Dead Lock

- If two threads are waiting for each other forever such type of infinite waiting is called dead lock

- Synchronized keyword is the only reason for dead lock situation hence while using synchronized keyword we have to take special care

- There is no resolution technique for dead lock but several prevention techniques are available

```
class A {
    d1(B b){
        b.last();
    }
    last(){
    }
}
```

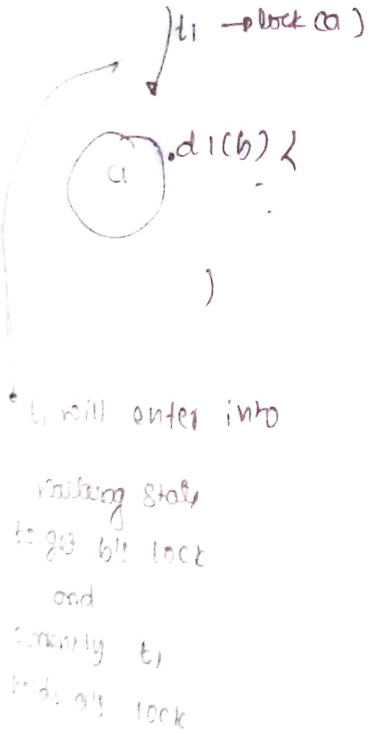
```
class B {
    d2(A a){
        a.last();
    }
    last(){
    }
}
```

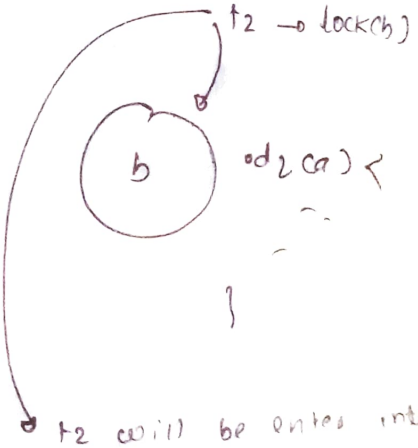
```
t1
a.d1(b){
}
```

```
t2
b.d2(a){
}
```

```
class A {

synchronized d1(B b){
    ;
    b.last();
}
last() {

}
```

```
class B {

synchronized d2(A a){
    ;
    a.last();
}
last() {
    :
    :
}
}
```





t1 will enter into waiting state to get b's lock and currently t1 holds a's lock

t2 will be enter into waiting state to get a's lock and t2 currently holds lock b's lock

```
class A {
public synchronized void d1(B b) {
    Sop("Thread1 starts execution
        of d1 method");
    try {
      Thread.sleep(5000);
    }
    catch(InterruptedException e){ }
        Sop("Thread1 trying to call
            B's last() method");
        b.last();
    }
    public synchronized void last() {
      Sop("Inside A last() method");
    }
}
```

```
class B {
  public synchronized void d2(A a){
    Sop("Thread2 starts execution
        of d2 method");
    try {
      Thread.sleep(5000);
    }
    catch(InterruptedException e) { }
      Sop("Thread2 trying to call
          A's last() method")
      a.last();
    }
  public synchronized void last() {
      Sop("Inside B last() method");
    }
}
```

```
class MyThread extends Thread {
    A a = new A();
    B b = new B();
    public void run() {
        this.start();
        a.d1(b);  ------> main thread
    }
    public void run() {
        b.d2(a);  ------> child thread
    }
public static void main(String[] args) {
    MyThread t = new MyThread();
    t.run();
}
}
```

output : Thread 1 starts execution of d_1.

Thread 2 starts execution of d_2

Thread 2 trying to call A's last method.

Thread 1 trying to call B's last meth

—

deadlock occurs

In the previous problem / program

If we remove atleast one synchronized keyword Then the program won't enters into deadlock

Hence synchronized keyword is only reason for deadlock situation due to this while using synchronized keyword we have to take special care

Deadlock V/s Starvation

Long waiting of a thread where waiting never ends is called deadlock.

where as long waiting of a thread where waiting ends at certain point is called Starvation

ex. low priority thread has to wait until completing all high priority thread, it may be long waiting but ends at certain point, ~~it is called~~ which is nothing but Starvation

Daemon threads

- GC
- Attach Listner
- Signal Dispatcher

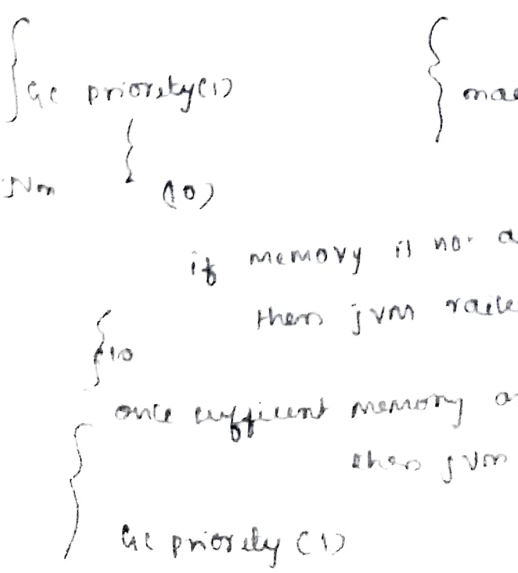
The threads which are executing in the background are called daemon threads

ex
- Garbage collector
- Signal Dispatcher
- Attach Listner

etc

The main objective of Daemon threads is to provide support for Non-Daemon threads (main thread)

ex: If main thread runs with low memory then JVM runs GC to destroy useless objects so that number of bytes of free memory will be improved. With this free memory main thread can continue its execution



Usually daemon threads having low priority, but based on our requirements, daemon threads can run with high priority also

We can check daemon nature of a thread by using isDaemon method of thread class

- Public boolean isDaemon()

We can change daemon nature of a thread by using setDaemon method.

- public void setDaemon(boolean b)

But changing daemon nature is possible before starting of a thread only,

After starting thread if we are trying to change daemon nature then we will get RuntimeException saying: IllegalThreadStateException

Default Nature of Thread

By default main thread is always Non Daemon

and for all remaining threads daemon nature will be inherited from parent to child

i.e: If parent thread is daemon then automatically child thread is also daemon and if parent thread is Non-daemon then automatically child thread is also Non-daemon

Note: It is impossible to change the main thread as daemon

- because main thread is already started by JVM yet beginning

```
class Test{
  psvm(String[] args){
    Sop(Thread.currentThread().isDaemon()); // false
    Thread.currentThread().setDaemon(true);
```

o/p: RuntimeException, IllegalThreadStateException

```
    MyThread t = new MyThread();
    Sop(t.isDaemon()); // false
    t.setDaemon(true);
    Sop(t.isDaemon()); // true
  }
}
```

o/p: false, true

```
class MyThread extends Thread{
}
```

whenever last Daemon thread terminates automatically all Daemon threads will be terminated irrespective of their position

```
class MyThread extends Thread {
    public void run() {
        for(int i=0; i<10; i++) {
            SOP("child Thread");
            try {
                Thread.sleep(2000);
            }
            catch (IllegalMonitorStateException e) {}
        }
    }
}

class DaemonThread Demo {
    psvm(String[] args) {
        MyThread t = new MyThread();
        t.setDaemon(true);  ——— (1)
        t.start();
        SOP("end of main thread");
    }
}
```

If we are commenting line ①

both main and child threads are non Daemon and hence, both threads will be executed until there completion

If we are not commenting ①

main thread → Non-Daemon
child thread → Daemon

hence whenever main thread terminates automatically child thread will be terminated

in this case O/P :

① End of main thread
child Thread

② End of main thread

③ child Thread
End of main thread

* when ever last Non-Daemon thread terminates automatically all daemon threads will terminate