```
// A is printed before F
Semaphore Amutex = new Semaphore(1);
Semaphore Fmutex = new Semaphore(0);
thread P:{
          Amutex.acquire();
          print('A');
          Fmutex.release();
          Amutex.release();
          print('B');
          print('C');
}

thread Q:{
          print('E');
          Fmutex.acquire();
          print('F');
          print('G');
}
```

```
// using Semaphore to have an output of RACES
Semaphore Rmutex = new Semaphore(1);
Semaphore Amutex = new Semaphore(0);
Semaphore Emutex = new Semaphore(0);
thread P:{
          Amutex.acquire();
          print('A');
          print('C');
          Emutex.release();
}

thread Q:{
          Rmutex.acquire();
          print('R');
          Amutex.release();
          Rmutex.release();
          Emutex.acquire();
          print('E');
          print('S');
}
```

```
// using Semaphore to have an output of
// R I O OK OK OK
Semaphore Imutex = new Semaphore(0);
Semaphore Omutex = new Semaphore(0);
Semaphore OKmutex = new Semaphore(0);
thread P:{
          print('R');
          Imutex.release();
          OKmutex.acquire();
          print("OK");
}

thread Q:{
          Imutex.acquire();
          print('I');
          Omutex.release();
          OKmutex.acquire();
          print("OK");
}

thread X:{
          Omutex.acquire();
          print('O');
          OKmutex.release();
          OKmutex.release();
          print("OK");
}
```

```
// n toilets -> free from deadlock
// women is same as men
Semaphore mutex = new Semaphore(1);
Semaphore wUse = new Semaphore(1);
Semaphore mUse = new Semaphore(1);
int count = 0;
thread Men: {
          mutex.acquire();
          mUse.acquire();
          count++;
          if(count == 1){
                    wUse.acquire();
          }
          mutex.release();
          mUse.release();
          mUse.acquire();
          count--;
          if(count==0){
                    wUse.release();
          }
          mUse.release();
}
```

```
// men can enter for every two women
Semaphore wEnter = new Semaphore(0);
Semaphore mEnter = new Semaphore(1);
boolean late = false;

itGotLate() {
          late = true;
          while(true){
                    ticket.release();
                    ticket.release();
          }
}
thread Men: {
          if(!itGotLate){
                    mEnter.acquire();
                    wEnter.acquire();
                    wEnter.acquire();
                    mEnter.release();
          }
}

thread Women: {
          wEnter.release();
}
```

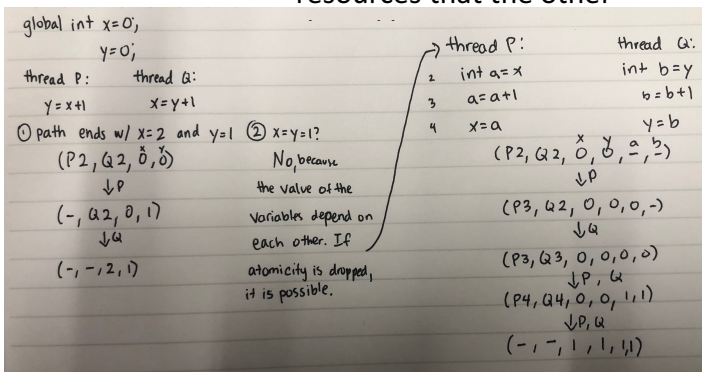
```
//ferry has N capacity
//It waits in one coast until it fills up and then automatically
//switches to the other coast.
//When it arrives at a coast, all the passengers get off and,
//only after all thepassengers have gotten off, all those that
//are waiting to board do so.

Semaphore[] permToBoard = { new Semaphore(0), new Semaphore(0) }
Semaphore[] permToExit = { new Semaphore(0), new Semaphore(0) }
Semaphore[] okToSail = { new Semaphore(0), new Semaphore(0) }

thread Passenger(i): {
          permToBoard[i].acquire();
          okToSail[i].release();
          permToExit[1-i].acquire();
          okToBoard[i].release();
}

thread Ferry: {
          i=0;
          repeat(N){
                    permToBoard[i].release();
          }
          repeat(N){
                    okToSail[i].acquire();
          }
          i = 1-i;
          repeat(N){
                    permToExit[i].release();
          }
          repeat(N){
                    okToBoard[i].release();
          }
```

```
/ Complete the }
/ so that informa-tion can be read only when the buffer is not empty.
/ You may assume that there is a method LinkedList.isEmpty()that determines
/ whether a linked-list is empty or not.
/ Also, if you need to add condition variables, then declare them just after the data field.
/ Extend the monitor with two new methods disableWrite() and enableWrite().
/ When disableWrite() is executed no new writes can be performed.
/ These writes can be resumed only after the enableWrite() method has been called.
/ The operation c.signalAll(), for c a condition variable, may be handy:
/ it puts all the processesin its associated queue in a READY state.
/ Also, you may need to modify operations read or write from the previous item.

onitor Buffer {
          private LinkedList data = new LinkedList ();
          bool disWrite = false;
          condition canRead, canWrite;
          public void write(Object o) {
                    while(disWrite){
                              canWrite.wait();
                    }
                    data.addFirst(o);
                    canRead.signal();
          }
          public Object read() {
                    if(data.isEmpty()) {
                              canRead.wait();
                    }
                    object ret_val = data.getLast ();
                    data.removeLast ();
                    return ret_val;
          }
          public void disableWrite(){
                    disWrite = true;
          }
          public void enableWrite(){
                    disWrite = false;
                    canWrite.signalAll();
          }
```

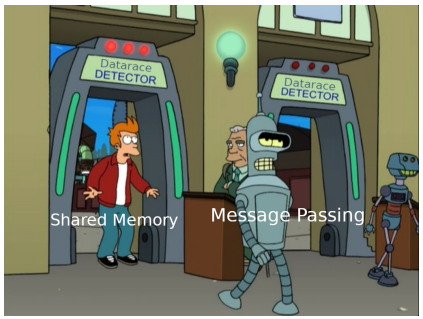
```
//A hotel serves coffee using a dispenser.
//The dispenser has two buttons, so it can serve two clients at the same time.
//Every now and then the dispenser is refilled by an employee.
//There are multiple employees but only one should refill the
//dispenser at any given time. Moreover, she must ensure there are no
//clients when she refills the dispenser.
//Model this scenario using semaphores. Model both the client and the employee.
Semaphore mutex = new Semaphore(1);
Semaphore button = new Semaphore(2);
thread Client: {
          button.acquire();
          button.release();
}

thread Employee: {
          mutex.acquire();
          button.acquire();
          button.acquire();
          refillDispenser();
          button.release();
          button.release();
          mutex.release();
}
```

Livelock – loops and never hits CS

Deadlock – race condition in which two threads need two resources that the other




```
monitor Stack: {
          private stack = new ArrayList();
          int size = 0;
          condition canPop;
          function push(i) {
                    arr[size] = i;
                    size++;
                    canPop.signal();
          }
          function pop(){
                    if(size == 0){
                              canPop.wait();
                    }
                    size--;
                    return arr[size+1]
          }
```



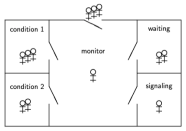
Readers/Writers

```
public void StartWrite() {
  if (writers != 0 or readers != 0) {
    OKtoWrite.wait();
  }
  writers = writers + 1;
}

public void EndWrite() {
  writers = writers - 1;
  if (OKtoRead.empty()) {
    OKtoWrite.signal();
  } else {
    OKtoRead.signal();
  };
 }
}
```

Readers/Writers

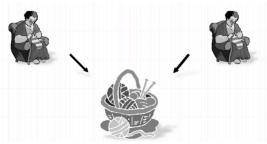
```
monitor RW {
  int readers = 0;
  int writers = 0;
  condition OKtoRead, OKtoWrite;

  public void StartRead() {
    if (writers != 0 or not OKtoWrite.empty()) {
      OKtoRead.wait();
    }
    readers = readers + 1;
    OKtoRead.signal();
  }

  public void EndRead {
    readers = readers - 1;
    if (readers==0) {
      OKtoWrite.signal();
    }
  }
  // continues
```

ignal and Continue: $E = W < S$ (Java)



- Process from S which executes the signal continues executio
- Process from W which is unblocked joins competition for th lock
- Problem: signaling process can modify the condition after it executed the signal

```
//We would like to model a control system for an automatic car wash.  Each car traverses three phases:  blast, rinse and dry.  Each of these
phases is executed by a machine.  All vehicles follow these three phases in that exact order.

Semaphore[] permToProcess = { new Semaphore(0), new Semaphore(0), new Semaphore(0) };
Semaphore[] doneProcessing = { new Semaphore(0) , new Semaphore(0) , new Semaphore(0) };
Semaphore[] permToEnter = { new Semaphore(1) , new Semaphore(1) , new Semaphore(1) };


thread Car : {
          permToEnter[0].acquire(); // See if it is ok to enter station 0
          advanceToNextStation(); // assuming no cars are in any station, and the first car is in a station before station 0 that does not
          permToProcess[0].release(); // Start processing on car in station 0
          doneProcess[0].acquire(); // Wait for the station thread to be done on station 0
          permToEnter[1].acquire(); // See if it is ok to enter station 1
          advanceToNextStation(); // Advance to station 1
          permToEnter[0].release(); // Allow cars into station 0

          permToProcess[1].release(); // Start processing on car in station 1
          doneProcess[1].acquire(); // Wait for the station thread to be done on station 1
          permToEnter[2].acquire(); // See if it is ok to enter station 2
          advanceToNextStation(); // Advance to station 2
          permToEnter[1].release(); // Allow cars into station 1

          permToProcess[2].release(); // Start processing on car in station 2
          doneProcess[2].acquire(); // Wait for the station thread to be done on station 2
          advanceToNextStation(); // Advance out of station 2: done with cars
          permToEnter[2].release(); // Allow cars into station 2
}

thread MachineAtStation (i): {
          while (true) {
                    permToProcess[i].acquire();
                    // process vehicle
                    doneProcessing[i].release();
          }
}
```



- Deadlock: each granny takes a needle and waits indefinitely until the other one has freed the one she has.
- Livelock: each granny takes a needle, sees that the other granny has the other needle and returns it (this repeats indefinitely).
- Starvation: one of the grannies always takes the needle before the other one.

ıttempt 0 – `flag` to indicate whether someone is in the CS

```
global boolean flag = false;

thread P: {                          thread Q: {
  // non-critical section              // non-critical section
  while (flag) {};                     while (flag) {};
  flag = true;                         flag = true;
  // CRITICAL SECTION                  // CRITICAL SECTION
  flag = false;                        flag = false;
  // non-critical section              // non-critical section
}                                    }
```

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

```
while (cond) {}   ⟶   await !cond
```

ritical Section

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

```
1 thread P: {                          1 thread Q: {
2   while(true) {                      2   while(true) {
3   // non-critical section            3   // non-critical section
4   entry to critical section;         4   entry to critical section
5   // CRITICAL SECTION                5   // CRITICAL SECTION
6   exit from critical section         6   exit from critical section
7   // non-critical section            7   // non-critical section
8   }                                  8   }
9 }                                    9 }
```

Race Condition

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

- Whether it happens depends on how threads are scheduled
- Once thread T1 starts doing something, it needs to "race" to finish it because if thread T2 looks at the shared variable before T1 is done, it may see something inconsistent
- Hard to detect

The Bakery Algorithm

```
global boolean[] choosing = new boolean[n]; // {false,..,}
global int[] number = new int[n]; // {0,0,...0}

thread {
  // non-critical section
  choosing[id] = true;
  number[id] = 1 + maximum(number);
  choosing[id] = false;
  for (all other processes j) {
    await !choosing[j];
    await (number[j] == 0) or (number[id]<<number[j]);
  }

  // CRITICAL SECTION

  number[id] = 0;
  // non-critical section
}
```

This algorithm solves the problem for n threads.

- Mutex: Yes
- Absence livelock: Yes
- Free from starvation: Yes

The Mutual Exclusion Problem (MEP)

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

Properties of Paths

A property is an assertion that is true for every possible path

- Safety: A path never reaches a "bad" state
- Liveness: Eventually, every path will reach a "good" state

Synchronization
Mechanism that restricts the possible paths of a concurrent program in order to ensure safety and liveness properties.

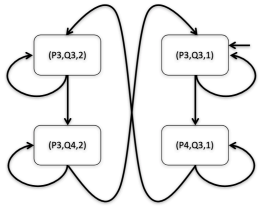
Atomicity Assumption on Assignments

Atomic Operation
An operation is atomic if it is executed until completion without interleaving statements from other processes

- Atomic operations are the smallest unit in which a path can be listed
- We assume throughout this course that assignment statements are atomic for scalar values (i.e. numbers, booleans, chars, etc.)
- Eg. `x=2` is atomic but `x=x+1` is not.

Proving the Desired Properties I

- Mutex: Holds if all accesible states do no contain a state of the form $(p4, q4, turn)$ for some value of turn



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

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

Absence of livelock (AoL) means that if two or more processes are trying to enter their CS, then at least one must eventually succeed

- If our processes could block when trying to access the CS, then we would speak of absence of deadlock: there are no states that are not endstates and that have no outgoing edges
- If processes cannot block (as is our case for now) we talk about absence of livelock rather than AoD. Livelock means that even though processes are not blocked (i.e. in a waiting state) they make no progress (i.e., there is no path leading to the CS)

The Semaphore Solution for the MEP ($k = 1$)

`#criticalSection`: number of processes in their critical sections

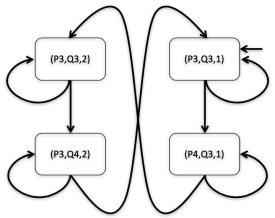
1. `#criticalSection` + `permissions` $= 1$
2. `#criticalSection` $=$ `#acquires` $-$ `#releases`

Item 1 guarantees:

- Mutual Exclusion (`#criticalSection` ≤ 1 since $0 \leq$ `permissions`)
- Absence of deadlock (it never happens that `permissions` $= 0$ and `#criticalSection` $= 0$)
- No starvation between two processes

Proving the Desired Properties II

- Absence of livelock (if two or more processes are trying to enter their CS, i.e. executing an await, then at least one must eventually succeed)
- Recall that the awaits are on line 3



Proving the Desired Properties III

Freedom from starvation (if any process is about to execute its preprotocol, then eventually it will succeed in entering the CS)

- Consider what happens if Q is trying to access its critical section but $turn = 1$ and P fails or loops in its NCS (*)

```
global int turn = 1;

1 thread P: {                          1 thread Q: {
2   while (true) {                     2   while (true) {
3     // non-critical section (*)      3     // non-critical section
4     await (turn==1);                 4     await (turn==2);
5     // CRITICAL SECTION              5     // CRITICAL SECTION
6     turn = 2;                        6     turn = 1;
7     // non-critical section          7     // non-critical section
8   }                                  8   }
9 }                                    9 }
```

Attempt III – Assessment

```
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;
    await !wantQ;                        await !wantP;
    // CRITICAL SECTION                  // CRITICAL SECTION
    wantP = false;                       wantQ = false;
    // non-critical section              // non-critical section
  }                                    }
}                                    }
```

Processes should:

1. either back out if they discover they are contending with t other (Naive back-out);
2. take turns (Dekker's algorithm);
3. give priority to the first one that wanted access (Peterson's algorithm).

Readers/Writers

- There are shared resources between two types of threads
  - Readers: access the resource without modifying it
  - Writers: access the resource and may modify it

- Mutual exclusion is too restrictive
  - Readers: can access simultaneously
  - Writers: at most one at any given time

```
non-atomic -> write is as assembly
      - write a temp (int a = x)
      - store a stamp (a = a + 1)
      - set the variable to the temp value (x = a)
atomic -> do it
```

Condition Variables

`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.signal()`

- Unblocks the first process in the waiting queue of the variable `Cond` and sets it to the READY state
- If there are no processes in the waiting queue, it has no effect.

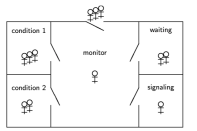
`Cond.empty()`

- Checks if waiting queue of `Cond` is empty or not

Example: Buffer of Size 1

```
1  monitor Buffer {
2    private Object buffer = null; // shared buffer
3    private condition full;  // wait until space available
4    private condition empty; // wait until buffer available
5
6    public Object consume() {
7      if (buffer == null)
8        full.wait();
9      Object aux = buffer;
10     buffer = null;
11     empty.signal();
12     return aux;
13   }
14
15   public void produce(Object o) {
16     if (buffer != null)
17       empty.wait();
18     buffer = o;
19     full.signal();
20   }
21
22 }
```

Signal



- Two strategies:
  - Signal and Urgent Wait: $E < S < W$ (classical monitc
  - Signal and Continue: $E = W < S$ (Java)

where the letters denote the precedence of

- S: signalling processes
- W: waiting processes
- E: processes blocked on entry

Signal and Urgent Wait (or Immediate Resumption Requirement – IRR)

$$E < S < W$$

- When a process blocked on a condition variable is signaled, it immediately begins executing ahead of the signaling process
  - Resumes at the instruction immediately following the call to wait that blocked it
  - No need not check the condition
- Rationale: Signaling process changed the state of the monitor so that the condition now holds
- Cons: signaling process unnecessarily delayed (unless signal is the last operation)

Monitor that Defines a Semaphore

```
1  monitor Semaphore {
2    private condition nonZero;
3    private int permissions;
4
5    public Semaphore(int n) {
6      this.permissions = n;
7    }
8
9    public void acquire() {
10     if (permissions == 0)
11       nonZero.wait();
12     permissions--;
13   }
14
15   public void release() {
16     permissions++;
17     nonZero.signal();
18   }
```

Dining Philosophers

```
1  monitor ForkMonitor {
2    int[] fork = {2,2,2,2,2};
3    condition[] OKtoEat; // 0-4
4
5    public void takeForks(integer i) {
6      if (fork[i] != 2) {
7        OKtoEat[i].wait();
8      }
9      fork[i+1] = fork[i+1] - 1;
10     fork[i-1] = fork[i-1] - 1;
11   }
12
13   public void releaseForks(integer i) {
14     fork[i+1] = fork[i+1] + 1;
15     fork[i-1] = fork[i-1] + 1;
16     if (fork[i+1] == 2) {
17       OKtoEat[i+1].signal();
18     }
19     if (fork[i-1] == 2) {
20       OKtoEat[i-1].signal();
21     }
```