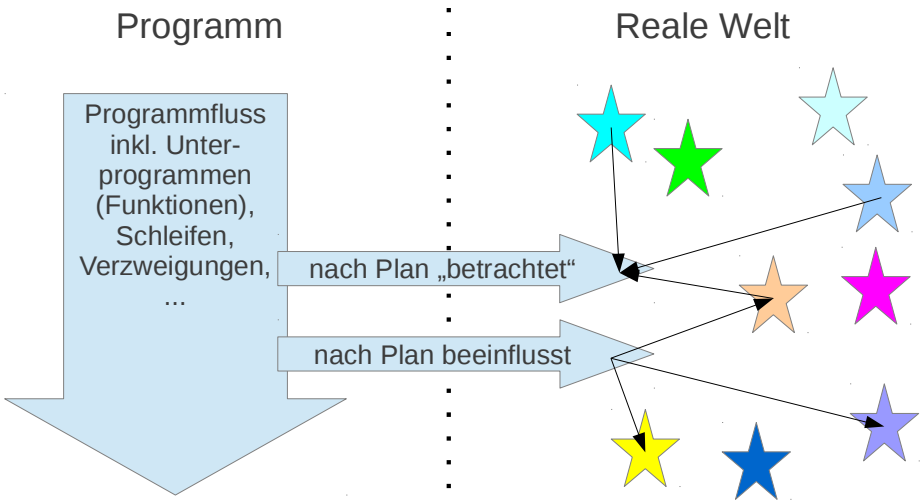
Designing Event-Driven Tcl-Applications

Dipl.-Ing. Martin Weitzel
Technische Beratung für EDV
64380 Roßdorf, Germany
`www.tbfe.de`

Prozedurales Planen und Denken

- Für hinreichend komplexe Vorhaben machen sich Menschen einen „Plan“
  - Ein solcher enthält typischerweise Schritte:
    - diese sind nacheinander,
    - ggf. wiederholt oder auch
    - alternativ auszuführen.
  - Dem Plan folgen sie dann:
    - Ist er gut, wird auch das Ergebnis gut
    - Aber es darf nichts Unerwartetes passieren

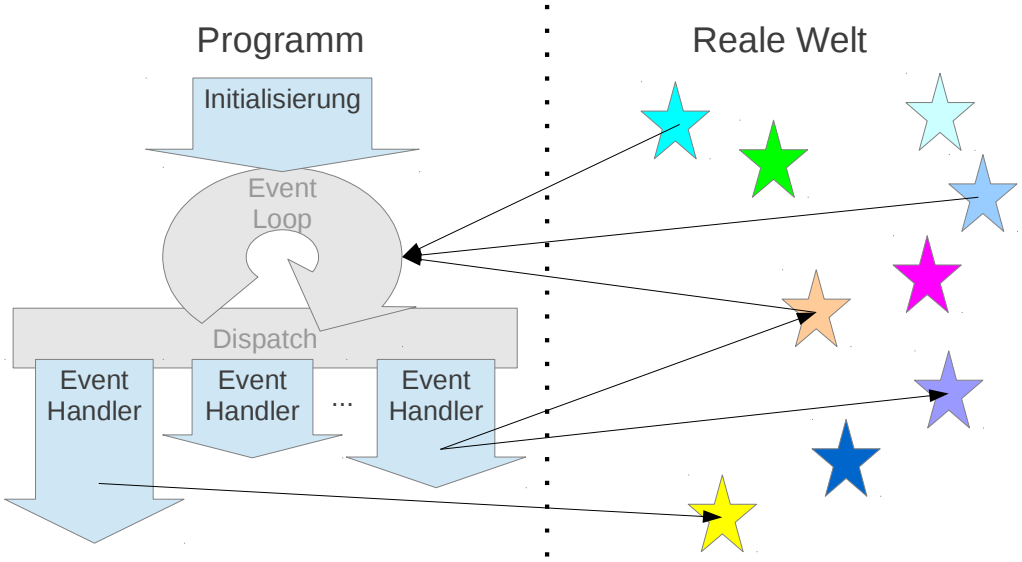
Programmfluss „betrachtet“ reale Welt



Ereignisgesteuertes Handeln und Denken

- Hineingeworfen ins „echte Leben“ ist oft angemessenes Handeln entscheidend
  - Es gibt nicht den großen Plan sondern
  - viele kleine Pläne, einen für jedes Ereignis
- Ereignisgesteuert heißt also:
  - Möglichst alle Eventualitäten voraussehen,
  - ohen dabei (zu viel) Reihenfolge vorzugeben
- Ereignisgesteuert heißt nicht: „Planlos handeln“

Programmfluss durch „reale Welt“ gesteuert



Lesen aus Dateien (und von seriellen Schnittstellen)

- Viele Applikationen „verarbeiten Daten“ (EDV !), diese
  - stammen oft aus Dateien, oder
  - kommen über Hardware-Schnittstellen (USB, RS232, …)
- Unix/Linux macht hier keinen einen Unterschied
  - Serielle Schnittstellen haben einen Namen im Dateisystem (typisch: `/dev/...`), damit natürlich auch Zugriffsrechte
  - Öffnen, Lesen, Schreiben (und Schließen) funktioniert darüber geräteunabhängig
  - Schnittstellenparameter werden mittels eines speziellen Systemaufrufs (System-Call) gesetzt

Schnittstellenparameter einstellen

- Serielle Schnittstellen werden konfiguriert mit fconfigure
  - Alles Übliche wird unterstützt:
    - Baudrate, Parität, Start/Stop-Bits …
    - Hard-/Software Flow-Control (treiberabhängig)
  - Translation Modes und Buffering:
    - Z.B. Konvertierung CR/NL <=> NL oder Binary Mode
    - Z.B. Daten nicht zeilenweise sondern zeichenweise absenden
  - Auch synchrones vs. asynchrones Lesen

```
fconfigure stdin -blocking 0 ;# boese, boese :-)
```


Lesen nach Plan: Polling

- Eine Datei / serielle Schnittstelle „nach Plan“ lesen heißt:
  - nachschauen, ob Daten da sind,
  - wenn nicht …
  - … vielleicht etwas Nützliches tun …
  - … und dann gleich wieder nachschauen
- Was aber wenn ...
  - … es nichts (wirklich) Nützliches zu tun gibt?
  - … das Betriebssystem den Prozess beim Nachsehen suspendiert bis Daten anstehen?

Nachteile von Polling

- Typisch notwendig sind Möglichkeiten um
  - vor dem synchronen Lesen in Erfahrung zu bringen, ob der Proezss suspendiert würde
  - ohne CPU-Zeitverbrauch zu schlafen, wenn gerade keine Daten anstehen
- Das Ergebnis:
  - Latenzzeit ist im Schnitt “halbe Schlafenszeit”
  - kurze Latenzzeiten => viel CPU-Last

Tcl-Befehl `chan event` (ehemals `fileevent`)

- Mit Tcl kann das Problem vermieden werden:
  - Für zu überwachende File-Descriptoren werden Event-Handler registriert
    - Der betreffende Befehl ist `chan event`
    - In älteren Tcl-Versionen heißt er `fileevent`
  - Dann geht die Applikation in die *Event-Loop*
    - Das betreffende Befehl ist `vwait`
    - Der zuständige Handler wird dadurch aufgerufen, immer wenn Daten zur Verarbeitung anstehen

Ein oder mehrere Handler?

- Registrierte Handler
  - werden ggf. mit dem File-Deskriptor als Argument aufgerufen, der lese- oder schreibbereit ist
  - müssen bei ähnlich gelagerten Aufgaben somit nur einmal implementiert werden
- Üblicherweise werden Handler für Lesen und Schreiben getrennt implementiert und registriert
  - da es so gut wie nie wesentliche Gemeinsamkeiten gibt,
  - auch wenn es sich um ein und denselben File-Deskriptor handelt, etwa bei einer bidirektionalen TCP-Verbindung

Fallgrube: EOF- und Fehler-Zustand

- Zum Lesen registrierte Handler werden aufgerufen
  - sobald am wenigstens 1 Byte zum Lesen ansteht
  - die betreffenden Datei (bzw. serielle Schnittstelle oder Tcp-Socket)
    - bis *End of File* gelesen wurde
    - oder in den Fehler-Zustand geht
- Der Fehler- bzw. EOF-Zustand ist angemessen zu berücksichtigen:
  - Oft ist es dann sinnvoll, den File-Deskriptor zu schließen
  - Die Registrierung des Handlers wird dadurch aufgehoben
  - **Andernfalls wird der Handler im Fehler- oder EOF-Fall in einer (Endlos-) Schleife immer wieder aufgerufen!**

Fallgrube: Voller Schreibpuffer

- Zum Schreiben registrierte Handler werden aufgerufen
  - sobald wenigstens 1 Byte geschrieben werden kann, ohne dass der Prozess suspendiert wird
  - wieviel genau geschrieben werden kann, hängt von Pufferung und Treiber ab
- Sind kurz-laufende Event-Handler auch in seltenen Ausnahmesituationen unverzichtbar,
  - müssen robuste Applikationen sehr umsichtig vorgehen ...
  - … z.B. ein Protokoll implementieren, welches ggf. die „Aufnahmefähigkeit“ der Datensenke garantiert ...
  - … was aus Bequemlichkeit allerdings oft unterbleibt

TCP/IP Kommunikation (Grundlagen)

- In einem TCP-Netzwerk werden die
  - die Schnittstellen der beteiligten Rechner durch eindeutige IP-Nummern identifiziert:
    - 32 Bit bei klassischem IP (= max. ~4 Milliarden Nodes)
    - 128 Bit bei IPv6 (= reicht für … na egal, wir ham's ja!)
  - die (Server-) Applikationen auf jedem beteiligen Rechner durch eindeutige Port-Nummer identifiziert:
    - 16 Bit = gut 65.000 Werte (pro IP-Adresse!) und ...
    - … mit Aufbau der Client-Server-Verbindung sofort wiederverwendbar – nicht erst nach deren Abbau!

TCP/IP Kommunikation („Halb-offene“ Sockets)

- Das Verstehen der Socket-Abstraktion ist wichtig:
  - Halb-offener Zustand:
    - ein *Server-Socket* wartet auf den Verbindungswunsch eines *Client-Sockets* …
    - … der I.d.R. zu einem nicht vorhersehbaren Zeitpunkt eingeht

Fortsetzung

TCP/IP Kommunikation (verbundene Sockets)

Fortsetzung

  - Verbundener Zustand:
    - Bidirektionaler serieller Datenstrom
    - Programmtechnsch wie Lesen/Schreiben von Dateien
    - Low-level Details für Client- und Server-Applikation transparent auf der Treiberebene abgehandelt
- Die kommunizierenden Partner müssen
  - zwar ein gemeinsames Protokoll miteinander „sprechen“
  - **aber keineswegs in ein und derselben Programmiersprache geschrieben sein!**

Server-Sockets

- Tcl-Befehl `socket -server` *`connectionhandler port`*
  - registriert *`connectionhandler`* für *`port`*
  - kehrt sofort zurück
- Callbacks an *`connectionhandler`* erfolgen
  - sobald ein Client-Verbindungswunsch ansteht,
  - unter Übergabe des File-Deskriptors, der die (nun bestehende) bidirektionale Datenverbindung zum Client repräsentiert
  - aber erst, wenn die Event-Loop aktiv ist:
    - `vwait` in Tcl-Shell (typisch ganz zum Schluss)
    - grundsätzlich am Ende des Scripts in Tk

Client-Sockets

- Tcl-Befehl `socket` *`host port`*
  - versucht eine TCP-Verbindung aufzubauen zu
    - *`port`* (= Applikation) auf
    - *`host`* (= Rechner bzw. Schnittstelle)
  - für *`host`* kann eine IP-Nummer oder ein symbolischer Name stehen (letzteres natürlich nur bei funktionierendem DNS!)
  - der Client-Rechner selbst kann als `localhost` oder `127.0.0.1` angesprochen werden
- liefert im Erfolgsfall den File-Deskriptor, der die (nun bestehende) bidirektionale Datenverbindung zum Server repräsentiert
- zeigt Fehler auf Tcl-übliche Weise an (= ggf. mit *`catch`* abzufangen)

Datenaustausch über Socket

- Programmtechnisch
  - kann die bidirektionale Datenverbindung wie eine Datei behandelt werden
  - erfolgt Lesen und Schreiben per Default synchron
- Die asynchrone Verwendung ist
  - mittels `chan event` registrierter Handler jederzeit möglich
  - im *Server* optional – aber sinnvoll um parallele Client-Verbindungen zu unterstützen
  - im *Client* optional – aber sinnvoll, damit andere Event-Handler nicht blockiert werden (z.B. ein Tk GUI)

Tcl Event-Queuing

- Es ist leicht vorstellbar,
  - dass während der Reaktion auf ein Ereignis weitere Ereignisse eintreffen …
  - … insbesondere bei langer Laufzeit eines zuvor aktivierten Event-Handlers
- Solche Ereignisse gehen nicht verloren
  - Sie werden in eine Warteschlange gestellt
  - Der zuständige Event-Handler wird direkt nach Ende des aktiven Handlers gestartet

No-Go #1:
Nichts tun außer warten

- Erste wichtige Grundregel
  - Event-Handler sollten nie schlafen
  - Event-Handler sollten nie schlafen
  - Event-Handler sollten nie schlafen
  - …
- Andernfalls behindern sie die Reaktion auf Ereignisse, die
  - währenddessen bearbeitet werden müssen oder
  - zumindest währenddessen bearbeitet werden könnten

No-Go #2:
Zu viel auf einmal tun

- Zweite wichtige Grundregel
  - Event-Handler sollten nicht zu lange laufen und
  - sind notfalls an geeigneten Stellen „aufzubrechen“
- Andernfalls behindern sie die Reaktion auf Ereignisse, die eventuell vorrangig bearbeitet werden müssen oder sollten
- Diese garantierte Unterbrechungsfreiheit
  - vereinfacht vieles im Vergleich zu Multi-Threading
  - macht aber „kurz-laufende“ Event-Handler zwingend
- In sehr eingeschränktem Umfang (zwei Stufen) wird die Priorisierung von Handlern in Tcl unterstützt

Kurz-laufende Event-Handler

- Faustformeln zur Latenzzeit:
  - Typisch geringer als die halbe, mit der Aufrufhäufigkeit gewichtete, durchschnittliche Laufzeit aller Event-Handler
  - Maximal die Laufzeit des längsten Event-Handlers (ohne Berücksichtigung der Frequenz anderer, möglicherweise zwischenzeitlich eintreffender Ereignisse)
- **Die obigen Faustformeln sind absolut ungeeignet, um ggf. „weiche“ oder gar „harte Echtzeifähigkeit” zu beurteilen!**

Aufbrechen von Event-Handlern

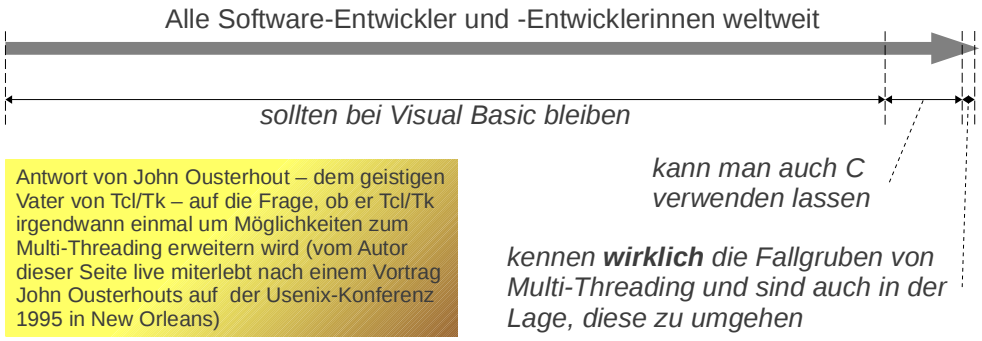
- Grundlegende Technik:
  - **Iteration in Form potenziell langlaufender Schleifen vermeiden**
  - **Stattdessen „Tail-Recursion“ unter Einbeziehung der Event-Loop**
- Realisierung:
  - Tcl-Befehl `after idle` *`continuation`* …
  - Dabei steht *`continuation`* … für die geeignete Fortsetzung des laufenden Handlers

Rekursiver Einstieg in die Event-Loop

- **Bitte aber nur bei wirklichem Verstehen der Problematik des rekursiven Einstiegs Tk-Event-Loop:**
  - Innerhalb von EventHandlern Tcl-Befehl `update`
- In Tk eventuell erforderlich in Event-Handlern, die visuelle Attribute des GUI verändern:
  - Tcl-Befehl `update idletasks`

Ist Multi-Threading die bessere Alternative?

- „Event Driven“ und „Multi-Threading“ Design
  - lösen ähnliche bzw. teilweise dieselben Probleme ...
  - … auf jeweils sehr unterschiedliche Art und Weise



Antwort von John Ousterhout – dem geistigen Vater von Tcl/Tk – auf die Frage, ob er Tcl/Tk irgendwann einmal um Möglichkeiten zum Multi-Threading erweitern wird (vom Autor dieser Seite live miterlebt nach einem Vortrag John Ousterhouts auf der Usenix-Konferenz 1995 in New Orleans)

Pro
Multi-Threading (1)

- **Multithreading skaliert direkt in Abhängigkeit von der eingesetzten Hardware (Anzahl Cores)**
  - Voraussetzung: angemessene Architektur / passendes Design
  - Insbesondere auch beim Downsizing (“Right-Sizing”)
- In einfachen Fällen auch von „weniger Geübten“ beherrschbar
  - Geeignete Entwurfsmuster (Design Patterns) sind dabei einzuhalten!
  - Solche sind auch ausführlich in der Fachliteratur diskutiert

Fortsetzung

Pro
Multi-Threading (2)

Fortsetzung

- Entsprechende Denk- und Vorgehensweisen werden bei Software-Entwickler(inne)n heute stärker geschult als früher
- **Hoher „Coolness-Faktor“ durch positive Konnotation im Zusammenhang mit „modern“ und „schnell“:**
  - Multi-Core
  - Hyper-Threading

Contra Multithreading (1)

- **Testergebnisse sind in der Regel nicht gezielt reproduzierbar!**
  - Behindert automatisiertes Testen und “Test Driven Development” (TDD)
  - Im Realbertrieb (gelegentlich) auftretende Fehlersituationen lassen sich für die nähere Untersuchung nicht gezielt reproduzieren
  - Wirksamkeit von Fehlerkorrekturen sind schwer nachweisbar

Fortsetzung

Contra Multithreading (2)

Fortsetzung

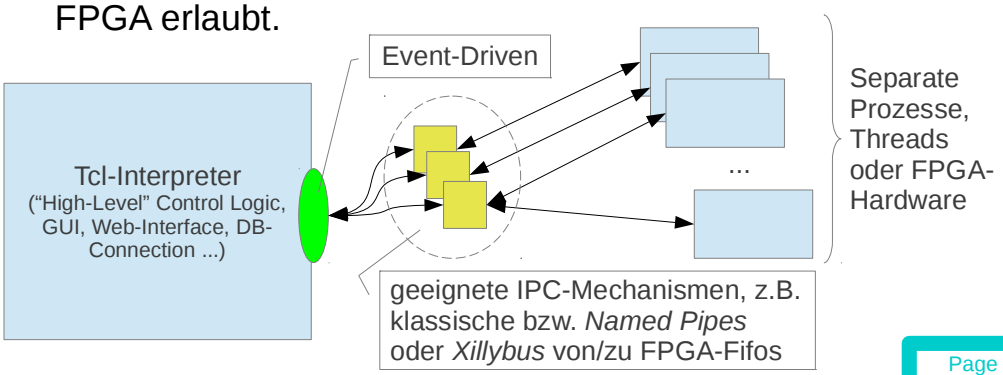
- **Viel Design-Erfahrung erforderlich, insbesondere im korrekten Umgang mit Synchronisationsmechanismen (Mutex, Semaphoren, Spin-Locks …)**
  - Zu wenige → Risiko von Daten-Inkonsistenzen
  - Zu viele → Risiko von Deadlocks
- Hoher „Coolness-Faktor“ … bei gleichzeitiger Unkenntnis der Alternativen
  - *Wer als Werkzeug nur den Hammer kennt, dem scheint die Welt voll von Nägeln zu sein!*

Auswahlkriterien *Multi-Threading* vs. *Event-Driven*

- Multi-Threading ist um so leichter beherrschbar,
  - je mehr es nur ein privater und kein gemeinsamer Datenbestand ist, den jeder Thread bearbeitet und
  - am Ende nur die Ergebnisse zusammenzuführen sind
- Event-Driven Designs werden um so komplexer,
  - je mehr lange laufende Schleifen aufzubrechen sind,
  - insbesondere wenn tiefe oder gar rekursive Hierarchien von Funktionsaufrufen beteiligt sind
- Praktische Probleme liegen meist „irgendwo dazwischen“ und insofern sollte stets eine sorgfältige Abwägung erfolgen

Mit „Multi-Core“ skalierende Tcl/Tk Architekturen

- Tcl selbst ist zwar „Single-Threaded“ und „Event-Driven“
  - lässt sich aber in eine „parallele Architektur“ einbetten,
  - welche ggf. mit der Anzahl von CPU-Cores skaliert …
  - … und dabei auch die Realisierung zeitkritischer Teile im FPGA erlaubt.



Doch *“grau, teurer Freund, ist alle Theorie ...”*

… und deshalb nun zur Praxis!

Zwischenfragen und Anregungen sind dabei jederzeit willkommen!

Das war‘s

Noch Fragen?

Danke für Ihre Aufmerksamkeit