Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Virtuelle Maschinen

Virtualisierung

- Motivation von Virtualisierung:
  - Abstraktion der Maschine und Speicher
  - Abstraktion und einfache Wiederverwendung von Algorithmen
  - Abstraktion von Kommunikation
  - Abstraktion von Parallelisierung und Verteilung!
  - Abstraktion der Maschineninstruktion ⇒ Instruction Set Architecture (ISA) ⇒ ISA kann physisch oder nur virtuell existieren
- Virtualisierung *V* ist die Abstraktion und Vereinheitlichung von:
  - Mikroprozessor ISA
  - Multitasking, Synchronisation
  - Sensoren, per. Geräte, Aktoren
  - Kommunikation und Konnektivität (Protokolle)
  - Speicher (GC, Tupelräume)
  - Semantische Daten, Kontext und Ort

Virtualisierung

Virtualisierung *V* transformiert i.A. eine Applikationsprogrammierschnittstelle (API) in eine andere API höherer Komplexität.

$$V : API_i \rightarrow API_o, \Theta(API_o) \gg \Theta(API_i)$$

- VM für eingebettete Systeme:
  - Lua (eLua, 32 Bit, 64kB RAM, 256kB ROM, GC, speed-down 1:100)
  - FORTH (Mecrisp, 32 Bit (≥8), 1kB RAM, 16 kB ROM, JIT!, no GC, speed-down 1:10)

[A]

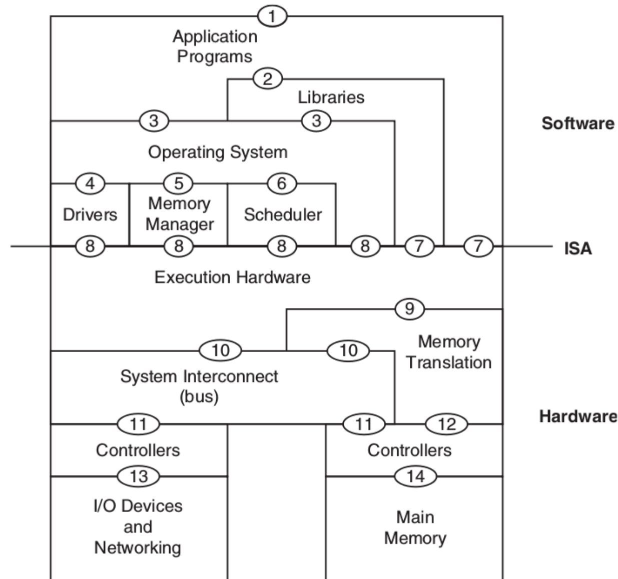
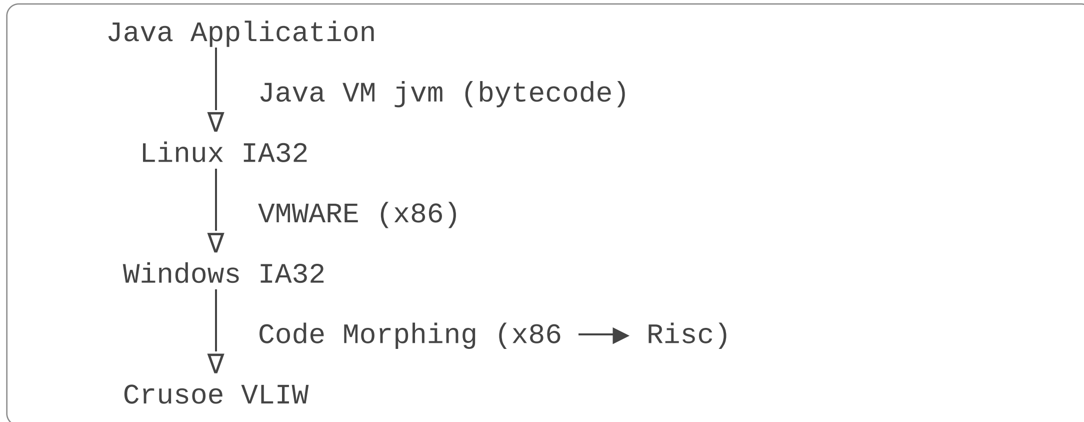


Abb. 1. Computer System Architektur mit verschiedenen Ebenen (Software, Betriebssystem, Hardware) → verschiedene Virtualisierungsebenen

Virtualisierung



Bsp. 1. Drei Ebenen von VMs. Eine Java-Anwendung, die auf einer Java-VM ausgeführt wird (Linux OS), die auf einer System-VM ausgeführt wird (Windows OS), die wiederum auf einem Code Morphing Prozessor VM läuft (VLIW: Very Large Instruction Word)

Prozessvirtualisierung

- Einzelne Ausführungsprozesse werden virtualisiert
- Häufig werden Programme in einer eigenen Metasprache und einem speziellen Anweisungsformat ausgeführt (Bytecode, Text)
  - Java Virtual Machine (jvm)
  - Lua Virtual Machine (lua, luajit, plvm)
  - JavaScript Virtual Machine (Google v8, jerryscript, spidermonkey)
  - OCaML Bytecode VM (funktionale Programmierung)
- Benötigt spezielle Verabeitung der Programme (nur auf dieser VM ausführbar)
- Abstraktion von Hardware und Betriebssystemebene

[A]

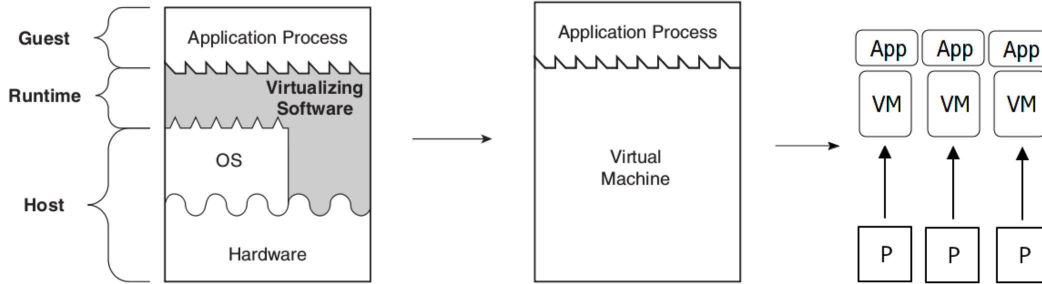


Abb. 2. Prozessvirtualisierung (Abstraktion von Hardware und Software) → Highlevel Language VM → Parallelisierung auf VM Instanzebene (1:1 Zuordnung P:VM)

Systemvirtualisierung

- Hier wird die Maschine (Hardware) virtualisiert
- Verschiedene Programme und Programmformate können konventionell ausgeführt werden
- Parallele Ausführung verschiedener gekapselter Betriebssysteme
    - Parallelisierung kann entsprechend des Betriebssystem auf OS und Prozessebene erfolgen
    - Die VM selbst kann parallele IO Ausführung nutzen

Systemvirtualisierung

[A]

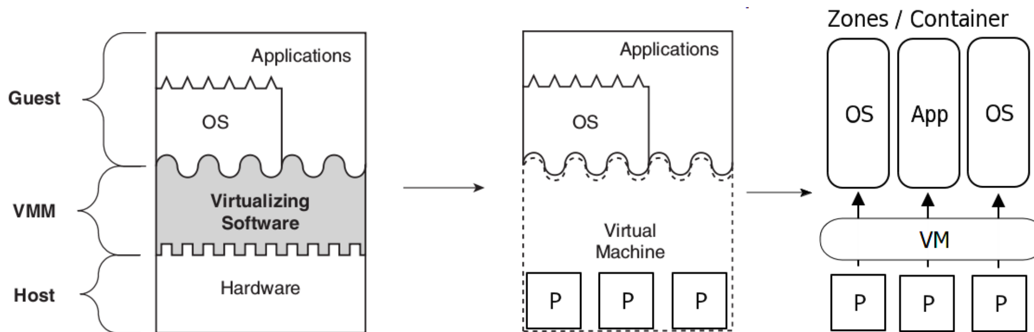


Abb. 3. Systemvirtualisierung (Abstraktion von Hardware) und Parallelisierung (1:1 Zuordnung einer VM Zone zu virtuellen Prozessor)

Systemvirtualisierung

[A]

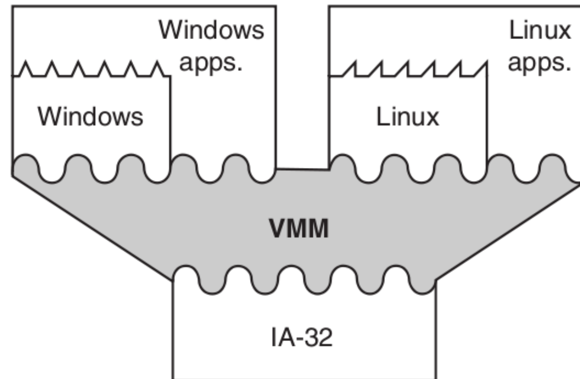


Abb. 4. Beispiel: Zwei Betriebssysteme werden auf virtualisierter Hardware (parallel) ausgeführt

Compiler

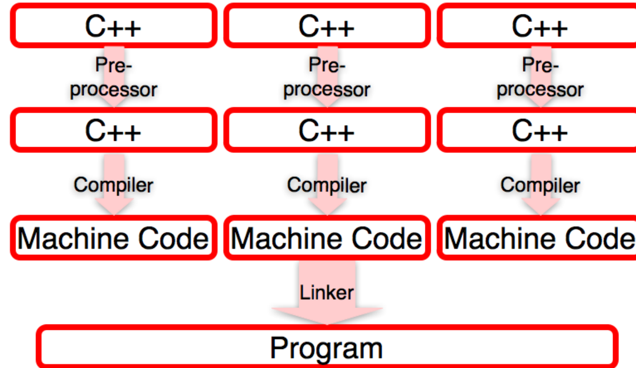


Abb. 5. Klassischer Softwareentwurf mit Compiler und Linker (C)

Interpreter

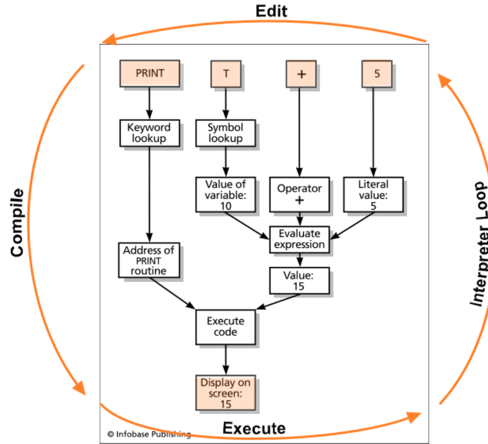


Abb. 6. Edit - Compile - Execute Zyklus bei einem Interpreter

Virtuelle Maschine und Bytecode Interpreter

- Die Übersetzung des Quelltextes in Bytecode kann vor und während der Ausführung des Programms erfolgen!

- Bytecode kann privat und abstrakt sein (nur interne Datenstrukturen) oder öffentlich in kodierter Binär- oder Assemblerform.

- Ausführung des Bytecodes durch VM. Vorteile:

  - Unabhängig von Hostplattform
  - Virtuelle Maschine kann optimiert werden ohne dass der Bytecode neu erzeugt werden muss
  - Abstraktion der Speicherverwaltung (automatisches Speichermanagement)
  - Parallelisierung kann durch VM automatisch erfolgen (wenn möglich)
  - Abstraktion der IO Schnittstellen und Kommunikation

Virtuelle Maschine und Bytecode Interpreter

[Python]

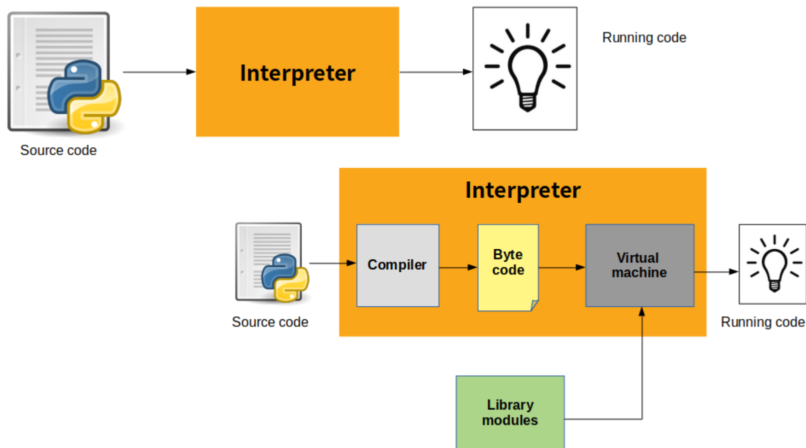


Abb. 7. Vergleich Interpreter mit Bytcode Compiler-Interpreter System

Bytecode Ausführung

Bytecode wird i.A. durch eine große Ausführungsschleife ausgeführt. Diese besteht i.A. aus einem Dekoder (Mehrfachauswahl) und operationalen Programmabschnitten. Nach jeder Instruktion kann ein Taskwechsel erfolgen (Scheduling Point) um fein granuliertes Multitasking zu ermöglichen.

```
while not terminated do
  operation = decode(code[next])
  select operation.code of
    case OP1 => DO IT
    case OP2 => DO IT ..
  endcase
  compute next
  if scheduling point then yield endif
done
```

Serialisierung

- Da Bytecode unabhängig von der Hostplattform sein sollte, kann Bytecode einfach von einer Maschine zu einer anderen übertragen und ausgeführt werden

- Dazu ist eine **Serialisierung** von Daten und Code erforderlich (Flache Liste von Bytes), mit anschließender **Deserialisierung** (Wiederherstellung der Daten- und Codestruktur)

Serialisierung

[A]

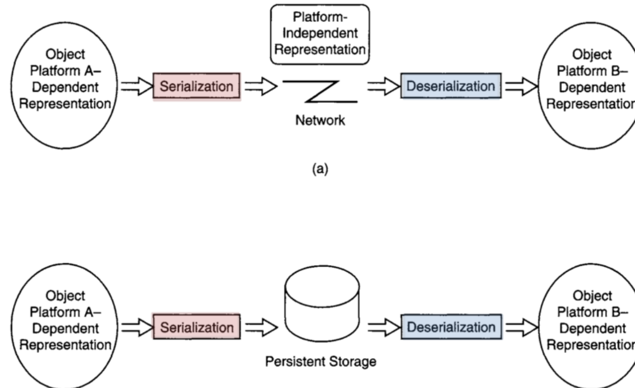


Abb. 8. Serialisierung transformiert ein Objekt in ein implementierungs- und systemunabhängiges Format (a) für die Netzwerkübertragung (Text!) (b) für die persistente Speicherung

[Peter Cawley]

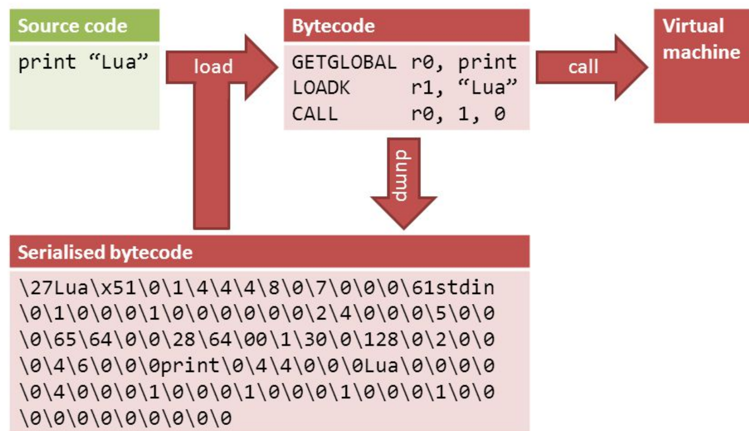


Abb. 9. Bytecode VM

Lua Bytecode

- Lua Bytecode besteht aus Instruktionen mit konstanter Wortlänge (4 Byte)

[102]

0–5	6–13	14–22	23–31
OP	A	B	C
OP	A	Bx	
OP	A	sBx	

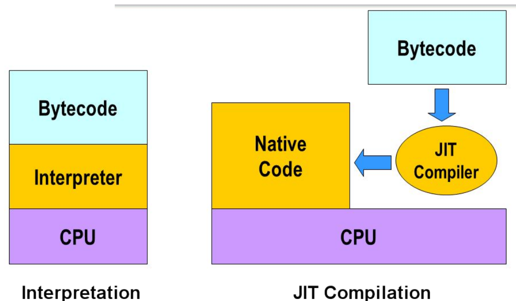
Instruction layout

```
function max (a,b)
  local m = a          1 MOVE     2 0 0    ; R(2) = R(0)
  if b > a then        2 LT       0 0 1    ; R(0) < R(1) ?
    m = b              3 JMP      1        ; to 5 (4+1)
  end                  4 MOVE     2 1 0    ; R(2) = R(1)
  return m             5 RETURN   2 2 0    ; return R(2)
end                    6 RETURN   0 1 0    ; return
```

Abb. 10. Lua Byte Code Format und Beispiel

JIT Compiler

- Neben der Bytecode Ausführung kann während der Laufzeit des Programms der Bytecode stückweise optimiert werden → Erzeugung von Maschinencode durch einen Just-in-Time Compiler (JIT)



[King Fahd University of Petroleum and Minerals]

Abb. 11. Vergleich Bytecode VM Interpreter mit JIT Compiler-Interpreter Systemen

Virtualisierung

Aufgabe

1. Überlege Dir die Vor- und Nachteile von
    - Compilern
    - Interpretern
    - Bytecode Interpretern
    - JIT Compiler und Interpreter
2. Was sind die Bewertungskriterien?
3. Welche Ausführungsarchitekturen könnten für Parallele Systeme mit VMs geeignet sein?
4. Wo gibt es Engpässe?

Virtualisierung

- Rechenleistung, Speicherbedarf, und Ein-Ausgabe Durchsatz/Latenz sind wichtige Metriken für den Einsatz in Eingebetteten Systemen → Sensornetzwerke / Internet der Dinge (IoT)

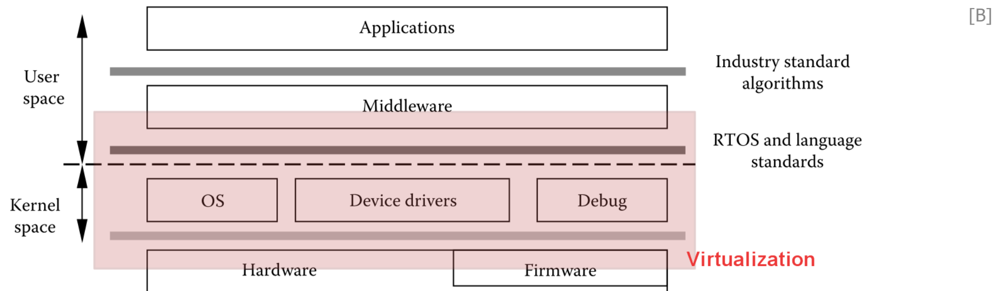


Abb. 12. Typische Datenverarbeitungsebenen in Eingebetteten Systemen und Virtualisierung

NODE.JS Architektur

- Neben der Bytecode+JIT VM gibt es einen großen Bereich für Ein- und Ausgabeverarbeitung (IO) und nativen Bibliothekenbindungen (Wrappers)

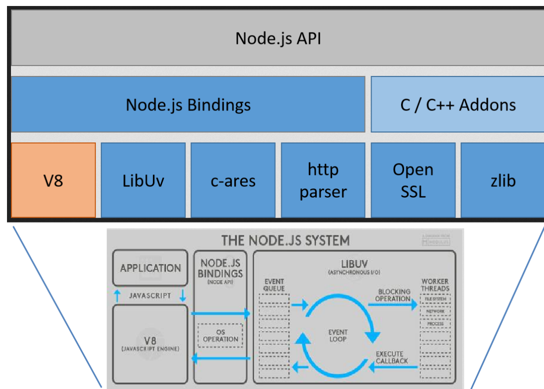


Abb. 13. Die "Eventloop" und die UV Bibliothek sind neben der Google V8 JS VM zentraler Architekturbestandteile

Performanz von Interpretern

Dhrystone Benchmark (Kombination aus Berechnung, Objekten, Arrays, Strings, Funktionen)

VM/OS-ARCH	Linux-i686	Linux-armv6l (PI-3B)	Linux-armv6l (PI Zero)
python2.7 [1]	105k/s	23k/s	4k/s
lua 5.1 [2]	140k/s	36k/s	-
luajit 2.0.5X/lvm [2]	660k/s	108k/s	40k/s
jerryscript 1.1.7X [3]	45k/s	-	-
nodejs/V8-4 [3]	6300k/s	879k/s	40k/s
C/gcc	18000k/s	3700k/s	?

(1: Python, 2: Lua, 3: JavaScript)

Speicherbedarf von Interpretern

Dhrystone Benchmark (Kombination aus Berechnung, Objekten, Arrays, Strings, Funktionen)

VM/OS-ARCH	**Linux-i686**	**Linux-armv6l (PI-3B)**	**Linux-armv6l (PI Zero)**
python2.7	-	-	-
lua 5.1	-	-	-
luajit 2.0.5X/lvm	2MB	2MB	-
jerryscript 1.1.7X	2MB	-	-
nodejs 4/10	24MB	32MB	-
C/gcc	1MB	?	?

Automatisches Speichermanagement

- In C/C++ muss für jedes zur Laufzeit dynamisch erzeugte Datenobjekt (Array, String, Record, ..) immer explizit Speicherplatz im Hauptspeicher angefragt werden (`malloc`,`new`) und wieder frei gegeben werden wenn das Datenobjket nicht mehr benötigt wird (`free`,`delete`)

- In Skriptsprachen gibt es i.A. ein automatisches Speichermanagement mit einem sog. **Garbage Collector** und Objektreferenzierung

- Jedes Datenobjekt welches verwendet wird (z.B. in Variablen oder Funktionen) besitzt eine Referenz

- Es muss eine Wurzeltabelle geben von der aus alle Referenzen auf verwendete Objekte auffindbar sind

Objekte und Referenzen

- Objekte: In C/C++/JAVA usw: Variablen, in Lua/JavaScript/OCaML: Werte!

```
local o1 = { a=1, b={1,2,3,4} }
local o2 = { op = o1, index=2 }
function show (o) print (o2.op.a-o.a } end;
show(o1)
```

Bsp. 2. Beispiel von Objektreferenzierungen in Lua: Wo existieren Referenzen?

Variablen, Referenzen, Werte in Lua

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT ref1 ref2

Objekte und Referenzen

[A]

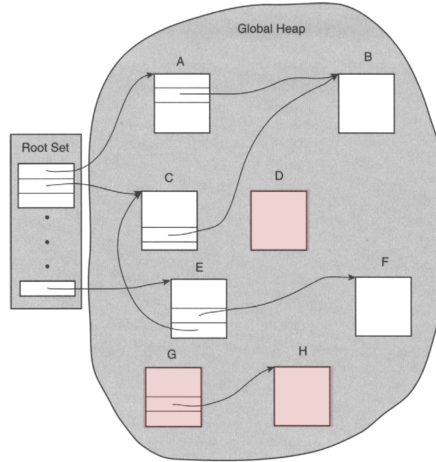


Abb. 14. Verschiedene Objekte: Objekte die nicht mehr referenziert sind, werden freigegeben

Speicherarchitektur von Programmen

Code

Statischer oder dynamischer Speicherbereich mit Maschinenanweisungen (und eingebetteten Daten als Anweisungsoperanden)

Heap

Datenobjekte mit längerer Lebensdauer (Tabellen- oder Listenstruktur) → Benötigt Speichermanagement

Stack

Datenobjekte mit kurzer Lebensdauer (Stapelstruktur) → Benötigt kein Speichermanagement

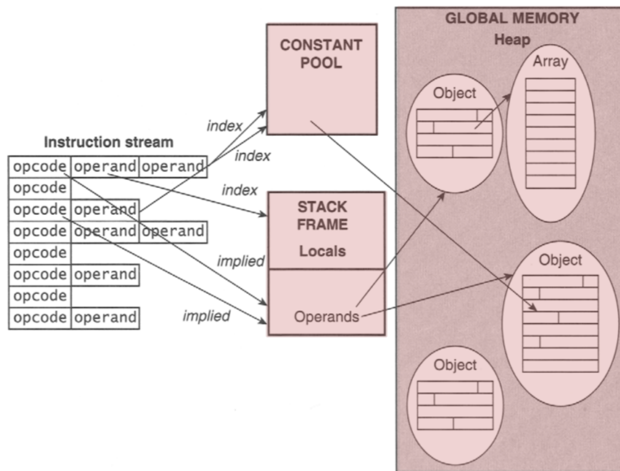
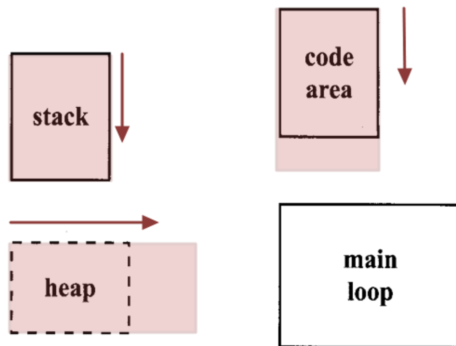


Abb. 15. Speicherhierarchie von Programmen und Zusammenhang mit Maschinenanweisungen

Speicherorganisation von Virtuellen Maschinen

- Speicherbereiche sind segmentiert
- Statisch
  - Größe
  - Inhalten (Daten)
- Dynamisch
  - Größe
  - Inhalten (Daten)
- Fragmentierung
  - Stack, Code: nein
  - Heap: ja!



Speicher

- Die meisten Objekte in Programmiersprachen benötigen Speicher
  - Lokale, temporäre, und globale Variablen
  - Objekte
  - Arrays
  - Zeichenketten
  - Funktionen (!)
  - Native Codereferenzen (Speicheradresse)
- Der Kontext von Speicherobjekten ist wichtig:
  - **Modul** → limitierte Programmsichtbarkeit
  - **Lokal** → stark limitierte Programmsichtbarkeit
  - **Global** → uneingeschränkte Sichtbarkeit

Speicher und Variablen

- Beispiele: Die globale Variable *Global* ist überall, auch außerhalb des Definitionsmoduls, sichtbar, die Variable *lokal* ist im gesamtem Modul und den Funktionen *f1* und *f2* sichtbar, wo hingegen *lokal1* und *lokal2* jeweils nur ihrem Funktionskontext sichtbar sind.

```
Global = 1
local lokal = 2
function f1 ()
  local lokal1 = 3
end
function f2 ()
  local lokal2 = 3
end
```

- Nur wenn Variablen bei der Ausführung sichtbar sind belegen sie Speicher!

Speicher und Variablen

- Ein Teil des Speicher ist vorbelegt:
  - Globale Variablen
  - Funktionen, ausführbarer Code, ...
- Ein anderer Teil des Speichers wird zur Laufzeit auf dem Stack und Heap belegt und benötigt Verwaltung:
  - Lokale und temporäre Variablen, ..
  - Funktionen, ausführbarer Code, ...
- Achtung: Bei Programmiersprachen bei denen Funktionen Werte erster Ordnung sind befindet sich auch Code im Heap oder auf dem Stack!

Polymorphie und Dynamische Typisierung

- Nativer Maschinencode unterscheidet Operanden nach ihren Datentypen → Speicherumfang von Datentypen ist sehr unterschiedlich (Boolean, Float64, Int16, Arrays, Records, ..)

- HLL Skript Programmierung bietet häufig dynamische Typisierung, d.h., eine Variable hat zunächst keinen Datentyp

    - *Der Datentyp einer Variable ergibt daher sich nur aus den Datentypen von konkreten Werten die die Variable referenziert!*

Welche Vor-und welche Nachteile hat dynamische Typisierung zur Laufzeit?

Polymorphie und Dynamische Typisierung

```
typedef struct {   typedef union {
  int t;             GCObject *gc;
  Value v;           void *p;
} TObject;           lua_Number n;
                     int b;
                   } Value;
```

Def. 1. Lua Werte werden als "tagged unions" repräsentiert (C)

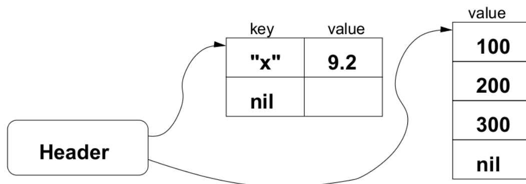


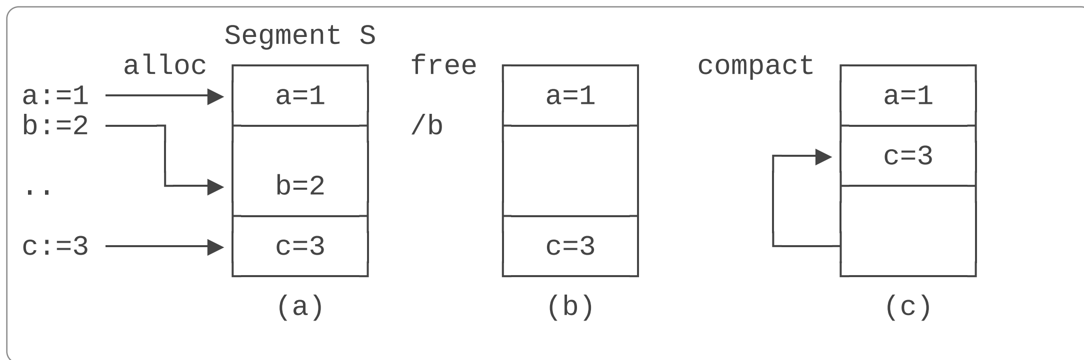
Abb. 16. Lua Tabellen (Arrays, Records) sind Referenzlisten auf weitere Objekte aus Werten

Speichersegmente und Fragmentierung

- Ein (statisches oder dynamische) Speichersegment ist ein linearer zusammenhängender Bereich
  - Mit Virtuellen Speichermanagement kann ein Segment aus einer Reihe verstreuter physischer Speicherbereiche zusammengesetzt werden
  - Es wird eine Speicherzuordnungsfunktion $MMU(VA) \rightarrow PA$ benötigt (LUT, nur in Hardware effizient)
- Hochsprachen haben kein Speichermodell, sondern nur Speicherobjekte (Variablen)
  - Man unterscheidet Name, Daten, Datenkontainer und Referenz auf Datenkontainer

- Für die Ausführung mit einer Maschinen müssen diese abstrakten Speicherobjekt
  - Speicherbereiche in Segmenten *Seg* belegen (allocate(O))
  - Diese Speicherbereiche referenzieren $MMU(O) \rightarrow Seg, SA$
  - Speicherbereiche in Segmenten müssen wieder frei gegeben werden wenn Objekt nicht mehr verwendet wird ($free(O)$)

> Durch Speicherallokation und Freigabe Zyklen (Speicherbedarf der Objekte variiert) kommt es zu Fragmentierung, d.h., der einst lineare Speicherbereich eines Segments besteht aus einer irregulären Folge aus freien und belegten Bereichen



Bsp. 3. (a) Speicherbelegung, (b) Freigabe, und (c) Kompaktierung

Welches Problem tritt bei Kompaktierung auf?

Speicherverwaltung

Listen

Freie und belegte Speicherbereiche werden durch Listen (einfach- oder doppelt verkettet) verwaltet

Tabellen

Tabellen (Hashtabellen) werden zur Speicherverwaltung verwendet

Manuelle Verwaltung

Speicherbelegung explizit vom Programmierer und teils vom Compiler auszuführen

Vorteile

- Optimale Speicherbelegung (tatsächlich naiv und falsch) und Performanz/Effizienz

Nachteile

- Speicherlecks (nicht mehr benötigter Speicher wird nicht frei gegeben)
- Mehrfachfreigabe (Inkonsistenz der Speicherverwaltung)
- Benutzung von Speicherobjekten nach Freigabe

Speicherverwaltung

Automatische Verwaltung

Die Freigabe von Speicherbereichen wird automatisch durch einen **Garbage Collector** durchgeführt. Dieser bestimmt auch automatisch ob Speicherobjekte noch benötigt werden.

Vorteile

- Speicherlecks werden vermieden
- Keine Mehrfachfreigabe (Inkonsistenz der Speicherverwaltung)
- Keine Benutzung von Speicherobjekten nach Freigabe

Nachteile

- Nicht optimale Speicherbelegung und schlechtere Performanz/Effizienz

Garbage Collection

- Wie soll der GC heraus finden ob ein Objekt noch benötigt wird?

1. Referenzzähler wird jedem Objekt hinzugefügt und bei jeder neuen Referenzierung erhöht (Allocate). Ein Objekt mit Referenzwert Null kann frei gegeben werden.
2. Der GC rechnet Objektabhängkeiten aus und markiert Objekte die freigegeben werden können

Beispiel von Referenzen und Sichtbarkeit (Programmkontext)

```
local o1 = { x=100,y=200 }e1           [e1.ref=1(o1)]
local o2 = { pos=o1, color="red" }e2  [e1.ref=2(o2,o1)]
function draw (o)
  local o3 = { pos=o.pos, color="white" }e3  [e3.ref=1(o3), e1.ref=4(o3,..)]
  local o4 = { pos={x=o.pos.x,y=o.pos.y}e4, color="black" } [e1.ref=5(o4,o4,..)]
  draw(o3)
  draw(o4)
  → [e3.ref=0, e4.ref=0, e1.ref=3, e2.ref=2]
end
draw(o2)  [e2.ref=2(o,o2), e1.ref=3(o,o2,o1)]
o2=nil    [e2.ref=0, e1.ref=1(o1)]
o1=nil    [e1.ref=0]
```

- Wichtig: Referenzzähler beziehen sich auf Werte (e_i), nicht auf die Variablen selber (die nur Zeiger auf Werte besitzen)
  - D.h. wenn ein Wert als Referenz an eine weitere Variable zugewiesen wird erhöht sich der Referenzzähler!

GC: Referenzzählung

- Wenn ein Objekt (also ein Wert) erzeugt wird, wird der Referenzzähler auf Null gesetzt
- Jede weitere Referenzierung erhöht den Zähler um eins (also bereits das zuweisen eines Objektes an eine variable!)
- Wird eine Referenzierung vom Objekt entfernt (explizit → `=null`, oder implizit durch Entfernen des referenzierenden Objektes), dann wird der Zähler um eins erniedrigt
- Ist der Zähler wieder Null, dann wird der Speicher freigegeben

GC: Referenzzählung

Nachteile

1. Zyklische Referenzen können nicht aufgelöst werden:

```
local o1 = { x=100,y=100 }
local o2 = { prev=o1 }
o1.next = o2
```

GC: Referenzzählung

2. Daher werden u.U. nicht alle Objekte freigegeben → Speicherlecks
3. Der Referenzzähler belegt selber Speicher (gehört zum Objekt). Bei low-resource Plattformen wird z.B. eine Datenwortbreite von 16 Bit für den Zähler verwendet, und der maximale Referenzwert ist 65536 (*jerryscript* ist so ein Kandidat)!
4. Allokation ist langsamer (verlangsamt Zuweisung)

Vorteile

1. Einfach zu implementieren
2. Freigabe ist effizient (wenig Rechenaufwand)

GC: Mark & Sweep

- Bestimmte Objekte sind direkt zugänglich, und es muss herausgefunden werden welche Objekte erreichbar sind → Graphensuche

- Es gibt einen Stammsatz von Speicherplätzen im Programm, dessen Objekte direkt erreichbar sind.

GC: Mark & Sweep

Mark-and-Sweep verläuft in zwei Phasen

Markierungsphase

- Erreichbare Objekte suchen:
  - Der Stammsatz wird zu einer Arbeitsliste hinzugefügt
  - Solange die Arbeitsliste nicht leer ist wird ein Objekt von der Liste entfernt. Wenn es nicht markiert ist, markiere es als erreichbar und alle Objekte die davon aus erreichbar sind werden zur Arbeitsliste hinzugefügt.

Sweepphase

- Für alle belegten Objekte:
  - Ist das Objekt nicht markiert so lösche es (Speicher freigeben)
  - Ist das Objekt markiert, lösche die Markierung

GC: Mark & Sweep

Vorteile

1. Auch zyklische Referenzen können über die Arbeitsliste aufgelöst werden
2. Alle Objekte können frei gegeben werden
3. Allokation schnell

Nachteile

1. Rechenintensiv und langsam!

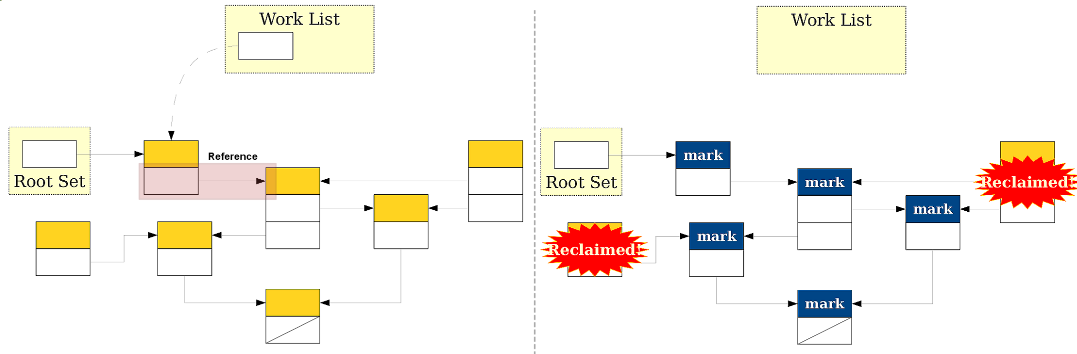


Abb. 17. Beispiel eines M&S Durchlaufes mit anschließender Freigabe von nicht mehr erreichbaren == benötigten Objekten

Multitasking

- Auf einem Rechnerknoten gibt es eine Vielzahl von Aufgaben die bearbeitet werden müssen:
  - Datenverarbeitung
  - Kommunikation
  - Speicherung und Ein/Ausgabe
- Zum Teil müssten Aufgaben parallel verarbeitet werden → Multitasking → Parallelisierung → nur bedingt möglich → Task Scheduling erforderlich
- Es gibt verschiedene Ausführungsebenen für Tasks:
  1. Prozesse
  2. Threads
  3. Fibers/Koroutinen

Prozesse, Threads, Fibers

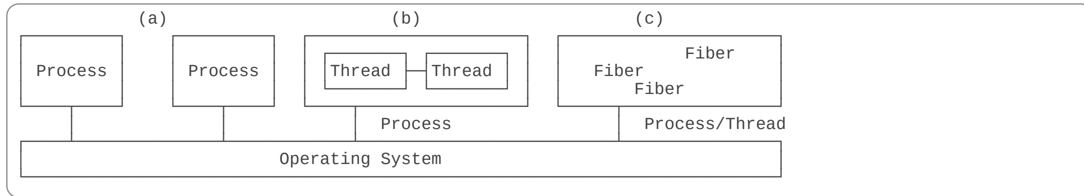


Abb. 18. (a) Zwei isolierte Prozess (b) Zwei gekoppelte Threads in einem Prozess (c) Koroutine (Fibers), nicht isoliert mit einem Kontrollpfad

Virtualisierung und Parallelisierung

> Grundkonzept: Aufteilung einer Berechnung in eine Menge von Teilberechnungen die auf verschiedenen Prozessoren parallel ausgeführt werden.

- Auf generischen Mehrprozessorsystemen (Multicore) wird Multithreading unf Multiprocessing Kontrollpfade
- Auf spezialisierten Mehrprozessorsystemen (GPU) wird sowohl Mutlithreading als auch Datenpfadparallelität ausgenutzt (Mehrere Kontroll- und Datenpfade)

Aber: Bei Virtuellen Maschinen lassen sich diese Konzepte nicht direkt übertragen!

- Automatisches Speichermanagement ("stop the world" Phase) schwierig zur parallelisieren (Konkurrenz und Ressourcenkonflikte)
- Speicherraum einer VM ist gekapselt (man spricht von einer Ausführungsinstanz oder einem isolierten Kontainer)
- VM arbeiten daher i.A. mit nur einem einzigen Kontrollfluss (auf Programmierebene)!

Architekturen Paralleler VMs

1. Background/Foreground Systeme
    - Es gibt einem Hauptthread der das Programm ausführt (VM Main Loop)
    - Es gibt einen Nebenthread der die GC ausführt
2. Multithreading
    - Es gibt einen oder mehrere Threads die das Programm ausführen
    - Es gibt einen oder mehrere Threads die den GC ausführen

Architekturen Paralleler VMs

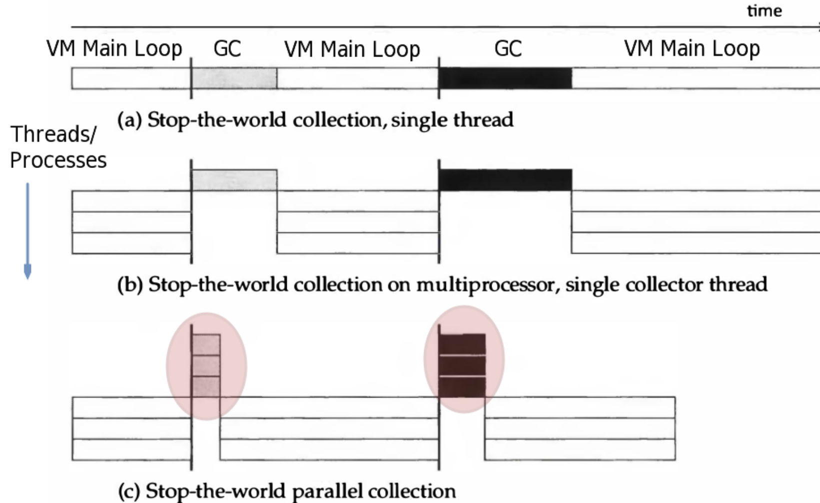


Abb. 19. (a) Sequenzielles System (1 Prozess/Thread) Stop-the-world GC (b) Multithreading Stop-the-world GC (c) Multithreading und paralleler/nebenläufiger GC

Parallelisierung des GC Algorithmus

1. Parallele Markierung
2. Paralleles Kopieren
3. Paralleles Sweeping
4. Paralleles Kompaktieren

Es wird weiterhin Synchronisation zw. den parallel ausgeführten GC Threads und dem eigentlichen VM Ausführungsthread (oder mehreren) benötigt!

- Es gibt zwei **Motivationen**:
  - GC soll möglich wenig Rechenzeit des Hauptprogramms "kosten" (Horizontale Par.)
  - GC soll beschleunigt werden (Vertikale Par.)

Synchronisation

- Neben der Synchronisierung von Operationen an *Kollektordatenstrukturen* kann es auch erforderlich sein, *Operationen auf einzelnen Objekten* zu synchronisieren.
- Das *Markieren* ist im Prinzip eine idempotente Operation: Es spielt keine Rolle, ob ein Objekt mehr als einmal markiert wird.
- Wenn z.B. ein *Kollektor* jedoch einen Vektor von Markierungsbits verwendet, muss der Marker diese Bits atomar setzen
  - Befehlssätze moderner Prozessoren bieten nicht die Möglichkeit, ein einzelnes Bit in einem Wort oder Byte zu setzen → Setzen einer Markierung erfordert eine Schleife, die versucht, den Wert des gesamten Bytes atomar zu setzen.
  - Wenn das Markierungsbit im Header des Objekts enthalten ist oder der Markierungsvektor ein Vektor von Bytes ist (einer pro Objekt), ist keine Synchronisation erforderlich

Synchronisation

⇒ Also: Die Details von Rechnerarchitektur und GC Algorithmus bestimmen notwendige Synchronisation

- Zum GC Algorithmus gehören auch immer die **Datenstrukturen** und **Allokationsmethoden**!
- Schließlich muss die **Beendigung** einer Sammelphase korrekt bestimmt werden!
  - Die Verwendung paralleler Threads macht die Terminierungserkennung deutlich komplexer.
  - Grundsätzlich besteht das Problem darin, dass ein Thread möglicherweise versucht zu erkennen, ob die Phase beendet wurde, während ein anderer noch arbeitet und eine neue Phase startet!
- *Barrieren können z.B. für die Synchronisation bei der konkurrierenden/parallelen Kollektion eingesetzt werden*

Parallele vs. Konkurrierend

- Und wieder muss zwischen paralleler und konkurrierender Ausführung unterschieden werden, was zu unterschiedlichen Abläufen des GC und der VM Main Loop führen
- Weitere Unterscheidung durch **Horizontale und Vertikale Parallelisierung** (Zeit/Raumdimension)

[Jones, 2012, TGCH]

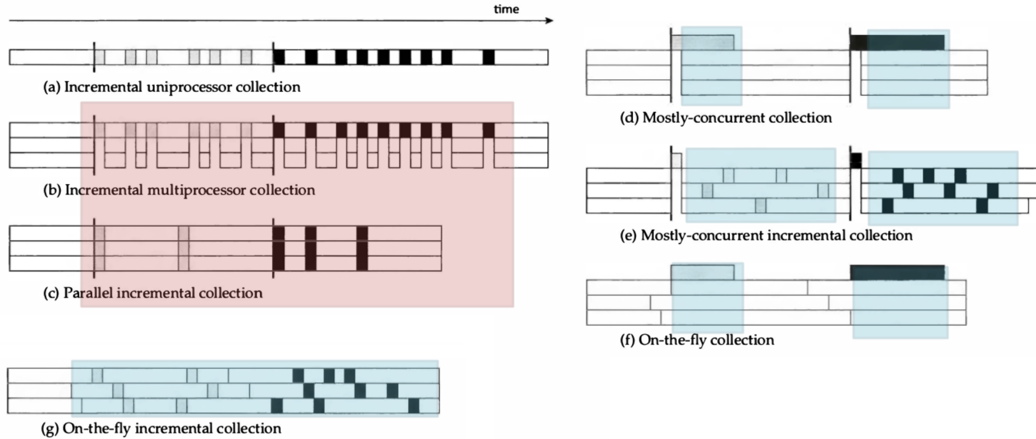


Abb. 20. (rot) Parallele und synchronisierte Ausführung (VM Main Loop angehalten) (blau) Konkurrierende Ausführung und überlappend mit VM Main Loop

Parallele VM Instanzen

- Parallelität daher durch mehrere VM Instanzen möglich.
    - Eine Instanz ↠ Ein Prozess (Thread)
    - Aber: Die Start- und Initialisierungszeit einer VM Instanz kann hoch sein (JS nodejs/v8: 100ms, LuaJit: 1ms)
    - Weiterhin ist der Speicherbedarf einer VM Instanz zu beachten ("baseline" ohne Nutzprogramm, nodejs: 20MB, LuaJit: 1MB)

> Bei hoher Initialisierungszeit einer VM Instanz muss diese im voraus gestartet werden (in-advance) und über eine Service Loop auf Prozesscode warten.
>
> Bei kleiner Initialisierungszeit einer VM Instanz kann ein VM Prozess ad-hoc und dynamisch gestartet werden.

Stack vs Register vs Speichermaschinen

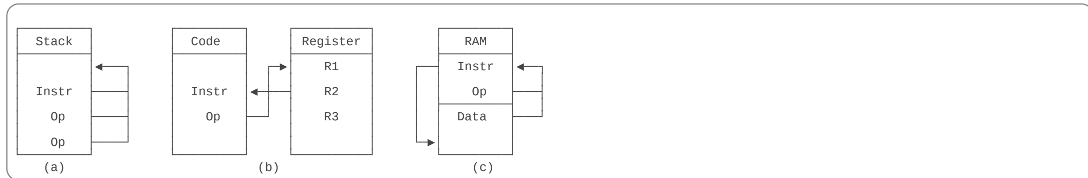
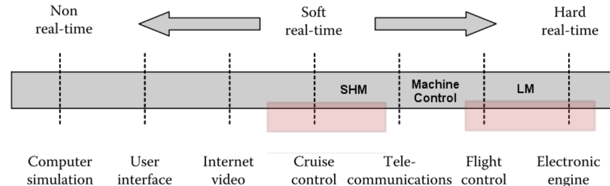


Abb. 21. (a) Stack Maschine (keine ext. Abhängigkeit) (b) Register Maschine (c) Speichermaschine

Parallelisierungsgrad (Multi-Instanz VM): Stack ≫ Register ≫ Speicher

Echtzeitverarbeitung

- Echtzeitverarbeitung bedeutet die Ausführung eines Tasks innerhalb eines vorgegebenen Zeitintervalls $[t_0, t_1]$
  - **Soft Realtime**: Zeitüberschreitung bei weicher Echtzeitanforderung → Tolerierbar
  - **Hard Realtime**: Zeitüberschreitung bei harter Echtzeitanforderung → Systemfehler



[B]

Abb. 22. Echtzeitspektrum: Weiche und harte Echtzeitanforderungen bei unterschiedlichen Applikationen

Echtzeitverarbeitung

[B]

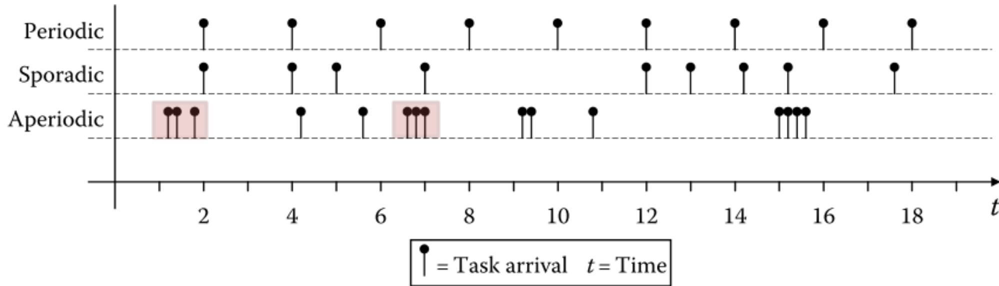


Abb. 23. Periodische, sporadische, und aperiodische Tasks

Echtzeitverarbeitung

Preemption

- Wenn schon nicht alles Tasks gleichzeitig verarbeitet werden können, dann wenigstens die wichtigsten vorrangig ausführen

- Das erfordert aber:
    - Festlegung einer Priorität
    - Bei Echtzeitanforderungen die Unterbrechung von unterlegenen (kleiner Priorität) Tasks

[B]

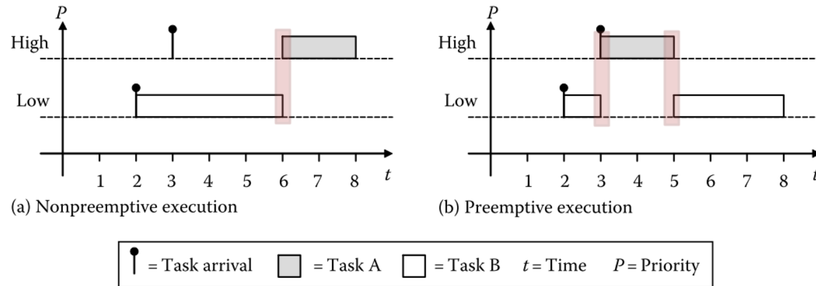


Abb. 24. Unterschied preemptive und nicht preemptive Ausführung von Tasks

Echtzeitverarbeitung

Virtuelle Maschinen

- Schon die Parallelisierung der Programmausführung ist schwierig und eher schwergewichtig!
- Dis Ausführungszeiten von Programmen sind nicht im voraus bekannt und deterministisch
  - JIT beeinflusst die Ausführungszeit zur Laufzeit signifikant (Übersetzung kostet Zeit, Maschinenkode spart Zeit)
  - Automatisches Speichermanagement (GC) führt zu teils stark variierenden Laufzeiten und Performanz von Programmen
  - Bei längerer Laufzeit kann sich die Performanz (Latenz/Datendurchsatz) verringern. Warum?

> Echtzeitfähigkeit bei automatischen Speichermanagement ("stop the world" Phase), JIT und umfangreiche Nutzung dynamischer Datenstrukturen kaum zu gewährleisten!

Multiinstanz VM: Beispiel (P)LVM

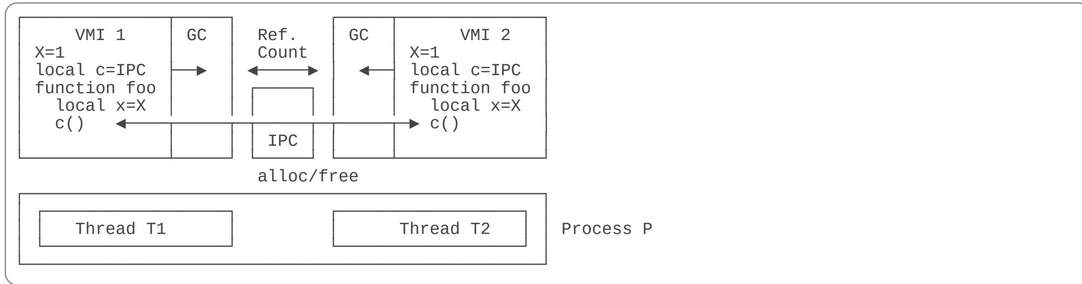


Abb. 25. Zwei Lua LVM Instanzen mit gemeinsamen eng gekoppelten Interprozeßkommunikationsobjekten und implizite GC Kopplung über Referenzzählung

Zusammenfassung

Die Parallelisierung von VMs ist stark eingeschränkt

Hauptsächlich Einschränkungen durch das autom. Speichermanagenent (GC)

Parallel ausgeführte getrennte VM Instanzen auf Prozessebene (Cluster) immer möglich (geringe Kopplung) → Interprozesskommunikation über Nachrichten optional mit generischen SM

Parallel ausgeführte VM Instanzen auf Threadebene mit gemeinsam geteilten Objekten und spezialisierten SM die teils durch GC behandelt werden ist die optimale Lösung!