Introduction to Concurrent Programming

The study of systems of interacting computer programs whichshare resources and run concurrently (i.e. at the same time). Concurrency, and hence process synchronization, is useful onlywhen processes interact with each other

Parallelism: Occurring *physically* at the same time

Concurrency: Occurring *logically* at the same time, but could be implemented without real parallelism

Competative Processes

Deadlock: each thread takes a resource and waits indefinitely until the other one has freed that recource.

Livelock: each thread takes a resource, sees that the other thread has the other resource and returns it (this repeats indefinitely).

Starvation: one of the threads always takes the resources before the other one.

> Communication mechanisms are necessary for cooperation tobe possible.

Modeling Program Execution

Number of interleavings is exponential in the number ofinstructions. If *P* has *m* instructions and *Q* has *n* instructions, then there are

$$\binom{m+n}{m} = \frac{(m+n)!}{m!n!}$$

A **Transition System** *A* is a tuple $(S, \rightarrow, I)$ where *S* is the set of states, $\rightarrow \subseteq S \times S$ is a transition relation, and $I \subseteq$ is the set of initial states.

> *A* is said to be finite if *S* is finito.

> We write $s \rightarrow s'$ for $(s, s') \in \rightarrow$

Mutual Exclusion

Atomicity Assumption on Statements

An instruction such as `x = x+1` is actually compiled into simpler sets of assembly instructions. This could cause undesired interleavings at the assembly level.

> Atomic Operation
>
> An operation is atomic if it cannot be interleaved at a lower level ofabstraction.

- Atomic operations are the smallest units in terms of which a path can be constituted
- In order to reason about concurrency at the `source` level, we need to know what is assumed atomic at the `source` level
  - We assume throughout this course that all (single-line) statements are atomic
  - In particular, assignments such as `counter = counter+1` are assumed to be atomic

We occasionally simulate lack of atomicity of assignment (atthe source level) by decomposing it into several simpleinstructions.

For example:

```
counter = counter +1;
```

can be decomposed into:

```
temp = counter +1;
counter = temp;
```

> An occurrence of variable `v` in `P` is a critical reference if
>
> 1. it is assigned in `P` and occurs in another process `Q` , or
> 2. it is read in `P` and is assigned in another process `Q` .
>
> A program satisfies the `Limited Critical Reference (LCR)` property if every statement contains **at most one** critical reference
>
> Concurrent programs that satisfy the LCR restriction yield the same set of behaviors whether the statements are considered atomic or are compiled to a machine architecture with atomic load and store.
>
> Attempting to present programs that satisfy the LCR restriction is thus convenient

Race Condition

Once thread `T1` starts doing something, it needs to *"race"* to finish it because if thread `T2` looks at the shared variablebefore `T1` is done, it may see something inconsistent

> Race Condition
>
> Arises if two or more threads access the same variables or objects concurrently and at least one does updates

This is hard to detect and there can be race conditions in programs that satisfy LCR.

Critical Section

> Critical Section
>
> A part of the program that accesses shared memory and which wewish to execute atomically

```
Thread.start {
  while(true) {
    // non-critical  section
    entry  to  critical  section;
    //  CRITICAL  SECTION
    exit  from  critical  section;
    // non-critical  section
  }
}
```

> The Mutual Exclusion Problem
>
> 1. `Mutex` : At any point in time, there is at most one thread inthe critical section
> 2. `Absence of livelock` : If various threads try to enter the criticalsection, at least one of them will succeed
> 3. `Free from starvation` : A thread trying to enter its criticalsection will eventually be able to do so

Reasonable Assumptions of MEP

- There are no shared variables between the critical section andthe non critical section (nor with the entry/exit protocol).
- The critical section always terminates.
- The **scheduler** is **fair** (a process that is waiting to run, will be able to do so eventually)

General Scheme to Address the MEP

```
while (cond) { }; // is called a busy-wait loop

while (cond) {} → await !cond
```

We will assume that `await` is atomic, for this an all subsequent programs. For most of our examples this makes no difference since it satisfies the LCR property.

Attempt I

```
1 int turn = 1;
2
3 Thread.start {  // P
4   while (true) {
5     await (turn==1);
6     turn = 2;
7   }
8 }
```

```
1 Thread.start {  // Q
2   while (true) {
3     await (turn==2);
4     turn = 1;
5   }
6 }
```

In this case, mutual exclusion does not hold since both threads can enter the critical section.

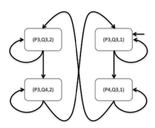
Attempt II

```
1  boolean wantP = false;
2  boolean wantQ = false;
3
4  Thread.start {  // P
5    while (true) {
6      // non-critical section
7      await !wantQ;
8      wantP = true;
9      // CRITICAL SECTION
10     wantP = false;
11     // non-critical section
12   }
13 }
```

```
1  Thread.start {  // Q
2    while (true) {
3      // non-critical section
4      await !wantP;
5      wantQ = true;
6      // CRITICAL SECTION
7      wantQ = false;
8      // non-critical section
9    }
10 }
```

Can guarentee MEP holds because there will never be a state `(P4, Q4, ?)` in the transition system. This also enjoys absence of livelock, but not freedom from starvation.

Attempt I: Transition System



This is because if thread P goes into an infinite loop/throws an exception it will never complete and set the value of turn to 2 causeing thread Q to never run.

Attempt III

```
1  boolean wantP = false;
2  boolean wantQ = false;
3
4  Thread.start {  // P
5    while (true) {
6      // non-critical section
7      wantP = true;
8      await !wantQ;
9      // CRITICAL SECTION
10     wantP = false;
11     // non-critical section
12   }
13 }
```

```
1  Thread.start {  // Q
2    while (true) {
3      // non-critical section
4      wantQ = true;
5      await !wantP;
6      // CRITICAL SECTION
7      wantQ = false;
8      // non-critical section
9    }
10 }
```

This solution enjoys mutual exclusion and freedom from starvation, however it does not avoid livelock.

> Contention
>
> - Naive back-out: either back out if they discover they are contending with the other
> - Dekker's algorithm: take turns
> - Peterson's algorithm: give priority to the first one that wanted access

Attempt IV - Naive back-out

```
1  boolean wantP = false;
2  boolean wantQ = false;
3
4  Thread.start {  //
5    while (true) {
6      // non-critical section
7      wantP = true;
8      while wantQ {
9        wantP = false;
10       wantP = true;
11     }
12     // CRITICAL SECTION
13     wantP = false;
14     // non-critical section
15   }
16 }
```

```
1  Thread.start {  // Q
2    while (true) {
3      // non-critical section
4      wantQ = true;
5      while wantP {
6        wantQ = false;
7        wantQ = true;
8      }
9      // CRITICAL SECTION
10     wantQ = false;
11     // non-critical section
12   }
13 }
```

This solution enjoys mutual exclusion and absence of livelock however, it is not free from starvation since there is an interleaving where P can enter and exit the critical section but Q will repeatedly try and fail.

Dekker's algorithm (1965) - Combines attempts I & IV

`Right to insist on entering` is passed between the two processes.

```
1  int turn = 1;
2  boolean wantP = false;
3  boolean wantQ = false;
4
5  Thread.start  {  // P
6    while (true) {
7      // non-critical section
8      wantP = true;
9      while wantQ
10       if (turn == 2) {
11         wantP = false;
12         await (turn==1);
13         wantP = true;
14       }
15     // CRITICAL SECTION
16     turn = 2;
17     wantP = false;
18     // non-critical section
19   }
20 }
```

```
1  Thread.start  {  // Q
2    while (true) {
3      // non-critical section
4      wantQ = true;
5      while wantP {
6        if (turn == 1) {
7          wantQ = false;
8          await (turn==2);
9          wantQ = true;
10       }
11     // CRITICAL SECTION
12     turn = 1;
13     wantQ = false;
14     // non-critical section
15   }
16 }
```

Peterson's algorithm - (1981)

Similar to Dekker except that if both want access, priority is givento the first one that wanted to access.

```
1  int last = 1;
2  boolean wantP = false;
3  boolean wantQ = false;
4
5  Thread.start  {  // P
6    while (true) {
7      // non-critical section
8      wantP = true;
9      last = 1;
10     await !wantQ or last==2;
11     // CRITICAL SECTION
12     wantP = false;
13     // non-critical section
14   }
15 }
```

```
1  Thread.start  {  // Q
2    while (true) {
3      // non-critical section
4      wantQ = true;
5      last = 2;
6      await !wantP or last==1;
7      // CRITICAL SECTION
8      wantQ = false;
9      // non-critical section
10   }
11 }
```

Complex Atomic Options

Solving the MEP using atomic load and store is not an easy problem to solve. This difficulty disappears if we allow more complicated atomic operations.

> Also known as `read-modify-write (RMW)` operations.

Specific atomic operations are provided by hardware to allow a fixed number of instructions to be executed atomically.

Test and Set

```
atomic boolean TestAndSet(ref) {
  result = ref.value; // reads the value before it changes it
  ref.value = true;   // changes it to true
  return result;      // returns the previously read value
}
```

Revisiting our example:

```
Ref shared = new Ref();
shared.value = false;

1  Thread.start {  //P
2    while (true) {
3      // non-critical section
4      while TestAndSet(shared);
5      // CRITICAL SECTION
6      shared.value = false;
7      // non-critical section
8    }
9  }
```

```
1  Thread.start {  //Q
2    while (true) {
3      // non-critical section
4      while TestAndSet(shared);
5      // CRITICAL SECTION
6      shared.value = false;
7      // non-critical section
8    }
9  }
```

Sepaphores I

States of a process

- **Inactive**:
- **Ready**: Code is not running but it is places in a "queue" of ready processes. From here the scheduler chooses code and runs it.
- **Running**: Code is being executed.
- **Completed**: Execution is finished.
- **Blocked**: A process needs access to some resource that is not currently available.

What is a semaphore?

A sepaphore is an Abstract Data Type with:

- Atomic operations:
  - Aqure (or wait)
  - release (or signal)
- Data fields:
  - permissions: non-negative integer
  - processes: a set of processes

Aquire

```
1  atomic acquire() {
2    currentThread = Thread.currentThread();
3    if (permissions > 0) {
4      permissions--;
5    } else {
6      processes.add(thread);
7      currentThread.state = BLOCKED;
8    }
9  }
```

Release

```
1  atomic release() {
2    if (processes.empty()) {
3      permissions++;
4    } else {
5      readyThread = processes.removeAny();
6      readyThread.state = READY;
7    }
8  }
```

Mutual Exclusion Using Semaphores

The MEP for two processes becomes trivial If we use a `mutex` .

- Entry protocol: `mutex.aquire();`
- Exit protocol: `mutex.release();`

```
1  Semaphore mutex = new Semaphore(1);

1  Thread.start {  //P
2    // non-critical section
3    mutex.acquire();
4    // CRITICAL SECTION
5    mutex.release();
6    // non-critical section
7  }
```

```
1  Thread.start {  //Q
2    // non-critical section
3    mutex.acquire();
4    // CRITICAL SECTION
5    mutex.release();
6    // non-critical section
7  }
```

This solution does not use busy waiting: a process that blocks in the acquire goes into the `BLOCKED` state and only returns to the `READY` state once it is given permission to do so.

Semaphores in Java

Class `Semaphore` in `java.util.concurrent`

```
import java.util.concurrent.Semaphore;
```

```
/** Creates a semaphore with the given number of permits */
Semaphore(int permits)

/** Acquires a permit from this Semaphore, blocking until  one is available */
void acquire()

/** Releases a permit, returning it to the semaphore */
void release()
```

> Example
>
> ```
> Semaphore mutex = new Semaphore(1);
> mutex.acquire();
> // Critical section
> mutex.release();
> ```

Semaphore Invariants

Let `k` be the initial value of the `permissions` field of a semaphore `s`

1. permissions ≥ 0
2. permissions = k + #releases − #acquires

where:

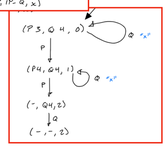
- `#releases` is the number of `s.release()` statements executed
- `#acquire` is the number of `s.acquire()` statements executed
- A blocked process is considered not to have executed an acquire operation.

Consider the program:

```
1 int x = 0;
2 Thread.start { //P          Thread.start { //Q
3   x = x + 1;                 while (x < 2)
4   x = x + 1;                   print(x);
5 }                          }
```

Construct the transition system for this algorithm.

State format : (P.P, P.Q, x)



Mutual Exclusion

1. Atomicity Assumptions on Statements
   - Atomic Operation: An operation is atomic if it cannot be interleaved at a lower level of abstraction
   - For this course: every single-line statement is going to be atomic
     - EX: counter = counter + 1;
   - To simulate something being non-atomic:
     - counter = counter + 1;
       Decomposed to:
       temp = counter + 1;
       counter = temp;
   - A var is a critical ref if:
     1. it is assigned in P and occurs in another process Q, or
     2. it is read in P and is assigned in another process Q
   - Limited Critical Reference (LCR) is satisfied if every statement contains at most one critical reference
   - Concurrent programs that satisfy the LCR restriction yield the same set of behaviors whether the statements are considered atomic or are compiled to a machine architecture with atomic load and store
     - Outputs from writing the transition states and compiling the code will not differ for whether it's atomic or not
2. Race Condition
   - Race Condition arises if two or more threads access the same variables or objects concurrently and at least one does updates
     - EX: order of read and write operations

Semaphores II

Patterns based on semeaphores

A **semaphore** is an abstract data type with two operations (*aquire and release*). It can be used to solve the mutual exclusion problem and to synchronize cooperative threads.

Producers/consumers

This is a common pattern of interaction which must cater for a difference in speed between each party.

Unbounded Buffer

The producer can work freely and the consumer must wait for the producer to produce.

Bounded Buffer

The producer must wait whan the buffer is full and the consuemr must wait for the producer to produce.

Readers/writers

There are shared resources between two types of threads.

- **Readers**: access the resources without modifying it (can access simultaneously)
- **Writers**: access the resource and may modify it (at most one at any given time)

Properties a Solution Should Possess

- Each read/write operation should occur inside the critical region
- Must guarantee mutual exclusion between the writers
- Must allow multiple readers to execute inside the critical region simultaneously

First Solution: *Priority Readers*

- One semaphore for controlling write access
- Before writing, the permission must be obtained and then released when done
- The first reader must "steal" the permission to write and the last one must return it
  - We must count the number of readers inside the critical section
  - This must be done inside its own critical section

Second Solution: *Priority Writers*

- The readers can *potentially* lock out all the writers
  - We need to count the number of writers that are waiting
  - Also, this counter requires its own critical section
- Before reading the readers must obtain a permission to do so

Monitors

We've seen that `semaphores` are and efficient tool to solve synchronization problems. Semaphores are elegant and efficient for solving problems in concurrent programs however, they are low-level constructs since they are not structured. Monitors will provide *synchronization by encapsulation*.

A monitor contains a set of operations encapsulated in modules and unique lock that ensures mutual exclusion to all operations in the monitor. Additionally special variables called condition variables, that are used to program conditional synchronization.

Counter Example

With Semaphores:

```
class Counter {
    private cnt = 0;
    private Semaphore mutex = new Semaphore(1);

    public void inc() {
        mutex.aquire();
        cnt++;
        mutex.release();
    }

    public void dec() {
        mutex.aquire();
        cnt--;
        mutex.release();
    }
}
```

With Monitors:

```
monitor Counter {
    private cnt = 0;

    public void inc() {
        cnt++;
    }

    public void dec() {
        cnt--;
    }
}
```

Each object has its own lock called `intrinsic` or `monitor` lock. It also has its own `wait-set` (more on this later). The class also has its own lock but it is not used in this example.

Apart from the lock, there are `conditional variables` associated with the monitor.

- Operations:
  - `Cond.wait()` : **always** blocks the process and places it in the waiting queue of the variable `Cond` . When it blocks, it releases the mutex on the monitor.
  - `Cond.notify()` : unblocks the first process in the waiting queue of the variableCondand sets it to the *ready* state. If there are no processes in the waiting queue, it has no effect.
  - `Cond.empty()` : checks if waiting queue ofCondis empty or not
- A queue of blocked processes

> Example: Buffer of Size 1
>
> ```
> monitor Buffer {
>
>     private Object buffer = null; // shared buffer
>     private Condition full;       // wait until space available
>     private Condition empty;      // wait until buffer available
>
>     public Object consume() {
>         while (buffer == null)
>             full.wait();
>         Object aux = buffer;
>         buffer = null;
>         empty.notify();
>         return aux;
>     }
>
>     public void produce(Object o) {
>         while (buffer != null)
>             empty.wait();
>         buffer = o;
>         full.notify();
>     }
>
> }
> ```

Wait

Blocks the process currently executing and associates it with a variable's queue. Upon blocking, it frees the `lock` allowing the entry of other processes.

Notify

Two strategies:

- Notify and Urgent Wait: `E < N < W` (classical monitors)
- Notify and Continue: `E = W < N` (Java ≈ We focus on this one)

where the letters denote the precedence of:

- `N` : notifying processes
- `W` : waiting processes
- `E` : processes blocked on entry

> Example: Monitor that Defines a Semaphore
>
> ```
> monitor Semaphore {
>
>     private Condition nonZero;
>     private int permissions;
>
>     public Semaphore(int n) {
>         this.permissions = n;
>     }
>
>     public void acquire() {
>         while (permissions == 0)
>             nonZero.wait ();
>         permissions--;
>     }
>
>     public void release() {
>         permissions++;
>         nonZero.notifyAll();
>     }
>
> }
> ```

Readers/Writers

```
monitor RW {

    int readers, writers = 0;
    Condition okToRead, okToWrite;

    public void StartRead() {
        while (writers != 0 || !okToWrite.empty()) {
            okToRead.wait();
        }
        readers++;
    }

    public void EndRead() {
        readers--;
        if (readers == 0) {
            okToWrite.notify();
        }
    }

    public void StartWrite() {
        while (writers != 0 || reader != 0) {
            okToWrite.wait();
        }
        writers = 1;
    }

    public void EndWrite() {
        writers = 0;
        okToWrite.signal();
        okToRead.signalAll();
    }
}
```

Upholds the readers–writers invariant, however it gives priority to readers over writers because:

- new readers can enter the moniotr without waiting as long as a reader is active
- waiting writers have to wait until the last reader calls `endRead` and signals `okToWrite`
- as long as readers keep arriving and queuing for entering the monitor, the waiting writers will never execute.

Fair Solution for Writers:

The problems above can be resolved by checking if there are waiting writers. *Creating a solution without deadlock that is fair for both readers and writers is particularly difficult*.

```
monitor RW {

    int readers, writers, writersWaiting = 0;
    Condition okToRead, okToWrite;

    public void StartRead() {
        // Check for waiting writers
        while (writers != 0 || writersWaiting != 0) {
            okToRead.wait();
        }
        readers++;
    }

    public void EndRead() {
        readers--;
        if (readers == 0) {
            okToWrite.notify();
        }
    }

    public void StartWrite() {
        while (writers != 0 || reader != 0) {
            writersWaiting++;
            okToWrite.wait();
            writersWaiting--;
        }
        writers = 1;
    }

    public void EndWrite() {
        writers = 0;
        okToWrite.signal();
        okToRead.signalAll();
    }
}
```

Concurrency is study of interacting comp. progs which share resources and run concurrently

Assumptions:

- No shared vars in critical and non-critical sections
- Crit section terminates
- Scheduler is weakly fair (A process waiting will eventually be able to run)

MEP

- Mutex : One thread at a time
- Absence of deadlock
- Free from starvation

Styles: Break symmetry take turns (+ + -) (Could loop in non–crit)

states:

- inactive
- ready
- running
- blocked
- completed

R/W Service queue solution If first reader acquire resource, otherwise read because another reader is already in

Barrier Synchonization

Creates a *one–time–use* barrier for synchronizing threads such that *N* threads must reach the barrier before they can continue.

```
monitor Barrier {

    final int N = 3;
    int count = 0;
    Condition readyToPass;

    public void pass() {
        if (c < N) {
            count++;
            while(count < N) {
                readyToPass.wait()
            }
            readyToPass.signal();
        }
    }
```

```
Coding tips:
Remember that when ur coding a semaphore you can make the permission greater than the set value

You can use semaphores .release() on a 0 perm semaphore to ensure that something acquires first before the release this works because it either increments the permission counter or it passes the baton

It is syntactically correct to utilize a int as a param to release or acquire to perform it multiple times


readers/writers

if a reader is in then disable the writer
if a writer is in the disable the readers
the last writer/reader that is in enables the next
This is a trivial unfair solution some ways to introduce fairness is wrapping the protocol in a mutex which ensures that the readers will get the next usage of entrance
Or use a queue like structure to make sure readers get next access
small example
```

```
monitor Pizzeria {
        // Data fields
        private int small_pizzas = 0;
        private int large_pizzas = 0;
        private Condition okToBuyLargePizza;
        private Condition okToBuySmallPizza;


        purchaseLargePizza() {
                while(large_pizzas == 0 && small_pizzas < 2) {
                        okToBuyLargePizza.wait();
                }

                if (large_pizzas != 0) { // to be fair to the small pizzas
                        large_pizzas--;
                } else if (large_pizzas == 0) {
                        small_pizzas -= 2;
                }
        }

        purchaseSmallPizza() {
                while (small_pizzas == 0) {
                        okToBuySmallPizza.wait();
                }
                small_pizzas--;
        }

        bakeSmallPizza() {
                small_pizzas++;
                okToBuySmallPizza.signal();
                if (small_pizzas > 1) {
                        okToBuyLargePizza.signal();
                }
        }

        bakeLargePizza() {
                large_pizzas++;
                okToBuyLargePizza.signal();
        }
}
```

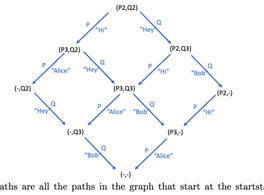
Exercise Booklet 1: Paths

Note: For this booklet you must assume that assignment is atomic and that the scheduler is fair. Also, you may use "?" for the value of uninitialized variables. Solutions to selected exercises (◊) are provided at the end of this document. Important: You should first try solving them before looking at the solutions. You will otherwise learn **nothing**.

Exercise 1. (◊) Assume that the **print** command is atomic. Build the transition system and then exhibit all possible paths of execution of the following program:

```
Thread.start { //P
  print("Hi");
  print("Alice");
}
Thread.start { // Q
  print("Hey");
  print("Bob");
}
```

Answer to exercise 1



The execution paths are all the paths in the graph that start at the startstate $(P2, Q2)$ and end in the endstate $(-, -)$.

Exercise 2. Draw the transition system for the following programs. What values can x take at the end of the execution?

1.

```
int x = 0;
Thread.start { //P
  int local = x;
  local = local + 1;
  x = local;
}
Thread.start { //Q
  int local = x;
  local = local + 1;
  x = local;
}
```

2.

```
int x = 0;
Thread.start { //P
  x = x + 1;
}
Thread.start { //Q
  x = x + 1;
}
```

Exercise 3. (◊) Given the following program:

```
int x = 0;
int y = 0;
Thread.start { //P2
  y = x + 1;
}
Thread.start { //Q
  x = y + 1;
}
```

1. Show an execution path such that at the end $x = 2$ and $y = 1$.
2. Is there a path s.t. $x = y = 1$? Justify your answer. What would happen if the assumption on atomicity of assignment is dropped?

Answer to exercise 3

1. State format (IP_P, IP_Q, x, y). Path: $(P2, Q2, 0, 0) \to (-, Q2, 0, 1) \to (-, -, 2, 1)$
2. There are not paths that result in x and y both holding one. If the assumption on atomicity of assignment is dropped, then there would be such a path.

Exercise 7. (◊) Consider the following two threads:

```
Thread.start { // P
  while (true) {
    print("A");
  }
}
Thread.start { // Q
  while (true) {
    print("B");
  }
}
```

1. Use semaphores to guarantee that at all times the number of A's and B's differs at most in 1.
2. Modify the solution so that the only possible output is ABABABABAB...

Answer of exercise 7

Item 1.

```
Semaphore allowA = new Semaphore (1);
Semaphore allowB = new Semaphore (1);

Thread.start { // P
  while (true) {
    allowA.acquire();
    print("A");
    allowB.release();
  }
}

Thread.start { // Q
  while (true) {
    allowB.acquire();
    print("B");
    allowA.release();
  }
}
```

What happens if we initialize both semaphores any $k > 0$?
Item 2. Have `allowB` be initialized with 0 permits.

Exercise 1. (◊) On Fridays the bar is usually full of men, therefore the owners would like to implement an access control mechanism in which one man can enter for every two women.

```
Semaphore ticket = new Semaphore(0);
Semaphore mutex = new Semaphore(1);

thread Men: {
    mutex.acquire();
    ticket.acquire();
    ticket.acquire();
    mutex.release();
}

thread woman {
    ticket.release();
}
```

Exercise 4. (◊) We wish to implement a `three-way sequencer` using monitors in order to coordinate N threads. A three-way sequencer provides the following operations `first`, `second`, `third`. The idea is that each of the threads can invoke any of these operations. The sequencer will alternate cyclically the execution of first, then second, and finally third.

Answer to exercise 4

```
monitor TWS{
  int state = 1;
  condition first, second, third;

  void first() {
      while (state != 1)
          first.wait();
      state = 2;
      second.signal();
  }

  void second() {
      while (state != 2)
          second.wait();
      state = 3;
      third.signal();
  }

  void third() {
      while (state != 3)
          third.wait();
      state = 1;
      first.signal();
  }
```

Exercise 8. (◊) Model a vehicle crossing between two endpoints. Since the crossing is narrow, it does not allow for vehicles to travel in opposite directions. You solution must allow multiple vehicles to use the crossing so long as they are travelling in the same direction. Use a thread `Vehicle(myEndpoint)` to model a vehicle located at endpoint `myEndPoint` (either 0 or 1).

Answer to exercise 8

```
global Semaphore useCrossing = new Semaphore(1); //mutex
global Semaphore[] endpointMutex = new Semaphore[2];
endpointMutex[0] = new Semaphore(1, true); // Strong: vehicles cross
endpointMutex[1] = new Semaphore(1, true); // on FCFS order
global int[] noOfCarsCrossing = {0,0};

thread Auto(myEndpoint) : {
  endpointMutex[myEndpoint].acquire();
  if (noOfCarsCrossing[myEndpoint] == 0)
    useCrossing.acquire();
  noOfCarsCrossing[myEndpoint]++;
  endpointMutex[myEndpoint].release();

  //Cross crossing

  endpointMutex[myEndpoint].acquire();
  noOfCarsCrossing[myEndpoint]--;
  if (noOfCarsCrossing[myEndpoint] == 0)
    useCrossing.release();
  endpointMutex[myEndpoint].release();
}
```

The Mutual Exclusion Problem

Guarantee that:

1. Mutex: At any point in time, there is at most one thread in the critical section
2. Absence of livelock: If various threads try to enter the critical section, at least one of them will succeed
3. Free from starvation: A thread trying to enter its critical section will eventually be able to do so

Processes should:

1. either back out if they discover they are contending with the other (Naive back-out);
2. take turns (Dekker's algorithm);
3. give priority to the first one that wanted access (Peterson's algorithm).

Peterson's Algorithm (1981)

```
global int last = 1;
global boolean wantP = false;
global boolean wantQ = false;

thread P: {                          thread Q: {
  while (true) {                       while (true) {
    // non-critical section              // non-critical section
    wantP = true;                        wantQ = true;
    last = 1;                            last = 2;
    await !wantQ or last==2;             await !wantP or last==1;
    // CRITICAL SECTION                  // CRITICAL SECTION
    wantP = false;                       wantQ = false;
    // non-critical section              // non-critical section
  }                                    }
}                                    }
```

Similar to Dekker except that if both want access, priority is given to the first one that wanted to access

Dekker's Algorithm (I + IV)

```
global int turn = 1;
global boolean wantP = false;
global boolean wantQ = false;

thread P: {                          thread Q: {
  while (true) {                       while (true) {
    // non-critical section              // non-critical section
    wantP = true;                        wantQ = true;
    while wantQ                          while wantP
      if (turn == 2) {                     if (turn == 1) {
        wantP = false;                       wantQ = false;
        await (turn==1);                     await (turn==2);
        wantP = true;                        wantQ = true;
      }                                    }
    // CRITICAL SECTION                  // CRITICAL SECTION
    turn = 2;                            turn = 1;
    wantP = false;                       wantQ = false;
    // non-critical section              // non-critical section
  }                                    }
}                                    }
```

Right to insist on entering is passed between the two processes

Exercise Booklet 5: Semaphores (cont)

Solutions to selected exercises (◊) are provided at the end of this document. Important: You should first try solving them before looking at the solutions. You will otherwise learn **nothing**.

Exercise 1. (◊) On Fridays the bar is usually full of Jets fans. Since the owners are Patriots fans they would like to implement an access control mechanism in which one Jets fan can enter for every two Patriots fans.

1. Implement such a mechanism, assuming that Jets fans will have to wait indefinitely if no Patriots fans arrive. You may assume, to simplify matters, that once fans go in, they never leave the bar.
2. Modify the solution assuming that, after a certain hour, everybody is allowed to enter (those that are waiting outside and those that yet to arrive). For that there is a thread that will invoke, when the time comes, the operation `itGotLate`. You may assume that the code for this thread is already given for you, the only thing that you must do is define the behavior of `itGotLate` and modify the threads that model the Jets and Patriots fans.

Answer to exercise 1

Groovy

```
import java.util.concurrent.Semaphore;

Semaphore ticket = new Semaphore(0);
Semaphore mutex = new Semaphore(1);

20.times{
    Thread.start { // Patriots
      ticket.release();
    }
}

20.times {
Thread.start { // Jets
    mutex.acquire();
    ticket.acquire();
    ticket.acquire();
    mutex.release();
}
}
```

```
import java.util.concurrent.Semaphore;

Semaphore ticket = new Semaphore(0);
Semaphore mutex = new Semaphore(1);
boolean itGotLate = false;

Thread.start { // Jets
    mutex.acquire();
    if (!itGotLate) {
      ticket.acquire();
      ticket.acquire();
    }
    mutex.release();
}

Thread.start { // Patriots
    ticket.release();
}

def itGotLate() {
    itGotLate=true;
    ticket.release();
    ticket.release();
}
```

```
Semaphore buttons = new Semaphore(2);
Semaphore mutex = new Semaphore(1);

thread Client {
        buttons.acquire();
        //coffee boi
        buttons.release();
}

thread Employee{
        while(true){
                mutex.acquire();
                repeat(2) { buttons.acquire(); }
                mutex.release();

                //refill
                repeat(2) { buttons.release(); }
        }
}
```

```
monitor PizzaShop {
        int smallPizza = 0;
        int largePizza = 0;
        condition waitForSmallPizza;
        condition waitForLargePizza;

        bakeSmallPizza() {
                smallPizza++;
                waitForSmallPizza.signal();
                waitForLargePizza.signal();
        }

        bakeLargePizza(){
                largePizza++;
                waitForLargePizza.signal();
        }

        buySmallPizza(){
                while(smallPizza==0){
                        waitForSmallPizza.wait();
                }
                smallPizza--;
        }

        buyLargePizza(){
                while(large==0 && small<2){
                        waitForLargePizza.wait();
                }
                if(largePizza > 0){
                        largePizza--;
                }else{
                        smallPizza-=2;
                }
        }
}
```

```
monitor PizzaTime {
        int smallPizza;
        int largePizza;

        Condition waitForSmallPizza;
        Condition waitForLargePizza;

        bakeSmallPizza(){
                smallPizza++;
                waitForSmallPizza.signal();
                waitForLargePizza.signal();
        }

        bakeLargePizza(){
                largePizza++;
                waitForLargePizza.signal();
        }

        orderSmallPizza(){
                while(smallPizza == 0){
                        waitForSmallPizza.wait();
                }
                smallPizza--;
        }

        orderLargePizza(){
                while(largePizza == 0 && smallPizza<2){
                        waitForLargePizza.wait();
                }

                if(largePizza){
                        largePizza--;
                }else{
                        smallPizza-=2;
                }
```

```
monitor barrierMonitor {
        Condition con;
        int count = 0;
        //N
        wait(){
                count++;
                while(count < N){
                        con.wait();
                }
                con.signalAll();
        }
}
```