Deadlock

- Definition
- Conditions
- Analysis
- Avoidance

Background

Mutual exclusion means that a thread can be delayed

- A thread will not be allowed to enter a critical region (using a shared resource) as long as it is occupied by another thread.
- For consistency (concurrency correctness), predictability (real-time correctness), and for efficiency (embedded computing), access of such a locked resource may not be interrupted or subject to a *roll-back*.

Hence, we have no preemption on resources (only on time as in preemptive scheduling; not to be confused).

- If the blocking never ends we have a **Deadlock** (Swe: Dödläge)

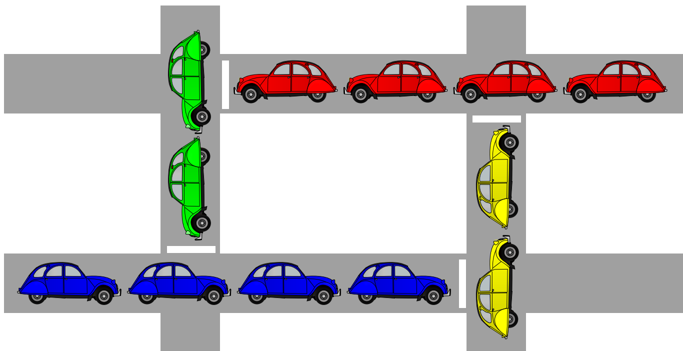
Wikipedia: Deadlock refers to a specific condition when two or more processes are each waiting for the other to release a resource, or more than two processes are waiting for resources in a circular chain.

The problem

If waiting can be or is circular:

- When several threads can be waiting for each other we have a Deadlock risk.
- When several threads are waiting for each other we have a Deadlock.

Thus, circular wait appears to be related too deadlock:



Example: deadlock with semaphores

P1

```
S1.take();
S2.take();
...
S2.give();
S1.give();
```

P2

```
S2.take();
S1.take();
...
S1.give();
S2.give();
```

P1:	P2:	
S1.take();		
	S2.take();	
	S1.take();	*blocked*
S2.take();		*blocked*

- *Deadlock may occur if*: one thread performs a take, followed by a context switch (swe: trådbyte).

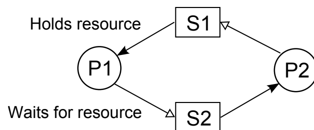
Example cont'd: resource allocation graph

P1

```
S1.take();
S2.take();
...
S2.give();
S1.give();
```

P2

```
S2.take();
S1.take();
...
S1.give();
S2.give();
```



Method: draw resources (boxes) and threads (circles). Draw arrows for **hold** (filled) + **wait** (outlined).

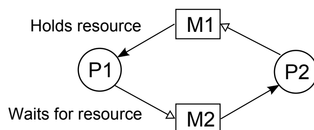
Deadlock with monitors

P1

```
m1.op1();
```

P2

```
m2.op1();
```




```
class M1 {
  synchronized void op1() {
    m2.op2();
  }
  synchronized void op2() {
    wait();
  }
}
```

```
class M2 {
  synchronized void op1() {
    m1.op2();
  }
  synchronized void op2() {
    wait();
  }
}
```

Necessary conditions for deadlock

Necessary conditions for deadlock to occur:

1. Mutual Exclusion - only one thread can access a resource at a time.
2. Hold and Wait - a thread can reserve a resource and wait for another.
3. No *resource* preemption - a thread can not be forced to release held resources.
4. Circular Wait - thread-resources dependencies must be circular.

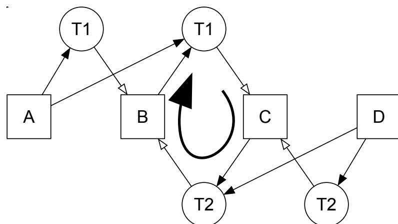
Monitor - satisfied conditions:

1. Monitor - one thread only is allowed to enter at a time.
2. Call of an operation in a monitor from inside a monitor operation in another monitor.
3. A monitor can only be released if a thread voluntarily waits (wait()) or exits the monitor.
4. But 4? Must prevent circular wait that can result in deadlock.

Resource allocation graphs

Tool for detecting circular hold-wait situations and to determine under which conditions deadlock can occur.

1. Draw resources
2. Draw all hold-wait situations (arrows from each held resource to a thread marker, arrows from thread marker to resource waited for)
3. Circular? Then risk for deadlock. The number of 'hold-wait' links in the circular chain shows how many and which threads are required for deadlock.



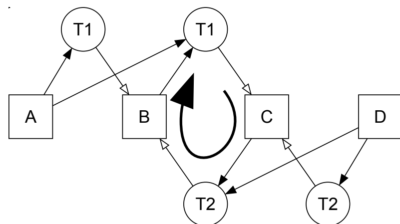
Resource allocation graphs - example

Thread 1

```
A.take();
B.take();
C.take();
C.give();
B.give();
A.give();
```

Thread 2

```
D.take();
C.take();
B.take();
B.give();
C.give();
D.give();
```



Conclusion: Deadlock possible when T1 is waiting for C and T2 is simultaneously waiting for B! Circular wait!

Monitors in Concurrent Pascal

Concurrent Pascal (Per-Brinch Hansen 1979) only has monitors and the rule: *no forward references*, i.e. the program we looked at earlier is illegal:

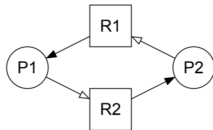
```
class M1 {
  synchronized void op1() {
    // Illegal since it introduces
    // a forward reference
    m2.op2();
  }
  synchronized void op2() { ... }
}
class M2 {
  synchronized void op1() {
    m1.op2();
  }
  synchronized void op2() { ... }
}
```

- Monitors, Semaphores, etc. are often referred to as *resources*.
- Generally: all resources is assigned a (partial) order, only allocate from lower to higher.
- M2 can call M1 but not the other way around.

Limitations in the language - a good idea?

- Deadlock impossible in Concurrent Pascal with Monitors as resources.
- Often inefficient and unpractical - might be necessary to prematurely allocate resources just to satisfy the demands on allocation order.
- Easy to implement ones own resource management using Monitors:

```
/*monitor*/ class R {
  boolean occupied;
  synchronized void request() {
    while (occupied) wait();
      occupied = true;
    }
  synchronized void release() {
    occupied = false;
    notify();
  }
}
```




```
R R1,R2;
class P1 extends Thread {
  ...
  R1.request();
  R2.request();
  ...
}
class P2 extends Thread {
  ...
  R2.request();
  R1.request();
  ...
}
```

Concepts - summary

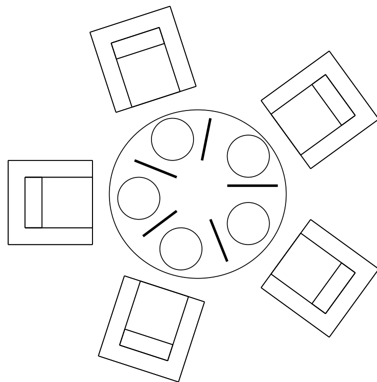
- Deadlock (swe: dödläge)
  - When several resources attempts to allocate the same resource one must be able to get it.
  - Bad enough if there exists an execution order such that Deadlock occurs - even if it happens only seldom. The system locks, hangs, nothing happens. Can apply to subsystems.
- Starvation (swe: svält)
  - If a thread attempts to allocate a resource it must be able to get it eventually.
  - We renounce the 'no starvation' property in favor of priority; less important activities might suffer from starvation.
- Livelock
  - Occurs when several threads attempts to allocate the same resource but none actually gets it due to the execution pattern.
  - Behaves like Deadlock, but if you study the system closely the threads actually run. They do no meaningful work though.

Dining Philosophers problem

- The life of a philosopher is boring:

```
class Philosopher
  while
    think();
    preProto();
    eat();
    postProto();
```

- Logical spaghetti:
  - Two forks are required to eat
- Solution requirements:
  - No deadlock
  - No starvation
  - Efficient



Semaphore solution 1

```
Semaphore[] fork = new MutexSem[5];
for (int i=0; i<5; i++) fork[i] = new MutexSem();

class Philosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}

  public void run() {
    while (true) {
      think();
      fork[i].take();
      fork[(i+1)%5].take();
      eat();
      fork[i].give();
      fork[(I+1)%5].give();
    }
  }
}
```

- Eat() acts as a critical region for a pair of forks, but for different pairs of forks
  - mutual exclusion for both
- Can this solution cause Deadlock? Two resources is required for each activity (hold-wait satisfied) so we have to make a more detailed analysis.

Philosophers 1 - deadlock?

- Draw a complete allocation graph, all Hold-Wait dependencies

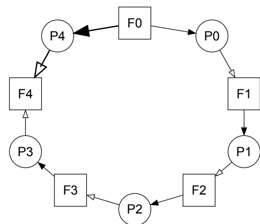
- Circular → unsafe program, Deadlock can occur.
- Step 2: Can we present a scenario where deadlock occurs or prove that the situation can not occur in practice?
- Scenario: Suppose all 5 philosophers starts simultaneously, takes their left forks and then (all of them) waits for their right forks.
- This solution can thus cause deadlock.
- Can we solve the problem in a better way?
    - avoid circularity or
    - make sure that all Hold-Wait can not occur simultaneously?

Philosophers 2 - one left handed philosopher

```
Semaphore[] fork = new MutexSem[5];
for (int i=0; i<5; i++) fork[i] = new MutexSem();
```

```
class Philosopher
  extends Thread {
  int i;
  Philosopher(int i)
    {this.i=i;}
  public void run() {
    while (true) {
      think();
      fork[i].take();
      fork[i+1].take();
      eat();
      fork[i].give();
      fork[i+1].give();
    }
  }
}
```

```
class LeftPhilosopher
  extends Thread {
  int i;
  LeftPhilosopher(int i)
    {this.i=i;}
  public void run() {
    while (true) {
      think();
      fork[0].take();
      fork[4].take();
      eat();
      fork[4].give();
      fork[0].give();
    }
  }
}
```

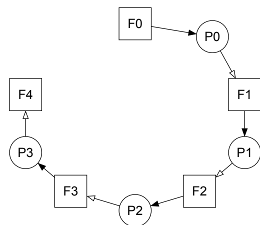


- No circular dependency
- No deadlock

Philosophers 3 - only four chairs

```
Semaphore[] fork = new MutexSem[5];
for (int i=0; i<5; i++) fork[i] = new MutexSem();
Semaphore room = new CountingSem(4);

class Philosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}
  public void run() {
    while (true) {
      think();
      room.take();
      fork[i].take();
      fork[(i+1)%5].take();
      eat();
      fork[i].give();
      fork[(i+1)%5].give();
      room.give();
    }
```



Complete allocation graph cyclic, 'unsafe' as before, but: At most four Hold-Wait can be active simultaneously → at least one philosopher can eat, no deadlock possible.

Philosophers 4 - polite philosophers

A philosopher only picks up the forks and starts to eat if BOTH forks are free.

- Implemented using a monitor or a *MultistepSem*.
  - Trivially deadlock free since no Hold-Wait situations occur.
  - but, starvation possible.
- Suppose two philosophers, e.g. 1 and 3 agrees to eat alternating:
  - I.e. philosopher 1 eats until philosopher 3 has begun to eat, and the other way around
  - Now will philosopher 2 never have two forks free at the same time, i.e. **philosopher 2 will starve!!**