Nebenläufige Programmierung

Prozesse:

Bei vielen Programmierprojekten gibt es Aufgaben, die gleichzeitig erfüllt werden müssen.
Server müssen mehrere Anfragen gleichzeitig bearbeiten können.
Prozesse können quasi parallel ausgeführt werden, d.h. dass das Betriebssystem die parallele Ausführung simuliert, indem die Prozesse gewechselt werden, ohne dass der Benutzer dies bemerkt.
Jeder Prozess hat : - sein eigenes Code- Segment
- einen eigenen Stack

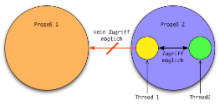
Ein Prozess setzt sich aus dem: Programmcode und vom Betriebssystem geschützten Adressbereich, der ausschließlich dem Prozess zugeordnet ist zusammen.
Somit kann ein Prozess P1 nicht auf den Adressraum eines zweiten Prozesses P2 zugreifen.
Prozesse sind also schwergewichtig: - der Kontextwechsel ist aufwendig
- die Interprozesskommunikation ist schwierig, weil der Speicherbereich zwischen Prozessen auf dem selben Computer strikt voneinander getrennt sind.

Anschauliches Beispiel von Prozessen:

Threads:

- Threads (engl. für Faden, Strang) werden von Prozessen erzeugt und laufen in deren Adressraum
- Könnte man als ‚leichtgewichtig' bezeichnen: - Interthreadkommunikation ist einfacher, weil verschiedene Threads jeweils auf die Daten der anderen Threads im selben Prozeß zugreifen können.
-Threadwechsel also schneller als Prozeßwechsel
- Thread hat seinen eigenen Stack und damit insbesondere seine eigenen lokalen Variablen.
- Es ist so, als ob mehrere Prozesse im gleichen Programm laufen.

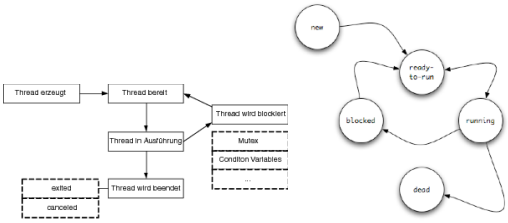
Anschauliches Bespiel von Prozessen & Threads:



Scheduling und Zustände von Threads:

Auch bei der Thread- Programmierung ist ein Scheduler entweder in der Thread- Bibliothek oder in dem Betriebssystem vorhanden, der bestimmt, wann welcher Thread Prozessorzeit erhält.
Hier kann die Zuteilung wie bei den Prozessen prioritäts- und zeitgesteuert erfolgen.
zeitgesteuerte Threads: jedem Thread wird eine bestimmte Zeit zur Verfügung gestellt, ehe dieser automatisch unterbrochen wird und anschließend ein anderer Thread an der Reihe ist.
prioritätsgesteuerte Therads: der Thread erhält mit der höchsten Priorität vom Scheduler den Zuschlag. Außerdem wird ein laufender Thread abgebrochen, wenn ein Thread mit einer höheren Priorität ausgeführt wird.
Also:
Der sogenannte Scheduler vergibt CPU-Rechenzeit an die konkurrierenden Threads.
Ein Thread läuft so lange, bis er
- die CPU freiwillig abgibt oder
- blockiert ist oder
- durch einen anderen Thread verdrängt wird

Anschauliches Beispiel



Ein Thread befindet sich immer in einem der folgenden Zustände:
new/ Thrad erzeugt: Der Thread ist erzeugt, ist aber noch nicht gestartet
ready-to-run/ Thread bereit: Der Thread ist laufbereit und konkurriert mit anderen Threads um die Zuteilung der CPU
blocked/ Thread wird blockiert: Der Thread ist blockiert, weil er entweder eine I/O-Operation angestoßen hat, sich freiwillig blockiert oder erfolglos synchronisierten Code ausführen wollte
running/ Thread in Ausführung: Der Thread hat die CPU vom Scheduler zugewiesen bekommen und wird ausgeführt
dead/ Thread wird beendet: Der Thread hat entweder seinen Programmcode komplett abgearbeitet oder wurde anderweitig beendet. Der Thread kann nicht wieder gestartet werden. Auf die Methoden und Attribute kann jedoch weiterhin zugegriffen werden

Threads in Java:

1. Threads müssen mit der Methode run() implementiert werden
2. Threads werden durch Aufruf der Methode start() nebenläufig gestartet
3. Mit start() überführen wir den Thread in eine Konkurrenz mit anderen Therads

Es gibt in Java zwei Möglichkeiten, Threads zu erstellen:
1. Ableitung der nebenläufigen Klasse von der Klasse Thread

```
public class MyThread extends Thread
```

Vorteil:
- Alle Thread-Methoden können direkt genutzt werden
- Erzeugung einfacher:

```
MyThread t1 = new MyThread(); t1.start();
```

Nachteil:
- In Java ist keine Mehrfachvererbung möglich. Eine von Thread abgeleitete Klasse kann nicht zusätzlich von einer anderen Klasse erben

2. Implementierung der Schnittstelle Runnable

```
public class MyThread implements Runnable
```

Threads werden gestartet, indem ein neues Thread-Objekt erzeugt wird, dem im Konstruktor als Parameter eine Referenz auf das Runnable-Objekt übergeben wird:

```
Thread t1 = new Thread( new MyThread() );
```

Vorteil:
• Ableiten von anderen Basisklassen möglich

Nachteil:
• Thread-Methoden sind aufwendiger zu verwenden. Um eine Thread-Methode aufrufen zu können, muss zunächst eine Referenz auf den aktuell ausführenden Thread durch Aufruf der statischen Methode Thread.currentThread() erlangt werden.

Beispiel zur Erzeugung von Threads durch Erweiterung der Klasse Thread:

```
public class MyThread extends Thread {

	private String text;
	public MyThread( String text ){
		this.text = text;
	}
	public void run(){
		for( int i = 0; i < 10; i++ ){
			System.out.println(text);    //Rechenzeit verbrauchen
			for( int j = 0; j < 100000; j++ ){
				double f = Math.sqrt(Math.exp(4545534.4 / 898997));
			}
		}
	}

}
public class MyMain {
	public static void main(String[] args){
		MyThread t1 = new MyThread("Thread1");
		MyThread t2 = new MyThread(" Thread2");
		t1.start(); t2.start();
	}
}
```

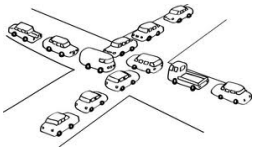
Deadlocks:

Problemstellung:

Eine Verklemmung (Deadlock) kann wie folgt definiert werden:
Eine Menge von Prozessen sperren sich gegenseitig, wenn jeder Prozess der Menge auf ein Ereignis wartet, das nur durch einen anderen Prozess der Menge ausgelöst werden kann.

Da alle am Deadlock beteiligten Prozesse warten, kann keiner ein Ereignis auslösen, so dass ein anderer geweckt wird. Also warten alle beteiligten Prozesse ewig.

Anschauliches Bespiel aus der realen Welt:



Bedingungen für Deadlocks

Damit ein Deadlock entsteht, müssen alle vier folgenden Bedingungen erfüllt sein:

1. Wechselseitiger Ausschluss:
   Jedes Betriebsmittel wird entweder von genau einem Prozess belegt oder es ist verfügbar.
2. Anforderung weiterer Betriebsmittel:
   Ein Prozess, der bereits Betriebsmittel belegt hat, kann weitere Betriebsmittel anfordern.
3. Ununterbrechbarkeit:
   Die von einem Prozess belegten Betriebsmittel können nicht von außen entzogen werden; der Prozess selbst muss sie explizit freigeben.
4. Zyklische Wartebedingung:
   Es muss eine zyklische Kette von Prozessen geben, so dass jeder Prozess ein Betriebsmittel anfordert, dass vom nächsten Prozess der Kette belegt ist.

Strategien der Deadlock-Behandlung:

Durch geschickte Auswahl der Prozesse können also Deadlock-Situationen behandelt werden.
Man kann grundsätzlich vier Strategien unterscheiden:

1. Ignorieren des Problems (Strategie von Unix Systemen)
2. Erkennen und Beheben von Deadlocks
3. dynamische Verhinderung durch vorsichtige Betriebsmittelzuteilung
4. Vermeidung durch Verbieten einer der 4 notwendigen Bedingungen