Noninterference in the Take-Grant Model for the seL4 Microkernel

Andrea Kuchar

INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Lehr- und Forschungseinheit für theoretische Informatik

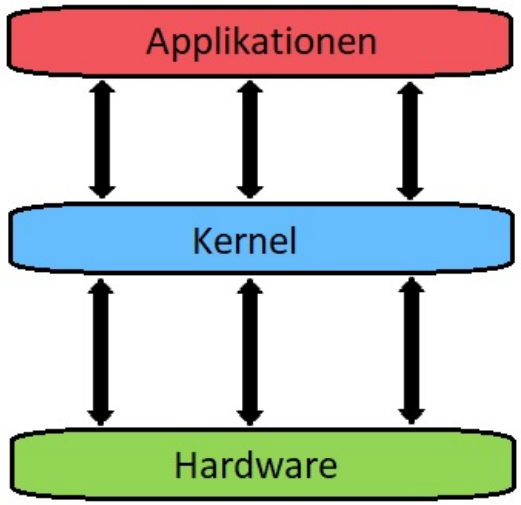
12. September 2018

Überblick

1. Motivation
2. seL4
   - Kernel Objekte
   - Memory Allocation Model
3. Take-Grant Model
4. Noninterference
5. Formalisierung des Take-Grant Models

Motivation

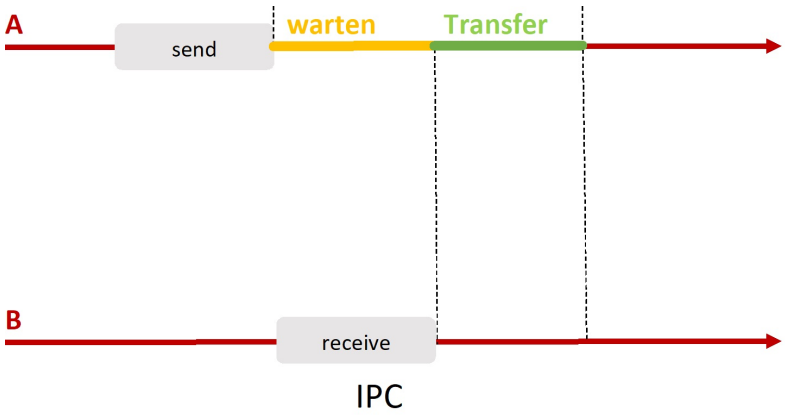
- Kernel = Schlüsselkomponente für sichere Systeme
- Zugriffsteuerung auf Hardwarekomponenten
- Fehler im Kernel kann die Sicherheit und Verlässlichkeit des kompletten Systems zum erliegen bringen.
- Monolithische Designs:
  - Große Menge Code
  - Integration weiterer Funktionen.
  - Folge: Grundlegend schwach durch größere Anfällgkeit für Bugs.

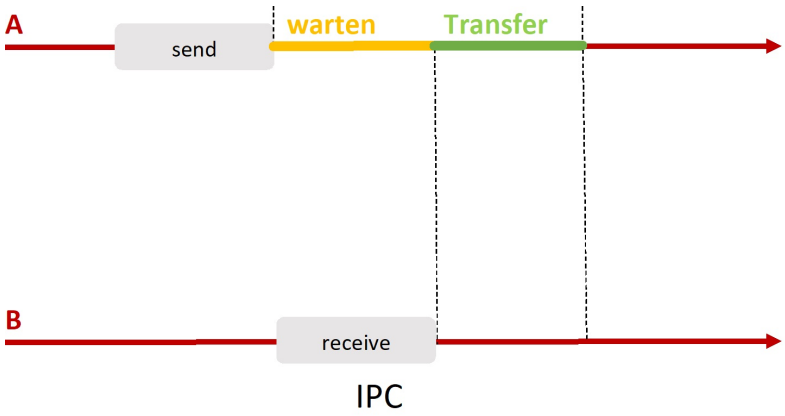


- Microkernel:
  Konzentration auf die fundamentalen Funktionen eines Kernels:
  z.B. Interprozesskommunikation, Scheduling, Speicherverwaltung
- Durch Microkernels: Fehleranfälligkeit verringern (weniger Code $\Rightarrow$ Fehlerfreiheit formell verifizierbar)

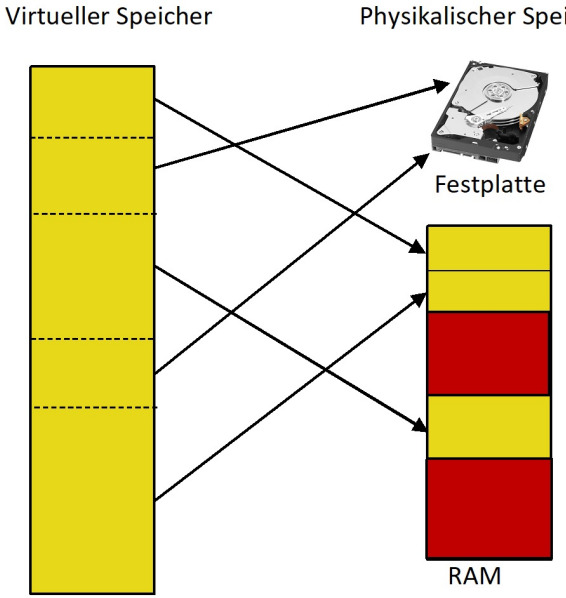
seL4

- In den 1990er Jahren entwickelt.
- Basiert auf dem L4 Microkernel.
- Stellt minimale Anzahl an services für Applikationen bereit.

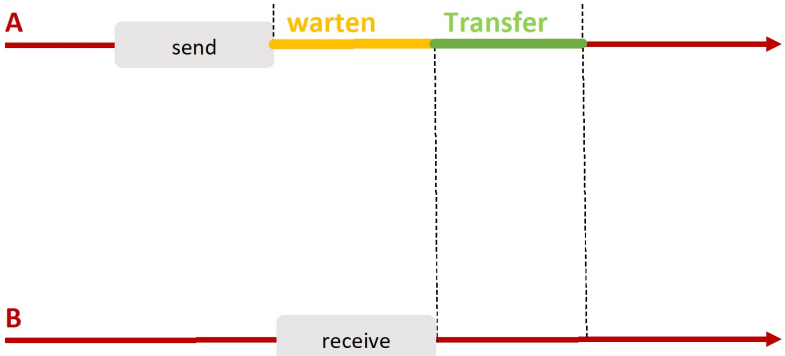

A
send
warten
Transfer
B
receive
IPC



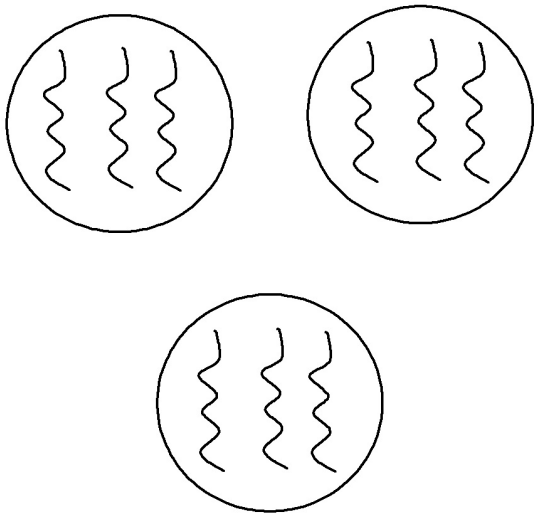
IPC



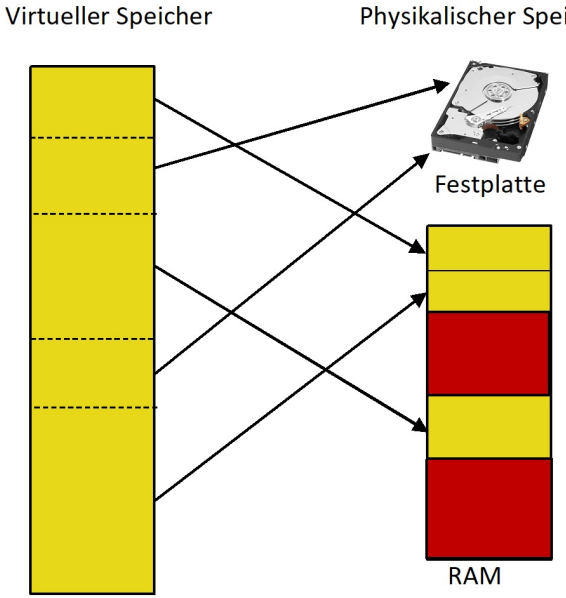
Speicherverwaltung



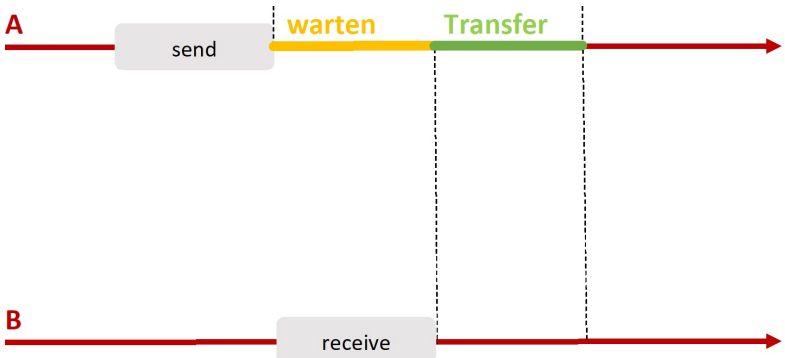
IPC



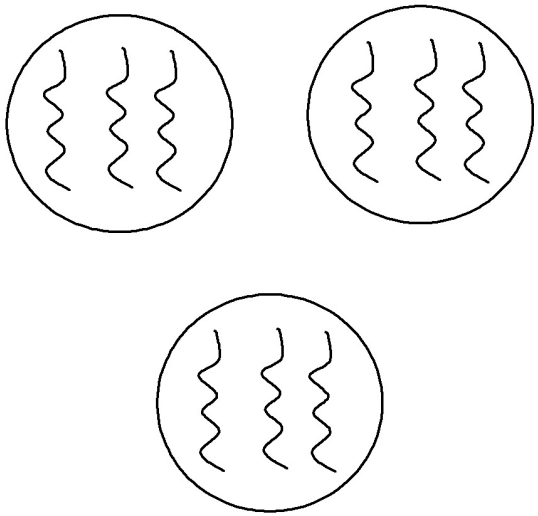
Threads



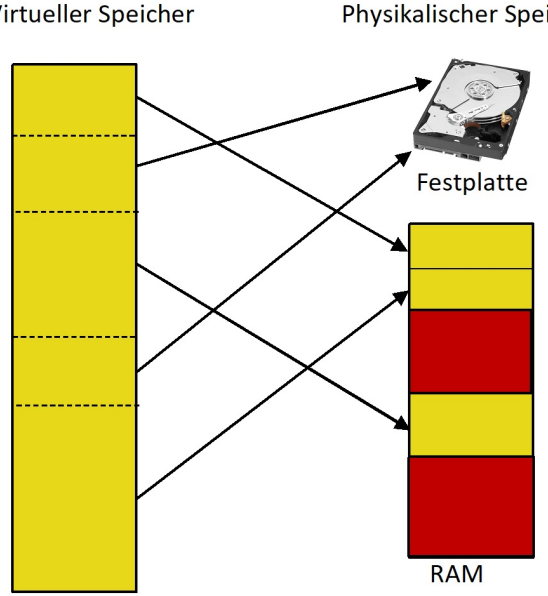
Speicherverwaltung



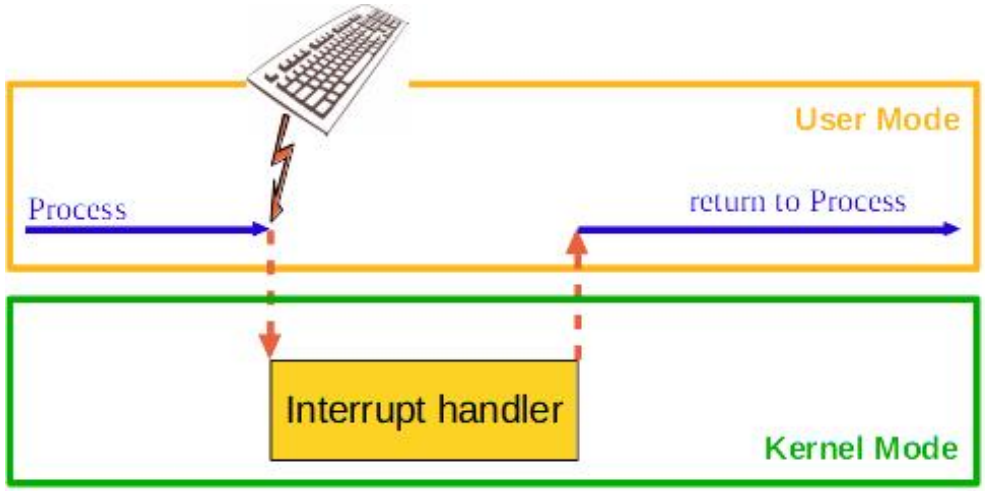
IPC



Threads



Speicherverwaltung

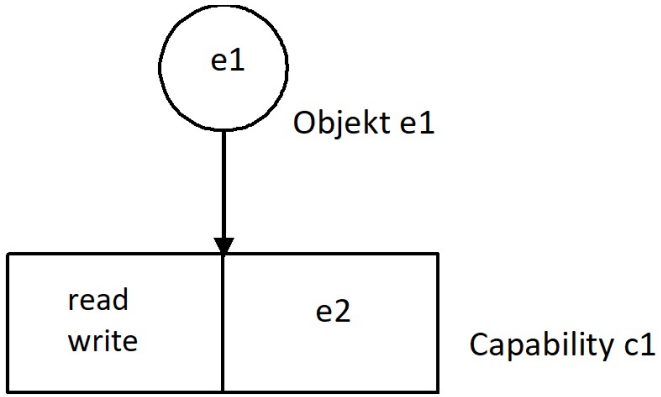


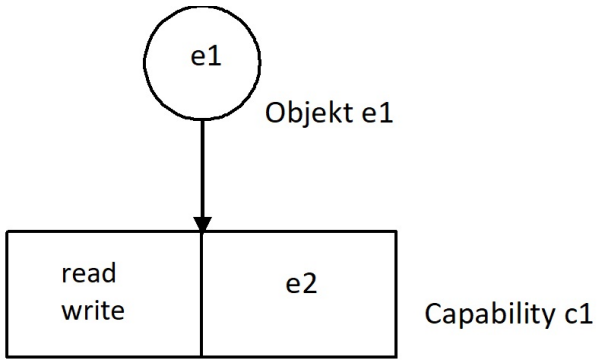
Interrupts

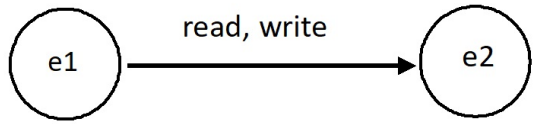
- In den 1990er Jahren entwickelt.
- Basiert auf dem L4 Microkernel.
- Stellt minimale Anzahl an services für Applikationen bereit.
- **Objekte:** implementieren jeweils die Abstraktion eines Services.

- In den 1990er Jahren entwickelt.
- Basiert auf dem L4 Microkernel.
- Stellt minimale Anzahl an services für Applikationen bereit.
- **Objekte:** implementieren jeweils die Abstraktion eines Services.
- **Capabilities:** von Applikationen benötigt, um einen Service zu nutzen.

- In den 1990er Jahren entwickelt.
- Basiert auf dem L4 Microkernel.
- Stellt minimale Anzahl an services für Applikationen bereit.
- **Objekte:** implementieren jeweils die Abstraktion eines Services.
- **Capabilities:** von Applikationen benötigt, um einen Service zu nutzen.
- Die Rechte `Read`, `Write`, `Grant` und `Create` können in den Capabilities enthalten sein.


e1
Objekt e1
read
write
e2
Capability c1

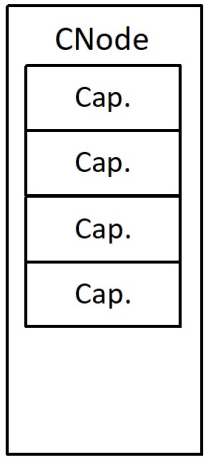

e1
Objekt e1
read
write
e2
Capability c1


e1
read, write
e2

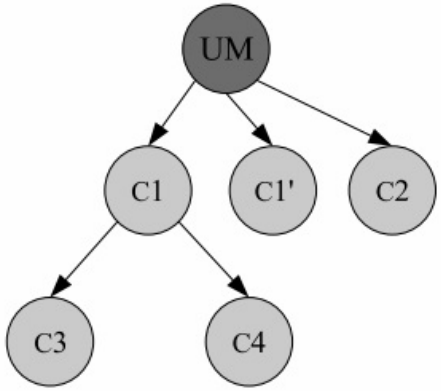
Kernel Objekte

- CNodes
- IPC Endpoints
- TCB
- Virtual Memory
- Interrupt Objects
- Untyped Memory

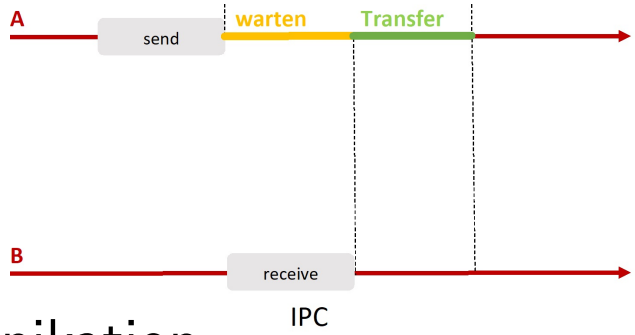
CNodes



- Lagern die Capabilities
- Erhalten feste Zahl an Slots
- Kernel konstruiert einen CDT (Capability Derivation Tree) zur Dokumentation der erstellten Capabilities und ihrer Verbindungen.
- Mehrere CNodes bilden eine CSpace.

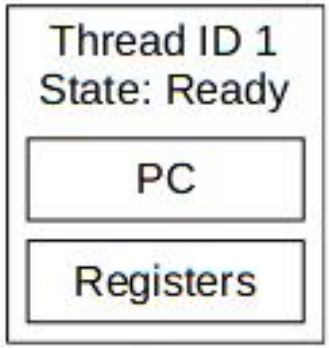


IPC Endpoints



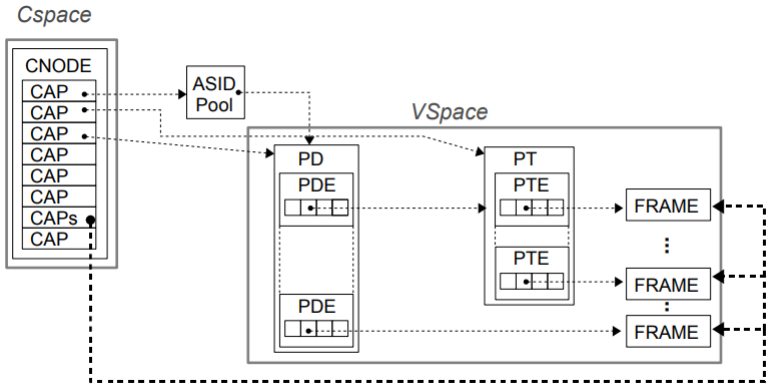
- Notwendig für die Interprocesskommunikation
- Unterscheidung: synchrone (SEP) und asynchrone (AEP) endpoints
- Unterteilung der Threads in security domains
- Kommunikation zwischen Threads aus verschiedenen security domains: nur über AEP
- Generelle Einschränkung von Capabilities auf Endpoints zu `read` oder `write` only ist möglich.

TCB



- *Thread control block*
- Repräsentiert einen Thread
- Immer verknüpft mit einer CSpace und einer VSpace.

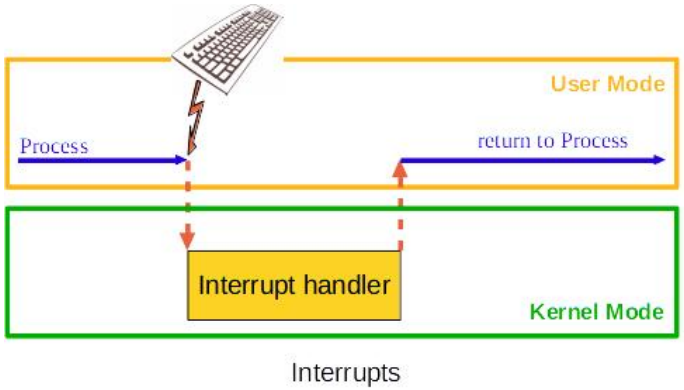
Virtual Memory



- VSpache (Virtual adress space)
  Verwaltung von virtuellem Speicher.
- ASID Table:
  - Globale Tabelle, feste Größe
  - Beim booten erstellt.
  - Dient zur Zuordnung von Mapping zu Adressräumen.
- PageDirectory (PD)
  - Oberes Level der 2-Level Page Tablestruktur.
  - Enthält PDEs (page directory entries)

- PageTable (PT)
  - 2. Level der 2-Level Page Tablestruktur.
  - Enthält PTEs (page table entries)
  - Ein PTE kann einen Pointer auf eine `Page` enthalten.
- Page
  - Abschnitt von physikalischem Speicher
  - Implementiert eine Seite im virtellen Speicher der VSpace.

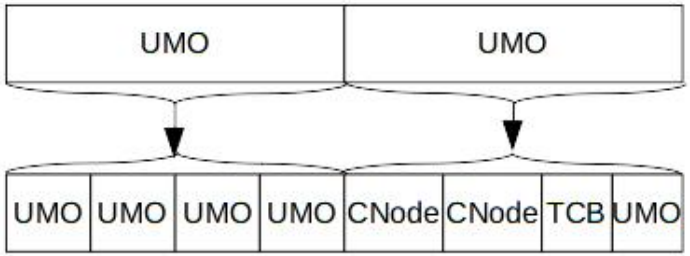
Interrupt Objects



Interrupts

- Geben Applikationen die Möglichkeit Unterbrechungen von Hardwarekomponenten zu erhalten und zu bestätigen.
- Eine InterruptHandler Capability erlaubt das Management einer Interruptursache.
- Eine InterruptController Capability erlaubt das Erstellen einer InterruptHandler Capability.

Untyped Memory Objects (UMO



- Kapseln eine Region physikalischen Speichers ein.
- Können in eine Gruppe kleiner UMOs geteilt werden.
- Können mit `Retype()` in andere Objekttypen umgewandelt werden.

Memory Allocation Model

- Speicher für Kernelobjekte wird nicht dynamisch erzeugt.
- Feste Speicherregionen zur Selbstverwaltung.
- Zur Erzeugung neuer Objekte: Capabilities auf UMOs nötig
- Initial User Thread einer Applikation erhält verfügbaren Speicher durch Capabilities auf UMOs.
- Durch fest zugewiesenen Speicher und Selbstverwaltung des zugewiesenen Speichers $\Rightarrow$ Isolation des physikalischen Speichers zwischen Applikationen.

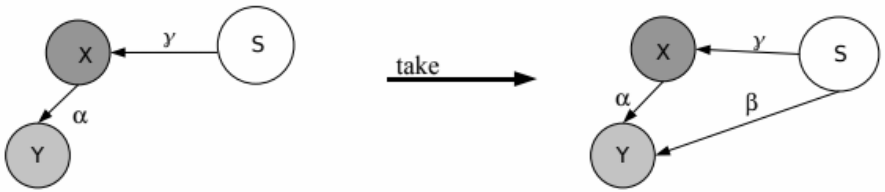
The Take-Grant Model

Das klassische Modell

- Subjekte und Objekte = Knoten
- Berechtigungen = Pfeile
- Gerichteter Graph = System
- Regeln zur Veränderung des Graphen = Verschiedene Systemoperationen zur Verteilung der Berechtigunen.
- Standardregeln: *take*, *grant*, *create*, *remove*

Take Regel

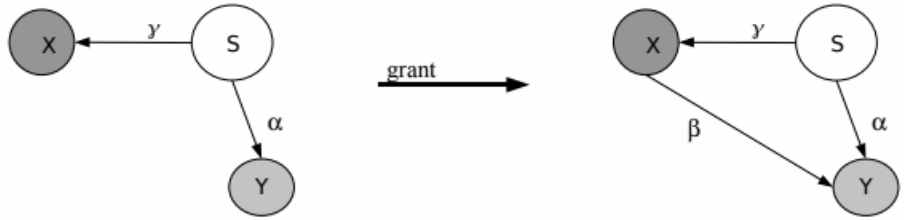
- S,X,Y = 3 verschiedene Knoten im Graph
- α = Pfeil von X nach Y
- γ = Pfeil von S nach X
- t"bezeichnet das *take* Recht
- t $\in \gamma$



Take fügt eine Kante von S nach Y, mit dem Lable $\beta \subseteq \alpha$ hinzu.

Grant Regel

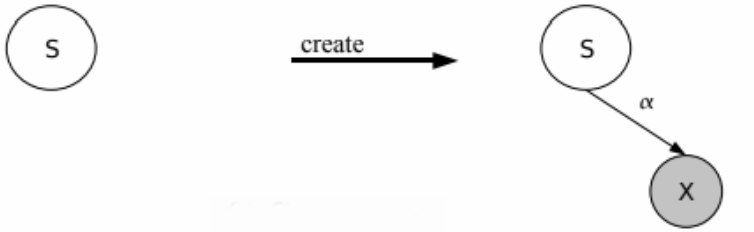
- S,X,Y = 3 verschiedene Knoten im Graph
- α = Pfeil von S nach Y
- γ = Pfeil von S nach X
- "g"bezeichnet das *grant* Recht
- g $\in \gamma$



Grant fügt eine Kante von X nach Y, mit dem Lable $\beta \subseteq \alpha$ hinzu.

Create Regel

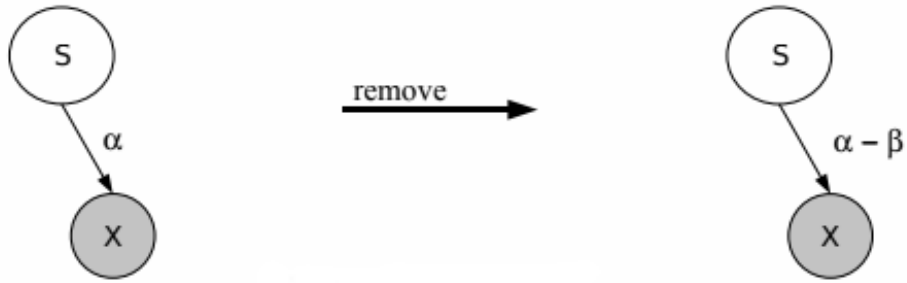
S = Knoten im Graph



Create fügt einen neuen Knoten und eine Kante von S nach X, mit dem Lable α hinzu.

Remove Regel

- S,X = verschiedene Knoten im Graph
- α = Pfeil von S nach X



Remove entfernt β Lable von α oder den kompletten Pfeil, falls $\alpha - \beta = \{\}$

Take-Grant für den seL4 verändert

Modifikationen:

- *Create:*
  - Objekt das *create* ausführt (e1), benötigt create Rechte in einer seiner Capabilities.
  - Neuer Knoten wird erstellt.
  - Objekt e1 kann anderem Objekt e2 alle verfügbaren Rechte auf das neue Objekt geben, wenn e1 *grant* Rechte auf e2 besitzt.
- *Remove:*
  Entfernt nicht mehr Teile eines Lables, sondern die komplette Capability.
- *Revoke:*
  Entfernt im CDT alle Capabilities ab einer angegebenen.
- *Grant* wurde nicht verändert und *take* wurde entfernt.

- Objekte u. Subjekte = *entities*
- Kopieren von Capabilities:
  - `mint`: kopierte Capability = Kindknoten der originalen im CDT (weniger oder gleich viel Rechte wie das Original)
  - `imitate`: kopierte Capability = Geschwisterknoten der originalen im CDT (gleich viel Rechte wie das Original)
- Isolation durch Subsysteme (durch *grant* verbundene entities)

Noninterference

- Kontrolliert den Informationsfluss in einem System
- Objekte aus unterschiedlichen Sicherheitslevels dürfen sich nicht gegenseitig beeinflussen.
- Variablen im Model = L (low security) oder H (high security) Variablen.
- Geoffrey Smith in "Principles of Secure Information Flow Analysis": "**Program c satisfies noninterference if, for any memories μ and ν that agree on L variables, the memories produced by running c on μ and on ν also agree on L variables (provided that both runs terminate successfully).**"
- Notation für die Noninterference policy: $\rightsquigarrow$
- L$\rightsquigarrow$H $\equiv$ Erlaubter Informationsfluss: von L nach H
- μ und ν stimmen auf L Variablen überein, wenn sie eine Äquivalenzrelation $\mu \overset{\mathsf{L}}{\sim} \nu$ erfüllen.

Formalisierung des Take-Grant Models

- Entities werden Indentifieziert durch Speicheradresse (modelliert durch eine natürliche Zahl):
  type_synonym `entity_id = nat`
- *Zugriffsrechte*
  datatype `rights = Read | Write | Grant | Create`
- *Capabilities*
  record `cap = entity :: entity_id`
  `rights :: rights set`
- *Entities*
  record `entity = caps :: cap set`
- *Status (state) des Systems*
  record `state = heap :: entity_id ⇒ entity`
  `next_id :: entity_id`

- *Verschiedene Systemoperartionen*

```
datatype  sysOps  =  SysNoOp entity_id
                  |  SysRead entity_id cap
                  |  SysWrite entity_id cap
                  |  SysCreate entity_id cap cap
                  |  SysGrant entity_id cap cap rights set
                  |  SysRemove entity_id cap cap
                  |  SysRevoke entity_id cap
```

- *legal* prüft ob eine Systemoperation legal ist.

```
legal ::  "sysOPs ⇒ state ⇒ bool" where

    "legal  (SysNoOp e) s           =  isEntityOf s e"
  | "legal  (SysCreate e c₁ c₂) s   =  (isEntityOf s e ∧ c₁, c₂ ⊆ caps_of s e ∧
                                       Grant ∈ rights c₂ ∧ Create ∈ rights c₂)"
  | "legal  (SysRead e c) s         =  (isEntityOf s e ∧ c ∈ caps_of s e ∧ Read
                                       ∈ rights c)"
  | "legal  (SysWrite e c) s        =  (isEntityOf s e ∧ c ∈ caps_of s e ∧ Write
                                       ∈ rights c)"
  | "legal  (SysGrant e c₁ c₂ r) s  =  (isEntityOf s e ∧ isEntityOf s (entity c₁)
                                       ∧ c₁,c₂ ⊆ caps_of s e ∧ Grant ∈ rights c₁)"
  | "legal  (SysRemove e c₁ c₂) s   =  (isEntityOf s e ∧ c₁ ∈ caps_of s e)"
  | "legal  (SysRevoke e c) s       =  isEntityOf s e ∧ c ∈ caps_of s e"
```