    }
}
```

- This example shows two concurrent threads updating a **shared variable**

```
function thread_A() {
    const s = x;
    const t = s + 1;
    x = t;
}

function thread_B() {
    const s = x;
    const t = s - 1;
    x = t;
}

let x = 0;

concurrent_execute(thread_A,
                   thread_B);
delay(200);
display(x, "Final x:");
```

Show in Playground

Example 3: Interleaving of Threads

- Outputs from a few sample runs:

[Show in Playground]

```
Final x: 0
```

```
Final x: -1
```

```
Final x: 1
```

- How did we get these different results?
  - We have a **race condition**, where “correct” results are dependent on the ordering or timing of the concurrent events
- Which ones are considered “**correct**”?

Correct Behavior of Concurrent Programs

- A concurrent program is considered **correct** if it produces the same result ***as if*** the threads had run ***sequentially*** in ***some*** order
  - It does not require the threads *to actually run sequentially*, but only to produce results that are the same ***as if*** *they had run* ***sequentially***
  - There may be more than one possible “correct” result produced by the concurrent program, because we require only that the result be the same as for ***some*** sequential order

Mechanisms for Controlling Concurrency

- General mechanisms that allow us to **constrain the interleaving** of concurrent threads to **ensure correct** program behavior

- Many mechanisms have been developed
  - Also known as **synchronization mechanisms**

- Here, we consider one of them, the ***serializer***

Serialization

- **Serialization** implements the following idea:
  - Threads will execute concurrently, but **certain sets of functions** cannot be executed concurrently

- **Serialization** creates **distinguished sets of functions** such that only one execution of a function in each **serialized set** is permitted to happen at a time
  - If some function in the set is being executed, then a thread that attempts to execute any function in the set will be **forced to wait** until the first execution has finished

- We can use **serialization** to control access to **shared variables**

Serializers in Source

- **Serializers** are constructed by the `make_serializer` function
  - `make_serializer` is not a primitive or pre-declared function in **Source §3 Concurrent**
  - **Source §3 Concurrent** provides additional primitive functions to allow the implementation of `make_serializer`

- A **serializer** takes a function as argument and returns a serialized function that behaves like the original function

- All calls to a given **serializer** return serialized functions in the same set

Example Use of Serializers

- Executing the following

```
let x = 10;

const s = make_serializer();

concurrent_execute(s(() => { x = x * x; }),

                   s(() => { x = x + 1; }));
```

can produce only two possible values for x: 101 or 121

Example 4: Serializers

```
function make_serializer() {...}

function delay(ms) {...}


const ser = make_serializer();
```

- This example adds the use of a serializer to Example 3
- Its output is now always

```
Final x: 0
```

```
function thread_A() {
    const s = x;
    const t = s + 1;
    x = t;
}

function thread_B() {
    const s = x;
    const t = s - 1;
    x = t;
}

let x = 0;

concurrent_execute(ser(thread_A),
                   ser(thread_B));
delay(200);
display(x, "Final x:");
```

Show in Playground

Example 5: Serializers

```
function make_serializer() {...}
function delay(ms) {...}

const ser_x = make_serializer();
const ser_y = make_serializer();

function thread_A() {
   ser_y(() => {
        const s = y;
        const t = s - 1;
        y = t;
    })();

    ser_x(() => {
        const s = x;
        const t = s + 1;
        x = t;
    })();
}
```

```
function thread_B() {
    ser_x(() => {
        const s = x;
        const t = s - 1;
        x = t;
    })();

    ser_y(() => {
        const s = y;
        const t = s + 1;
        y = t;
    })();
}

let x = 0;
let y = 0;
concurrent_execute(thread_A, thread_B);
delay(200);
display(x, "Final x:");
display(y, "Final y:");
```

Show in Playground

Output :

```
Final x: 0
Final y: 0
```

Implementing Serializers

- We implement serializers in terms of a more primitive synchronization mechanism called a ***mutex***
  - Abbreviation for ***mutual exclusion***
  - A **mutex** is an object that supports two operations:
    - **acquired** and **released**
  - Once a mutex has been acquired, no other acquire operations on that mutex may proceed until the mutex is released
- In our implementation, each **serializer** has an associated **mutex**

Implementing Serializers

```
function make_serializer() {
    const mutex = make_mutex();
    return f => {
        function serialized_f() {
            mutex("acquire");
            const val = f();
            mutex("release");
            return val;
        }
        return serialized_f;
    };
}
```

Implementing Mutexes

```
function make_mutex() {
    const cell = list(false);
    function the_mutex(m) {
        return m === "acquire"
                   ? test_and_set(cell)
                     ? the_mutex("acquire") // retry
                     : true
                   : m === "release"
                   ? clear(cell)
                   : error(m, "Unknown request -- mutex");
    }
    return the_mutex;
}
```

Implementing Mutexes

- The `clear` and `test_and_set` functions are primitive functions provided in Source §3 Concurrent
  - Each is performed ***atomically*** (i.e. without interruption)
  - They have the following meanings*:

```
function clear(cell) {
    set_head(cell, false);
}
```

```
function test_and_set(cell) {
    if (head(cell)) {
        return true;
    } else {
        set_head(cell, true);
        return false;
    }
}
```

Outline

- Concurrency
- **Applications of Serializers**
- Concurrent programming for multiple shared resources
- More synchronization mechanisms
- Classical synchronization problems

Example 6: Bank Account

- Want to implement a **bank account** that allows **concurrent withdrawal** and **deposit** transactions
  - For example, a joint bank account concurrently accessed from different ATMs

Example 6a: Implementation without Serializers

```
function delay(ms) {...}

function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    function dispatch(m) {
        return m === "withdraw"
            ? withdraw
            : m === "deposit"
            ? deposit
            : m === "balance"
            ? balance
            : error(m,
                "Unknown request");
    }
    return dispatch;
}
```

```
const my_account = make_account(100);

function thread_A() {
    my_account("withdraw")(50);
    my_account("deposit")(100);
}

function thread_B() {
    my_account("withdraw")(50);
    my_account("deposit")(100);
}

concurrent_execute(thread_A, thread_B);

delay(200);
display(my_account("balance"),
        "Final balance:");
```

Show in Playground

Example 6a: Implementation without Serializers

- Outputs from a few sample runs:

```
Final balance: 100
```

```
Final balance: 150
```

```
Final balance: 200
```

```
Final balance: 250
```

Example 6b: Implementation with Serializers

```
function make_serializer() {...}
function delay(ms) {...}

function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }

    const protect = make_serializer();
    const protect_withdraw = protect(withdraw);
    const protect_deposit = protect(deposit);
```

```
    function dispatch(m) {
        return m === "withdraw"
               ? protect_withdraw
               : m === "deposit"
               ? protect_deposit
               : m === "balance"
               ? balance
               : error(m,
                   "Unknown request");
    }
    return dispatch;
}

// continue next page
```

Show in Playground

Example 6b: Implementation with Serializers

```
// continue from previous page

const my_account = make_account(100);

function thread_A() {
    my_account("withdraw")(50);
    my_account("deposit")(100);
}

function thread_B() {
    my_account("withdraw")(50);
    my_account("deposit")(100);
}

concurrent_execute(thread_A, thread_B);

delay(200);
display(my_account("balance"),
        "Final balance:");
```

- Its output is now always

```
Final balance: 200
```

Show in Playground

Outline

- Concurrency
- Applications of Serializers
- **Concurrent programming for multiple shared resources**
- More synchronization mechanisms
- Classical synchronization problems

Example 7: Exchanging Account Balances

- Want to implement a function to **exchange balances** of <u>**two**</u> **bank accounts** and can work **concurrently** with **withdrawal**, **deposit** and **other exchange balances** transactions

- A concurrent operation that manipulates **multiple shared resources**

Example 7a: Implementation (Incorrect)

```
function make_serializer() {...}
function delay(ms) {...}

function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }

    const protect = make_serializer();
    const protect_withdraw = protect(withdraw);
    const protect_deposit = protect(deposit);
```

```
    function dispatch(m) {
        return m === "withdraw"
            ? protect_withdraw
            : m === "deposit"
            ? protect_deposit
            : m === "balance"
            ? balance
            : error(m,
                "Unknown request");
    }
    return dispatch;
}

// continue next page
```

Show in Playground

Example 7a: Implementation (Incorrect)

```
// continue from previous page

function exchange(accounts) {
    const account1 = head(accounts);
    const account2 = tail(accounts);
    const difference = account1("balance")
                       - account2("balance");
    account1("withdraw")(difference);
    account2("deposit")(difference);
}

const acc1 = make_account(100);
const acc2 = make_account(200);

function thread_A() {
    exchange(pair(acc1, acc2));
}
function thread_B() {
    exchange(pair(acc1, acc2));
}

concurrent_execute(thread_A, thread_B);

delay(200);
display(acc1("balance"), "Acc.1 final balance:");
display(acc2("balance"), "Acc.2 final balance:");
```

- Outputs from a few sample runs:

```
Acc.1 final balance: 300
Acc.2 final balance: 0
```

```
Acc.1 final balance: 100
Acc.2 final balance: 200
```

```
Acc.1 final balance: 200
Acc.2 final balance: 100
```

[Show in Playground]

Example 7b: Implementation (Correct*)

```
function make_serializer() {...}
function delay(ms) {...}


function make_account_and_serializer(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }

    const serializer = make_serializer();
```

```
    function dispatch(m) {
        return m === "withdraw"
            ? withdraw
            : m === "deposit"
            ? deposit
            : m === "balance"
            ? balance
            : m === "serializer"
            ? serializer
            : error(m,
                "Unknown request");
    }
    return dispatch;
}

// continue next page
```

Show in Playground

Example 7b: Implementation (Correct*)

```
// continue from previous page

function exchange(accounts) {
    const account1 = head(accounts);
    const account2 = tail(accounts);
    const difference = account1("balance") - account2("balance");
    account1("withdraw")(difference);
    account2("deposit")(difference);
}

function serialized_exchange(accounts) {
    const account1 = head(accounts);
    const account2 = tail(accounts);
    const serializer1 = account1("serializer");
    const serializer2 = account2("serializer");
    serializer1(serializer2(exchange))(accounts);
}

// continue next page
```

Show in Playground

Example 7b: Implementation (Correct*)

```
// continue from previous page

const acc1 = make_account_and_serializer(100);
const acc2 = make_account_and_serializer(200);

function thread_A() {
    serialized_exchange(pair(acc1, acc2));
}
function thread_B() {
    serialized_exchange(pair(acc1, acc2));
}

concurrent_execute(thread_A, thread_B);

delay(200);
display(acc1("balance"), "Acc.1 final balance:");
display(acc2("balance"), "Acc.2 final balance:");
```

- Its output is now always

```
Acc.1 final balance: 100
Acc.2 final balance: 200
```

Show in Playground

Example 7c: Beware

- Now let’s make the following change to the previous program:

```
// continue from previous page

const acc1 = make_account_and_serializer(100);
const acc2 = make_account_and_serializer(200);

function thread_A() {
    serialized_exchange(pair(acc1, acc2));
}
function thread_B() {
    // serialized_exchange(pair(acc1, acc2));
    serialized_exchange(pair(acc2, acc1));  // changed here
}

concurrent_execute(thread_A, thread_B);

delay(200);
display(acc1("balance"), "Acc.1 final balance:");
display(acc2("balance"), "Acc.2 final balance:");
```

Show in Playground

Example 7c: Beware

- We get this error message:

[Show in Playground]

```
Potential infinite loop detected.
...
```

- Why?
  - Program has run into a **deadlock**
    - **Deadlock** — no thread can make any progress
- How?

Example 7c: How Deadlock Happened?

- How did the **deadlock** happen?

```
function serialized_exchange(accounts) {
    const account1 = head(accounts);
    const account2 = tail(accounts);
    const serializer1 = account1("serializer");
    const serializer2 = account2("serializer");
    serializer1(serializer2(exchange))(accounts);
}
```

```
function make_serializer() {
    const mutex = make_mutex();
    return f => arg => {
                  mutex("acquire");
                  const val = f(arg);
                  mutex("release");
                  return val;
                };
}
```

Example 7c: How Deadlock Happened?

- Original program:

thread_A:

```
acc1_mutex("acquire");
acc2_mutex("acquire");
exchange(pair(acc1, acc2));
acc2_mutex("release");
acc1_mutex("release");
```

thread_B:

```
acc1_mutex("acquire");
acc2_mutex("acquire");
exchange(pair(acc1, acc2));
acc2_mutex("release");
acc1_mutex("release");
```

- Modified program:

thread_A:

```
acc1_mutex("acquire");
acc2_mutex("acquire");
exchange(pair(acc1, acc2));
acc2_mutex("release");
acc1_mutex("release");
```

thread_B:

```
acc2_mutex("acquire");
acc1_mutex("acquire");
exchange(pair(acc2, acc1));
acc1_mutex("release");
acc2_mutex("release");
```

Example 7c: How Deadlock Happened?

- Modified program:

thread_A:

```
acc1_mutex("acquire");
acc2_mutex("acquire");
exchange(pair(acc1, acc2));
acc2_mutex("release");
acc1_mutex("release");
```

thread_B:

```
acc2_mutex("acquire");
acc1_mutex("acquire");
exchange(pair(acc2, acc1));
acc1_mutex("release");
acc2_mutex("release");
```

- Possible interleaving that led to deadlock:

Time	thread_A	thread_B
1	`acc1_mutex("acquire");`	
2		`acc2_mutex("acquire");`
3	`acc2_mutex("acquire");`	
4	*blocked*	`acc1_mutex("acquire");`
5	*blocked*	*blocked*
⋮	⋮	⋮

Deadlock Avoidance

- How to avoid deadlock (for this situation)?
  - Give each account a **unique identification**
  - Each concurrent thread will **protect the lowest-numbered account first**

- There are other situations that require more **sophisticated deadlock-avoidance techniques**, or where **deadlock cannot be avoided** at all

Outline

- Concurrency
- Applications of Serializers
- Concurrent programming for multiple shared resources
- **More synchronization mechanisms**
- Classical synchronization problems

Synchronization Mechanisms

- Synchronization mechanisms
  - **Serializers**
  - **Mutexes**
    - A **mutex** can be used to protect a **critical section** of a thread that accesses a **shared resource**
    - It guarantees **mutually exclusive** access to the **shared resource** at any given time
    - Threads that want to access a resource protected by a mutex must **wait until** the currently active thread is finished and **unlock** the mutex
  - **Semaphores**
  - etc.

Semaphores

- **Semaphore**
  - A generalized synchronization mechanism
  - Only behaviors are specified; can have different implementations
  - Provides
    - A way to **block** a number of processes/threads
      - Known as **sleeping process/threads**
    - A way to **unblock/wake up** one or more sleeping processes/threads
  - Proposed by **Edgar W. Dijkstra** in 1965

Semaphores

- A semaphore **S** contains an **integer** value
  - Initialized to any **non-negative** values initially

- Two **atomic** semaphore operations:

Down: (a.k.a. **P** and **Wait**)

```
1. If S <= 0, blocks (go to sleep)

2. Decrement S
```

Up: (a.k.a. **V** and **Signal**)

```
1. Increment S

2. Wakes up one sleeping process if any
```

Semaphores: Implementation using Mutex*

```
function make_semaphore(val) {
    const val_mutex = make_mutex();
    function semaphore(m) {
        if (m === "up") {
            val_mutex("acquire");
            val = val + 1;
            // Wake up one waiting thread if any.
            val_mutex("release");
        } else if (m === "down") {
            val_mutex("acquire");
            while (val <= 0) {
                val_mutex("release");
                // Put thread to sleep.
                val_mutex("acquire");
            }
            val = val - 1;
            val_mutex("release");
        } else {
            error(m, "Unknown request -- semaphore");
        }
    }
    return semaphore;
}
```

[Show in Playground]

Outline

- Concurrency
- Applications of Serializers
- Concurrent programming for multiple shared resources
- More synchronization mechanisms
- **Classical synchronization problems**

Classical Synchronization Problems

- The Producers-Consumers problem
- The Readers-Writers problem
- The Dining Philosophers problem

The Producers-Consumers Problem

- The **Producers-Consumers** problem specification
  - Threads share a **bounded circular buffer** (array) of size **N**
  - **Producers** produce items and insert them to buffer
    - Only when buffer is **not full** (< N items)
  - **Consumers** remove items from buffer
    - Only when buffer is **not empty** (> 0 items)

The Producers-Consumers Problem: A Solution

```
const N = ...;
const buffer = [];
let count = 0;
let in_pos = 0;
let out_pos = 0;
```

```
const mutex = make_semaphore(1);
const not_full = make_semaphore(N);
const not_empty = make_semaphore(0);
```

```
function producer() {
    while (true) {
        const item = produce_item(...);

        not_full("down");
        mutex("down");

        buffer[in_pos] = item;
        in_pos = (in_pos + 1) % N;
        count = count + 1;

        mutex("up");
        not_empty("up");
    }
}
```

```
function consumer() {
    while (true) {
        not_empty("down");
        mutex("down");

        const item = buffer[out_pos];
        out_pos = (out_pos + 1) % N;
        count = count - 1;

        mutex("up");
        not_full("up");

        consume_item(item);
    }
}
```

The Readers-Writers Problem

- The **Readers-Writers** problem specification
  - Threads share a data structure **D**
  - **Readers** retrieve information from **D**
    - Each reader can access **D** with other readers
  - **Writers** modify information in **D**
    - Each writer must have exclusive access to **D**

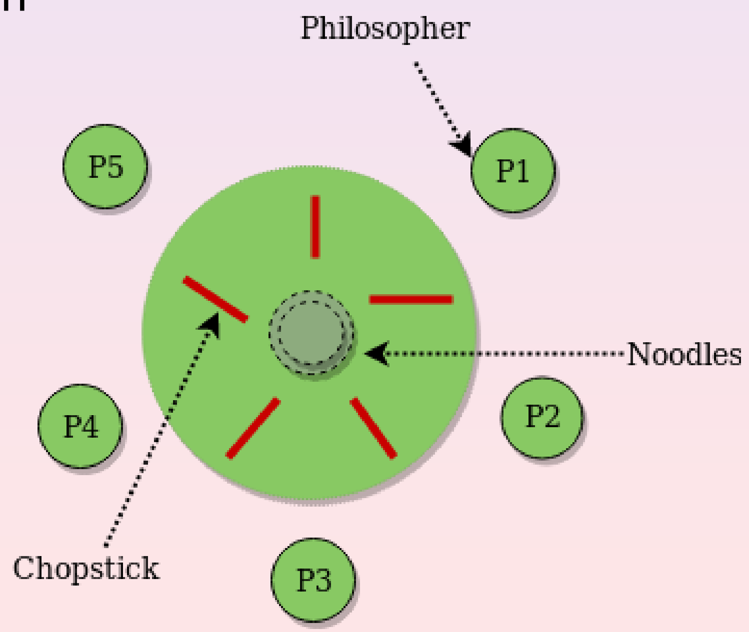
The Readers-Writers Problem: A Solution*

```
const D = make_data_structure(...);

let num_readers = 0;

const mutex = make_semaphore(1);
const room_empty = make_semaphore(1);



function writer() {
    while (true) {
        room_empty("down");

        modify_data(D, ...);

        room_empty("up");
    }
}
```

```
function reader() {
    while (true) {
        mutex("down");

        num_readers = num_readers + 1;
        if (num_readers === 1) {
            room_empty("down");
        } else {}

        mutex("up");

        read_data(D, ...);

        mutex("down");

        num_readers = num_readers - 1;
        if (num_readers === 0) {
            room_empty("up");
        } else {}

        mutex("up");
    }
}
```

The Dining Philosophers Problem

- The **Dining Philosophers** problem specification
  - Five **philosophers** are seated at a round table
    - With a big bowl of noodles at the center
  - Five single **chopsticks** are placed between each pair of adjacent philosophers
  - Each philosopher must alternately **think** and **eat**
  - A philosopher can only eat when he/she has **both left and right chopsticks**
  - Assume unlimited supply of noodles and unlimited appetite



[Image from www.geeksforgeeks.org]

The Dining Philosophers Problem: Attempt #1

```
function philosopher(i) {
    while (true) {
        think();
        take_left_chopstick(i);
        take_right_chopstick(i);
        eat();
        put_left_chopstick(i);
        put_right_chopstick(i);
    }
}
```

- Any problem?
  - **Deadlock**
    - E.g. all philosophers simultaneously take up left chopsticks, and none can proceed

The Dining Philosophers Problem: Attempt #2

```
function philosopher(i) {
    while (true) {
        think();
        mutex("acquire");
        take_left_chopstick(i);
        take_right_chopstick(i);
        eat();
        put_left_chopstick(i);
        put_right_chopstick(i);
        mutex("release");
    }
}
```

- Any problem?
  - Only **one philosopher** can eat at a time
  - Possible **starvation** of some philosophers

The Dining Philosophers Problem: A Good Solution

- An exercise or research for you

Summary

- Applications of Serializers
- Concurrent programming for multiple shared resources
  - Be careful of deadlocks
- More synchronization mechanisms
  - Mutexes
  - Semaphores
- Classical synchronization problems
  - The Producers-Consumers problem
  - The Readers-Writers problem
  - The Dining Philosophers problem