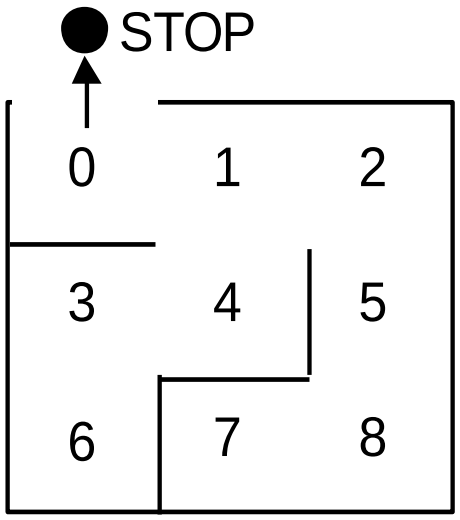
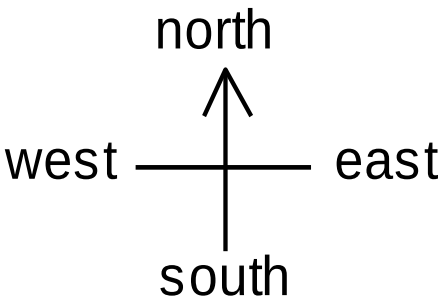
CS3211 Parallel and Concurrent Programming – Week 8 tutorial

Sample Exercises:
[Please conduct these as an interactive discussion, rather than an evaluation. Please also make it clear to the students that they are not being evaluated for their performance in these exercises, so that they are not afraid to make mistakes while answering.]

1. Following a maze, similar to what we discussed in class. Write process equations, such that using deadlock analysis (path to a state with no outgoing action) we can find the shortest path out of the maze.





Answer:

```
MAZE(Start) = P[Start],
P[0] = (north->STOP|east->P[1]),
P[1] = (east ->P[2]|south->P[4]|west->P[0]),
P[2] = (south->P[5]|west ->P[1]),
P[3] = (east ->P[4]|south->P[6]),
P[4] = (north->P[1]|west ->P[3]),
P[5] = (north->P[2]|south->P[8]),
P[6] = (north->P[3]),
P[7] = (east ->P[8]),
P[8] = (north->P[5]|west->P[7]).

GETOUT = MAZE(7).
```

2. It is possible for the following system to deadlock. Explain how this deadlock occurs.

   Alice = (call.bob -> wait.chris -> Alice).
   Bob = (call.chris -> wait.alice -> Bob).
   Chris = (call.alice -> wait.bob -> Chris).

   System = (Alice || Bob || Chris) / {call/wait}

Answer: Show the wait-for-cycle among Alice, Chris and Bob. The key is to identify what are the shared actions.

3. Following is an attempt to "fix" the system of Question 2.

```
AliceT = (call.bob   -> wait.chris -> AliceT
         |timeout.alice -> wait.chris -> AliceT).
BobT   = (call.chris -> wait.alice -> BobT
         |timeout.bob -> wait.alice -> BobT).
ChrisT = (call.alice -> wait.bob   -> ChrisT
         |timeout.chris -> wait.bob -> ChrisT).

System-fixed = (AliceT || BobT || ChrisT) /{call/wait}.
```

Is System-fixed really fixed? Are there any deadlocks?

Answer:
```
Trace to DEADLOCK:
        timeout.alice
        timeout.bob
        timeout.chris
```

4. A single-slot buffer may be modeled by
ONEBUF = (put -> get -> ONEBUF)

In an earlier tutorial we had programmed a Java class, OneBuf, that implements this one-slot buffer as a monitor. Replace the condition synchronization above by using two semaphore to control access to the one slot buffer.

/* This was the monitor based encoding */

```
public class OneBuf {
        private int buf;
        private boolean empty = true;

        public synchronized void put(int x) throws InterruptedException{
                while(!empty) wait();
                buf = x;
                empty = false;
                notifyAll();
        }

        public synchronized int get() throws InterruptedException{
                while(empty) wait();
                empty = true;
                notifyAll();
                return buf;
        }
}
```

Answer:

```
/* ONEBUF = (put -> get -> ONEBUF).  */

/* java Implementation using Semaphores

public class OneBuf {
    int slot = 0;   /*  use slot == 0 to denote unassigned */
    Semaphore empty = new Semaphore(1);
    Semaphore full  = new Semaphore(0);

    public void put(int x) throws InterruptedException {
        empty.down();
        synchronized(this){
                    slot = x;}
        full.up();
    }

      public Object get () throws InterruptedException {
        full.down();
        synchronized(this){
              int  x = slot;
              slot = 0; }
        empty.up();
        return x;
    }
}
```

Instead of using integer slot above, we could have defined it as an object. In that case, it should have been initialized to null (to denote that the slot is initially empty).