Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Basiert auf ALP IV, SS 2003
Klaus-Peter Löhr
Freie Universität Berlin

Überblick

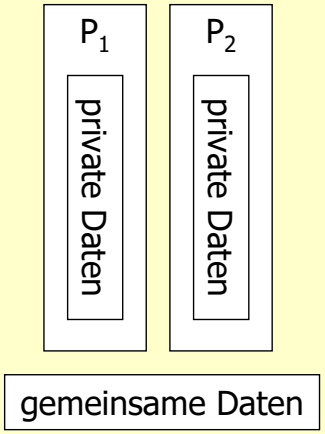
Thema

- Programmstruktur und Prozesse
- Prozesse in Java
- Implementierungsmöglichkeiten für Prozesse

2 Nebenläufige Prozesse

2.1 Programmstruktur und Prozesse

- Prozess = Anweisungen + Daten
  - private
  - gemeinsame



2.1 Programmstruktur und Prozesse

- Nichtsequentielle Programmierung nutzt Konstrukt „Prozess" zusätzlich zu den bekannten Programmiermitteln
- Aber:
  Wie verhält sich das Konstrukt „Prozess" zu den üblichen Strukturierungseinheiten wie
  - Prozedur,
  - Modul,
  - Klasse,
  - Objekt, ... ?

2.1.1 Prozesse ohne Bezug zur Programmstruktur

- Nebenläufigkeitsanweisung:
- co …||… oc
- par(…, …) Algol 68 (1968)
- cobegin 1 do … Edison (1982)
  also 2 do …
  also … end
- PAR Occam (1987)

```
INT x:
ch1 ? x -- receive from channel ch1
INT y:
ch2 ? y -- receive from channel ch2
```

- und ähnliche

2.1.1 Prozesse ohne Bezug zur Programmstruktur

- Gabelungsanweisung:

```
p = fork r=sin(x)+cos(y);
b = b+1;
a = a+b;
join p
```

Sind keine Blöck
oder sonstwie erkennbare
Programmstrukturen

2.1.2 Prozeduren als Prozesse

- Variante 1 – prozedurbezogene Gabelungsanweisung:
- FORK ProcedureInvocation
  FORK p(x)
  - Aufrufer veranlaßt asynchrone Ausführung
  - nach vollzogener Parameterübergabe!
- process ProcedureInvocation
  - Burroughs Extended Algol (1971)
- Jede Prozedur kann asynchron ausgeführt werden: *Asynchronie an Aufruf gebunden und vom Aufrufer bestimmt*

2.1.2 Prozeduren als Prozesse

- Variante 2 – asynchrone Prozedur:
- process ProcedureDeclaration (1.1.3.1 ←)
  - d.h. *Asynchronie ist an die Prozedur gebunden*
  - ebenfalls nach vollzogener Parameterübergabe
- Jac - Java with Annotated Concurrency:
  (http://page.mi.fu-berlin.de/~haustein/jac/doku.html)
  - Annotation asynchroner Methoden
  - async MethodDeclaration oder auto MethodDeclaration
  - public class Storage {
    /**
    * save data to disk
    * @jac.async
    * @jac.require d != null */
    public void writeToDisk(Data d) {...}
  - Aufruf der Operation writeToDisk kehrt unmittelbar nach der Überprüfung der Voraussetzung zurück, während die Abarbeitung in einem separaten Kontrollfluss stattfindet

2.1.2 Prozeduren als Prozesse

- Variante 3 – In Skriptsprachen …
- …programmbezogene Gabelungsanweisung:
  - SimpleCommand & Unix Shell
- …programmbezogene Nebenläufigkeitsanweisung:
  - cmd1 & cmd2 & ... & cmdn
  - cmd1 | cmd2 | ... | cmdn „pipeline"

2.1.2 Prozeduren als Prozesse

- Variante 3a – Programmbezogene Gabelungsanweisung an der Systemschnittstelle von Unix – d.h. fork ist Systemaufruf, nicht Schlüsselwort
- pid = fork();
  - Klont den laufenden Prozeß
  - mit Ausnahme von pid:
  - Kindprozeß erhält 0,
  - Elternprozeß erhält Kind-Nummer
- if (pid==0) {... }
  - Kindaktivität
  - endet mit exit()
- else {... x = wait(&status); ...}
  - Elternaktivität, kann mit wait
  - auf exit eines Kinds warten

2.1.3 Module als Prozesse

- Zur Erinnerung: ein Modul enthält
  - Vereinbarungen von Exporten
  - Vereinbarungen von Importen aus anderen Modulen
  - Vereinbarungen von Typen, Konstanten, Variablen
  - Vereinbarungen von Prozeduren
  - Block von Anweisungen zur Initialisierung
- „Prozessmodul“:
  - Block beschreibt die Aktivität eines eigenständigen Prozesses, der bei der Initialisierung gestartet wird, ...

2.1.3 Module als Prozesse

- z.B. mit Java-ähnlichem Modul wie folgt:

```
class Myprocess {
   static ...
   static ...
   .....

   process { Statements }

}
```

statt statischer Initialisierung mit static

2.1.3 Module als Prozesse

- Ähnlich – wenngleich nicht identisch – in Ada (1979)

```
task Identifier is
... Declarations ...
begin
   Statements
end Identifier;
```

- Dies ist lokale Vereinbarung in Prozedur oder Modul.

2.1.4 Objekte als Prozesse

- (genauer: als Prozeß-Inkarnationen)
- ... wie Module – aber in beliebig vielen Exemplaren:
- PROCESS als Variante von class, z.B. so:

```
PROCESS ProcessIdentifier {
   Declarations
   public  ProcessIdentifier (Parameters) {
      Statements
   }
      ...
}
```

2.1.4 Objekte als Prozesse

- Ada (1979):

```
task type MyTask is
... Declarations ...
begin
   Statements
end MyTask;
```

- Vereinbarung von Prozessobjekten: t1: MyTask; t2: MyTask;
- SR (1980):

```
resource Res ...
body Res(Parameters)
initial ...
process P ... end
end Res
```

- Erzeugung von Prozessobjekten:

```
r = create Res()
```


2.2 Prozesse in Java

- Java sieht kein Schlüsselwort für Prozesse vor, sondern bestimmte Klassen/Schnittstellen.
- Mit anderen Worten: der Prozessbegriff wird mit Mitteln der Objektorientierung eingeführt.
- Bewertung:
- hübsche Übung in Objektorientierung, aber nicht angenehm für den Benutzer, weil eher implementierungsorientiert als problemorientiert

2.2.1 Thread und Runnable

- Im Paket java.lang befinden sich

```
interface Runnable {
  public void run();
}

public class Thread implements Runnable {...
public void run()   {}    // thread activity
public void start() {...} // start thread
public void join()  {...} // wait for thread
}
```

- Ein Thread-Objekt ist ein Prozess, der durch die Operation start gestartet wird und dann selbsttätig run ausführt ...

Beispiel

```
public class SayA extends Thread {
  public void run() {
    for(;true;System.out.print("A"));
  }
}
```

```
public class SayB extends Thread {
  public void run() {
    for(;true;System.out.print("B"));
  }
}
```

```
public class Talkshow {
  public static void main(String[] argv)
    SayA a=new SayA();
    SayB b=new SayB();
    a.start();
    b.start();
  }
}
```

```
BBBBBBBBBBBBBAAAAABBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
BBBBBBBBBBBBBBBBBBBBAAAAABBBBB
AAABBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBAAAABAAABBBBBBBBB
```

2.2.1 Thread und Runnable

- Nutzung:
  - Spezialisierung von Thread
  - Überschreiben von run()

```
class MyProcess extends Thread {
      .....
public run() { gewünschte Aktivität }
}
```

- und erzeugt und startet einen Prozeß wie folgt:

```
Thread t = new MyProcess();
t.start();
```

- oder einfach

```
new MyProcess().start();
```


2.2.1 Thread und Runnable

- Alternative Technik über alternativen Thread-Konstruktor
  public Thread(Runnable r) {...}

```
class  MyActivity implements Runnable {
   public MyActivity(...) {...}
   public void run() { gewünschte Aktivität }
}
```

- Start mit

```
new Thread(new MyActivity(...)).start();
```

- Vorteil:
  bessere Entkoppelung zwischen Thread-Klasse und anwendungsspezifischem Code

2.2.1 Thread und Runnable

- Kleine Übung: Komfortable Nachbildung von

```
p = fork P(par);
...
wait p;
```

- In Java:

```
Thread t = new C(par);
...
t.join();
```

- mit Klasse C statt Prozedur P (bzw. Exemplare davon)

2.2.1 Thread und Runnable

- Parameterübergabe über Konstruktur

```
class C extends Thread {
   private Par par;
   public C(Par par) {
      this.par = par;
      start(); }
   public run() { Prozessaktivität mit Bezug auf par }
}
```

- Achtung:
  Dieser Stil ist riskant, weil bei weiterer Unterklassenbildung start auf einem nicht vollständig initialisierten Objekt ausgeführt werden könnte!

2.2.1 Thread und Runnable

- Programmstart und -ende:
- Nicht explizit erzeugt wird der Urprozess (root process, main thread)
- Er wird beim Programmstart automatisch erzeugt und führt die Prozedur main aus.
- Er ist ein regulärer Thread wie die anderen auch.
- Achtung: Durch Bibliotheks-Import kann es auch unsichtbare Hintergrund-Threads geben!
- Ein Programm wird beendet, wenn
  - System.exit() aufgerufen wird
  - oder alle Threads beendet sind
    - regulär
    - durch nicht abgefangene Ausnahmen.

2.2.2 Wichtige Operationen der Klasse Thread

- Konstruktoren:
- public Thread()
  - erzeugt Thread , der run ausführen wird, mit dem Namen "Thread–n" im *Zustand neu* (created)
- public Thread(Runnable r)
  - erzeugt Thread, der r.run() ausführen wird, mit dem Namen "Thread–n" im Zustand neu
- public Thread(String name)
  - erzeugt Thread, der run ausführen wird, mit dem angegebenen Namen im Zustand neu

2.2.2 Wichtige Operationen der Klasse Thread

- Operationen:
- public void run()
  - vom Benutzer vorzugebende Aktivität; nach deren Ablauf befindet sich der Thread im *Zustand tot / beendet* (dead, terminated)
- public void start()
  - versetzt den Thread vom Zustand neu in den *Zustand lebendig* (alive), startet Operation run ;
  - Leeroperation, wenn der Thread bereits tot war.
  - wirft IllegalThreadStateException wenn der Thread bereits lebendig war
- public boolean isAlive()
  - beantwortet die Frage, ob der Thread lebendig ist
  - Beachte: die Antwort ist i.a. nichtdeterministisch!

2.2.2 Wichtige Operationen der Klasse Thread

- public String getName()
  - liefert den Namen des Threads
- public void interrupt()
  - setzt ein verborgenes Boolesches Attribut interrupt des Threads (das nach der Erzeugung des Threads zunächst false ist) auf true ;
  - der Zustand dieses Attributs kann abgefragt werden: public bolean isInterrupted()
- Die einzigen Methoden, mit denen Prozesse direkt aufeinander einwirken können, sind start, join, interrupt

2.2.2 Wichtige Operationen der Klasse Thread

- public final void join()
  - wartet, bis der Thread *nicht lebendig* (!) ist
  - Zustand *wartend*
  - wirft InterruptedException wenn das interrupt-Attribut des *ausführenden* (!) Threads gesetzt ist bzw. während des Wartens gesetzt wird; es wird gelöscht.
- public final void join(long milliseconds)
  - wie joint()
  - aber mit Abbruch bei Überschreitung der angegebenen Wartezeit (timeout)
  - Wirft InterruptedException unter gleichen Bedingungen wie join()

2.2.2 Wichtige Operationen der Klasse Thread

- Achtung:
  Es macht keinen Sinn, mit join auf die Beendigung eines Threads zu warten, der noch gar nicht gestartet wurde
  - Thread ist nicht lebendig nach seiner Beendigung
  - aber auch vor seinem Start
- Aus diesem Grund ist es auch riskant, join für einen Thread zu machen, der an anderer Stelle im Programm gestartet wurde – oder eben auch nicht gestartet wurde!

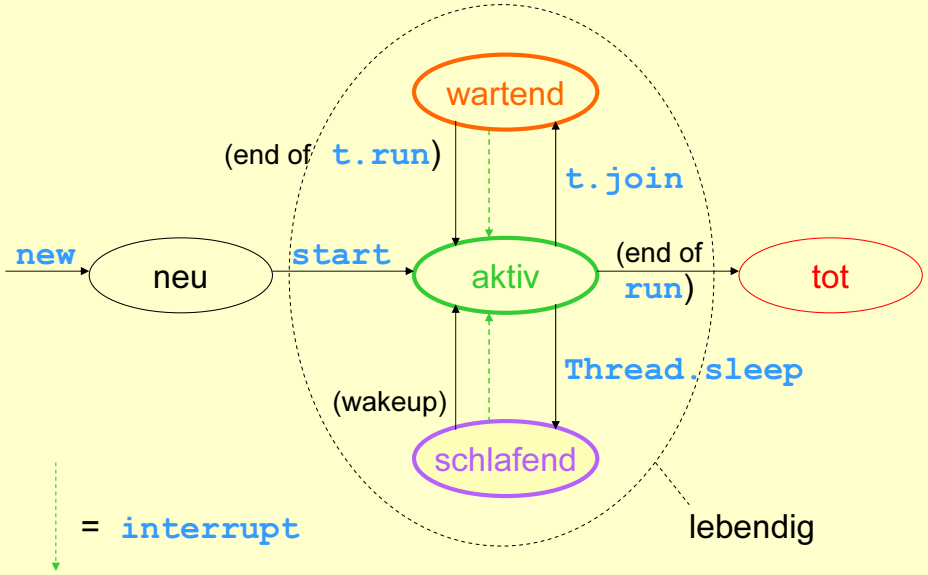
2.2.2 Wichtige Operationen der Klasse Thread

- Statische Prozeduren:
- public static Thread currentThread()
  - liefert einen Verweis auf den ausführenden Thread
  - nicht verwechseln mit this !
- public static boolean interrupted()
  - beantwortet die Frage, ob das interrupted-Attribut des ausführenden (!) Threads gesetzt ist, und löscht es (!)
  - Beachte: die Antwort ist i.a. nichtdeterministisch!

2.2.2 Wichtige Operationen der Klasse Thread

- public static void sleep(long milliseconds)
  - versetzt den ausführenden Thread in den Zustand *schlafend* (sleeping), aus dem er frühestens nach Ablauf der angegebenen Zeit erlöst wird
  - wirft IllegalArgumentException bei negativem Argument
  - wirft InterruptedException wie bei join

Zustandsübergänge im Bild



2.2.3 Speichermodell

- *gibt weniger Garantien* als es der naiven Vorstellung von der Ausführung eines Programms entspricht,
- erlaubt damit dem Übersetzer *beträchtliche Freiheiten* auf raffinierten Rechnerarchitekturen
- *gibt immerhin gewisse Garantien*, an die man sich beim Programmieren halten kann/muss

2.2.3 Speichermodell

- Einfache Variable (1.2.2←)
  - sind alle mit Ausnahme von long und double – und diese auch, sofern als volatile („flüchtig") vereinbart.
- Sichtbarkeit:
  - Der von einem Thread geänderte Wert einer gemeinsamen Variablen ist für andere Threads sofort sichtbar, wenn die Variable als volatile vereinbart ist.

2.2.3 Speichermodell

- Beachte:
  Wenn weder volatile noch Synchronisation (➔3) zum Einsatz kommen, ist nicht gesichert, dass
  - Lese- oder Schreibzugriffe auf long- oder double-Variable unteilbar sind,
  - die Änderung an einer Variablen von einem anderen Thread aus sofort sichtbar ist,
  - die Änderungen an mehreren Variablen von einem anderen Thread aus in der gleichen Reihenfolge beobachtet werden.

2.2.3 Speichermodell

- Beispiele:

```
char tag = '*';
double volume = 0.0;
volatile long count = 0;
volatile String[] s = {...};
```

- tag = ' '; unteilbar, aber u.U. nicht sofort sichtbar
- volume = 3.14; nicht unteilbar
- count = 4711; unteilbar und sofort sichtbar
- count++; nicht unteilbar (!), aber sofort sichtbar
- s[0] = "hallo" unteilbar, aber u.U. nicht sofort sichtbar (!)

2.3 Implementierung von Prozessen

- durch Prozessoren
- durch Prozesse des Betriebssystems
- durch Kernel-Level Threads des Betriebssystems
- durch User-Level Threads einer Threading-Bibliothek
- durch das Laufzeitsystem der Sprache
- durch den Interpretierer der Sprache (falls interpretiert)
- + Mischformen

2.3.1 Mehrprozessorsystem (multiprocessor system)

- erlaubt echte Parallelausführung, weil mehrere Prozessoren
- macht nur Sinn bei grobkörniger Prozessstruktur d.h. wenige, langlaufende Prozesse
- Code + Daten
  - in gemeinsamem Speicher,
  - teils in privaten,
  - teils in gemeinsamen Segmenten des VS
- Im Idealfall wird jeder Prozess von eigenem Prozessor ausgeführt (unrealistisch)

2.3.1 Mehrprozessorsystem (multiprocessor system)

- Prozesserzeugung:
  fork-Systemaufruf veranlasst das Betriebssystem,
  - in seiner Buchführung über die Prozessoren nach einem untätigen Prozessor zu suchen,
  - eine Prozessbeschreibung (Adressraum, Startadresse, ...) in einem diesem Prozessor zugeordneten Speicherbereich unterzubringen,
  - den Prozessor durch eine externe Unterbrechung zu veranlassen, den neuen Prozess auszuführen.

2.3.2 Mehrprozeßbetrieb

- (multiprogramming, multitasking)
- wenn es mehr Prozesse als Prozessoren gibt
- Kann bei Einprozessor- und bei Mehrprozessor-Systemen auftreten

- Code + Daten
  - in gemeinsamem Speicher,
  - teils in privaten,
  - teils in gemeinsamen Segmenten des VS
- Prozesse werden quasi-parallel ausgeführt, d.h. wechseln sich in der Benutzung der Prozessoren ab (processor multiplexing).
- „Prozess = virtueller Rechner"

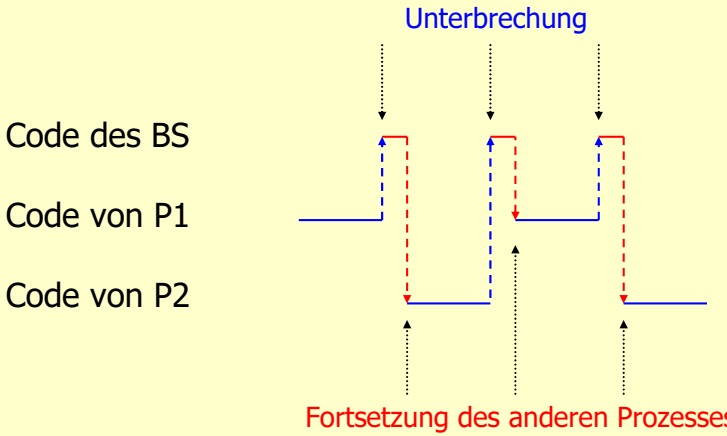
2.3.2 Mehrprozeßbetrieb

- **Beispiel** Einprozessorsystem:

- Gleichmäßige *Reihum-Vergabe* des Prozessors *(round-robin scheduling)*
  - genauer:
    Regelmäßige Zeitgeber-Unterbrechungen *(time slicing)* veranlassen das Betriebssystem zur *Prozeßumschaltung (process switching)*
    - genauer:
      Instruktionsausführung wird zunächst im Betriebssystem fortgesetzt und letztendlich mit den Instruktionen eines anderen Prozesses fortgesetzt (verbunden mit Umschaltung der Adressräume)

2.3.2 Mehrprozeßbetrieb

- Z.B. mit 2 Prozessen:



2.3.2 Mehrprozeßbetrieb

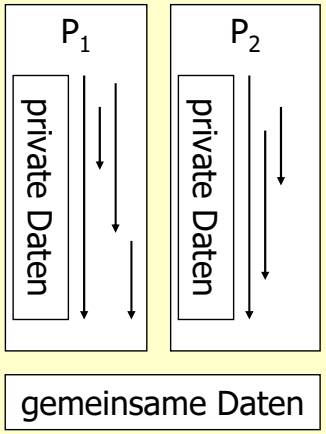
- F: „... aber wie kann es beim Fehlen echter Parallelität Probleme durch nichtatomare Ausdrücke, Anweisungen etc. geben ?"

- Unterbrechungen treten zu nicht verhersagbaren (und nicht reproduzierbaren) Zeitpunkten ein – z.B. mitten in der Auswertung von exact = 3.14D

- *Effekt*:
  Die verzahnte Ausführung ist zwar sehr „grobzähnig" – aber trotzdem unvorhersagbar verzahnt!

2.3.3 Kernel-Level Threads

- Thread (= „Faden"),
  - auch leichtgewichtiger Prozess (lightweight process),
  - im Gegensatz zu schwergewichtiger Prozess (heavyweight process) (= „Prozess" aus 2.3.2)
- entstammt der Betriebssystem-Terminologie,
- bedeutet „Prozess im Prozess", genauer virtueller Prozessor eines virtuellen Rechners



2.3.3 Kernel-Level Threads

- Kernel-Level (im Gegensatz zu „User-Level" ➔2.3.4) bedeutet, daß das Betriebssystem („der BS-Kern") das Thread-Konzept unterstützt, d.h. für alle Prozesse auch deren Threads verwaltet.
- Beachte 1:
  Bei Prozeßwechsel wird der Adreßraum gewechselt, bei Thread-Wechsel innerhalb eines Prozesses nicht!
- Beachte 2:
  Bei einem Mehrprozessorsystem kann das Betriebssystem die Threads eines Prozesses von verschiedenen Prozessoren ausführen lassen (➔ Parallelarbeit!)

2.3.4 User-Level Threads

- Das Betriebssystem kennt kein Thread-Konzept.
- Threading wird durch Code innerhalb des Benutzer-Adressraums realisiert.
- Varianten:
  - Threading-Bibliothek, von verschiedenen Sprachen aus benutzbar
  - Laufzeitsystem der Sprache realisiert Threading
  - Interpretierer (falls interpretierte Sprache) tut dies

Java ...

- Laufzeitsystem bzw. Interpretierer (JVM)
  1. praktiziert entweder eigenes Threading (user-level)
  2. oder nutzt Kernel-Level Threading (falls vorhanden) (Beispiele: Windows, Solaris)
  3. oder nutzt schwergewichtige Prozesse mit gemeinsamen Segmenten (Beispiel: Linux)
- Auf Mehrprozessorsystemen erlauben 2 und 3 parallele Java-Programme!

Zusammenfassung

Thema

- Programmstruktur und Prozesse
  - Prozeduren
  - Module
  - Objekte
- Prozesse in Java
  - Runnable, Thread
  - Operationen von Thread
  - Zustände
- Implementierungsmöglichkeiten für Prozesse
  - Mehrprozessorsystem
  - Mehrprozessbetrieb
  - Kernel-Level Threads
  - User-Level Threads