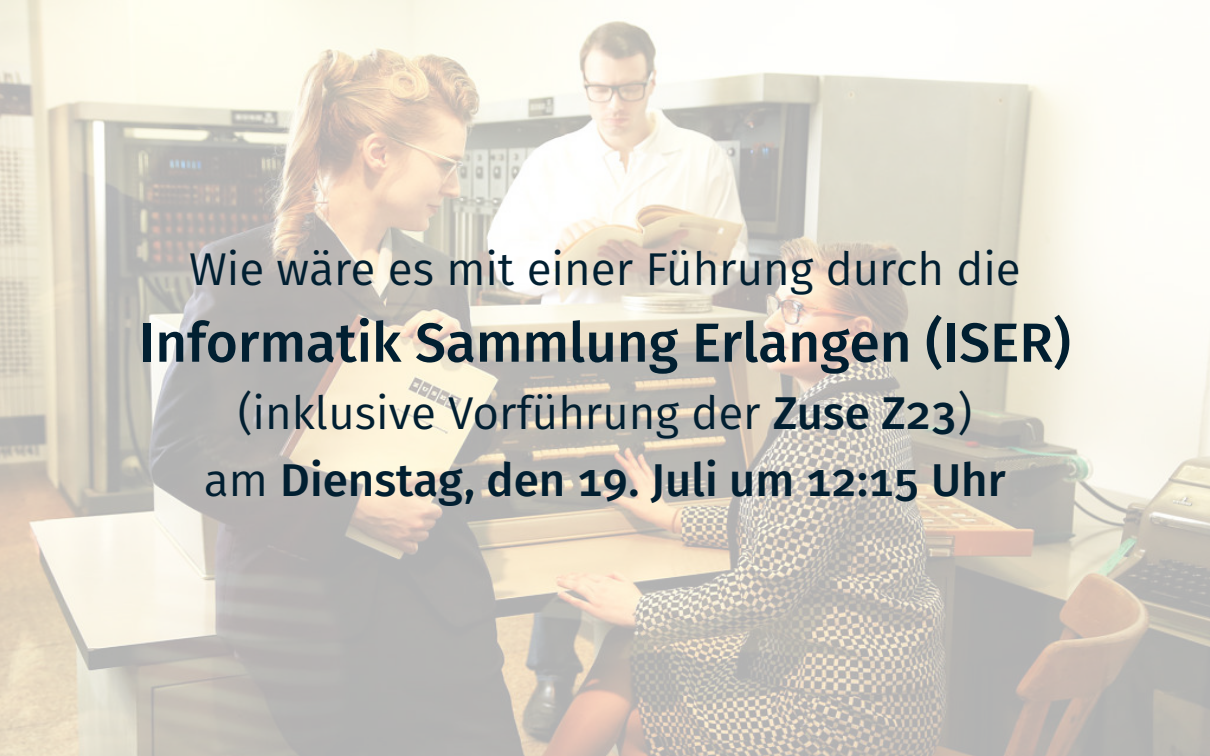
Prüfungsterminfindung via Waffel:

waffel.cs.fau.de/signup?course=452

Sollte kein Termin passen, einfach Wunschtermin per Mail anfragen!


Wie wäre es mit einer Führung durch die
Informatik Sammlung Erlangen (ISER)
(inklusive Vorführung der Zuse Z23)
am Dienstag, den 19. Juli um 12:15 Uhr

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; ***premature optimization is the root of all evil*** *(or at least most of it) in programming.*

Donald Knuth in *The Art of Computer Programming* (S. 671)

Übung zu Betriebssystemtechnik

Aufgabe 5: Prozess- & dynamische Speicherverwaltung

28. Juni 2022

Bernhard Heinloth, Phillip Raffeck & Dustin Nguyen

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Ein Prozess soll die Möglichkeit haben, sich dynamisch zu duplizieren, um dadurch einen neuen Prozess zu erstellen

Prozessverwaltung

Neue Systemaufrufe für Prozessverwaltung

- `int fork()`
  erstellt einen neuen [Kind-]Prozess in einem eigenen Adressraum, welcher ein **Duplikat des aufrufenden [Eltern-]Prozesses** zum Ausführungszeitpunkt ist.

Disclaimer

Der `fork()` Systemaufruf ist eine historische „Altlast“, welche wir in der Übung aus rein **pragmatischen Gründen** verwenden wollen.

Disclaimer

Der `fork()` Systemaufruf ist eine historische „Altlast“, welche wir in der Übung aus rein **pragmatischen Gründen** verwenden wollen.

Eine flexible Alternative ist z.B. `clone()` – jedoch für diese Aufgabe deutlich zu umfangreich.

Lesetipp: A fork() in the road *(HotOS ’19)*

Neue Systemaufrufe für Prozessverwaltung

- `int fork()`
  erstellt einen neuen [Kind-]Prozess in einem eigenen Adressraum, welcher ein **Duplikat des aufrufenden [Eltern-]Prozesses** zum Ausführungszeitpunkt ist.

Neue Systemaufrufe für Prozessverwaltung

- `int fork()`
  erstellt einen neuen [Kind-]Prozess in einem eigenen Adressraum, welcher ein **Duplikat des aufrufenden [Eltern-]Prozesses** zum Ausführungszeitpunkt ist. Der Rückgabewert des Aufrufs ist im Elternprozess die Prozess-ID des Kindes, im Kindprozess hingegen `0`.

Neue Systemaufrufe für Prozessverwaltung

- `int fork()`
  erstellt einen neuen [Kind-]Prozess in einem eigenen Adressraum, welcher ein **Duplikat des aufrufenden [Eltern-]Prozesses** zum Ausführungszeitpunkt ist. Der Rückgabewert des Aufrufs ist im Elternprozess die Prozess-ID des Kindes, im Kindprozess hingegen `0`.
- `int getpid()`
  gibt die **Prozess-ID des jeweiligen Aufrufers** zurück.

Neue Systemaufrufe für Prozessverwaltung

- `int fork()`
  erstellt einen neuen [Kind-]Prozess in einem eigenen Adressraum, welcher ein **Duplikat des aufrufenden [Eltern-]Prozesses** zum Ausführungszeitpunkt ist. Der Rückgabewert des Aufrufs ist im Elternprozess die Prozess-ID des Kindes, im Kindprozess hingegen `0`.
- `int getpid()`
  gibt die **Prozess-ID des jeweiligen Aufrufers** zurück.
- `int getppid()`
  Gibt die **Prozess-ID des Elternprozesses** zurück.

Neue Systemaufrufe für Prozessverwaltung

- `int fork()`
  erstellt einen neuen [Kind-]Prozess in einem eigenen Adressraum, welcher ein **Duplikat des aufrufenden [Eltern-]Prozesses** zum Ausführungszeitpunkt ist. Der Rückgabewert des Aufrufs ist im Elternprozess die Prozess-ID des Kindes, im Kindprozess hingegen `0`.
- `int getpid()`
  gibt die **Prozess-ID des jeweiligen Aufrufers** zurück.
- `int getppid()`
  Gibt die **Prozess-ID des Elternprozesses** zurück. Bei Prozessen, die nicht durch `fork()` erstellt wurden, ist dies `0`.

Neue Systemaufrufe für Prozessverwaltung

- `int fork()`
  erstellt einen neuen [Kind-]Prozess in einem eigenen Adressraum, welcher ein **Duplikat des aufrufenden [Eltern-]Prozesses** zum Ausführungszeitpunkt ist. Der Rückgabewert des Aufrufs ist im Elternprozess die Prozess-ID des Kindes, im Kindprozess hingegen `0`.
- `int getpid()`
  gibt die **Prozess-ID des jeweiligen Aufrufers** zurück.
- `int getppid()`
  Gibt die **Prozess-ID des Elternprozesses** zurück. Bei Prozessen, die nicht durch `fork()` erstellt wurden, ist dies `0`.

Jeweils sowohl interruptbasiert als auch als schneller Systemaufruf!

Beispiel

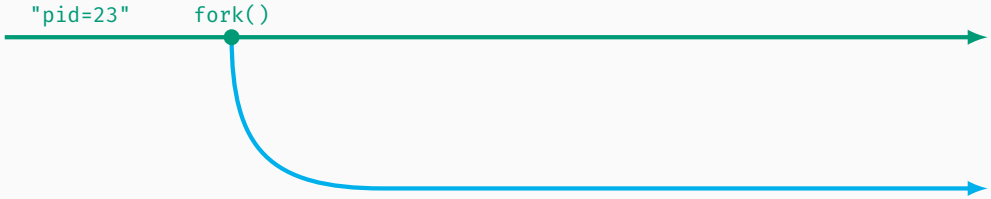
```
cerr << "pid=" << getpid() << endl;
```

Beispiel

```
cerr << "pid=" << getpid() << endl;


int ret = fork();
cerr << "ret=" << ret << endl;
```

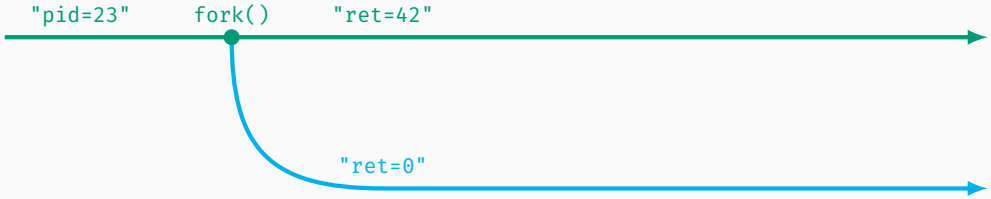
Beispiel

```
cerr << "pid=" << getpid() << endl;


int ret = fork();
cerr << "ret=" << ret << endl;
```



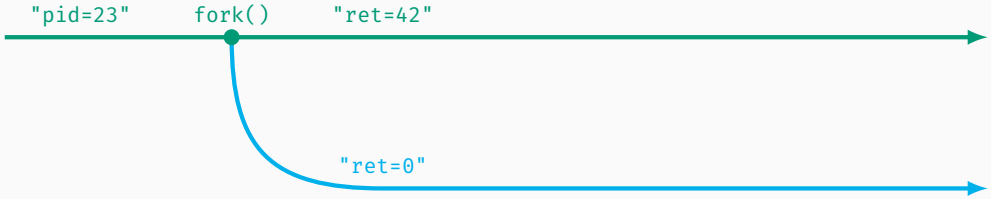
Beispiel

```
cerr << "pid=" << getpid() << endl;


int ret = fork();
cerr << "ret=" << ret << endl;
```



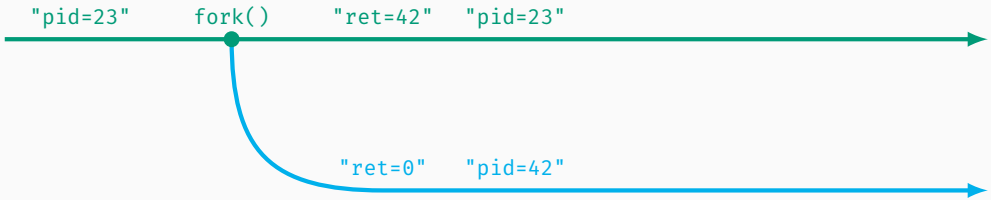
Beispiel

```
cerr << "pid=" << getpid() << endl;


int ret = fork();
cerr << "ret=" << ret << endl;
cerr << "pid=" << getpid() << endl;
```



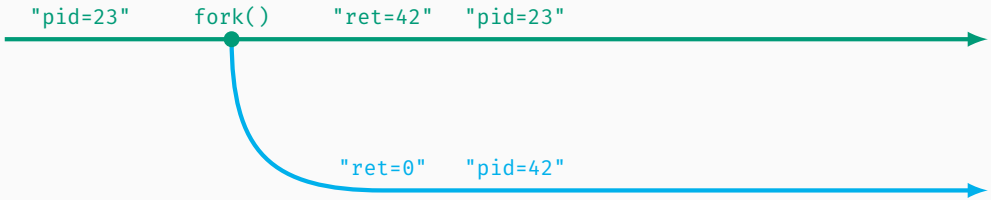
Beispiel

```
cerr << "pid=" << getpid() << endl;


int ret = fork();
cerr << "ret=" << ret << endl;
cerr << "pid=" << getpid() << endl;
```



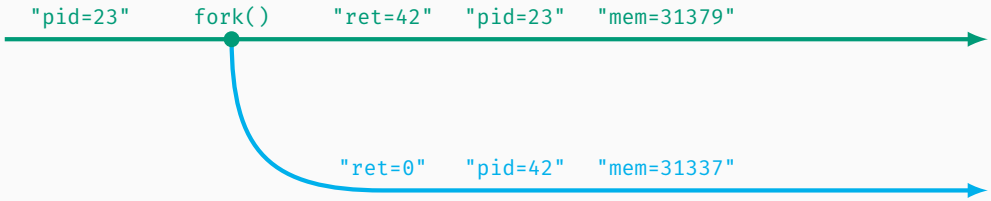
Beispiel

```
cerr << "pid=" << getpid() << endl;

volatile long mem = 1337;
int ret = fork();
cerr << "ret=" << ret << endl;
cerr << "pid=" << getpid() << endl;
mem += 30000 + ret;
cerr << "mem=" << mem << endl;
```



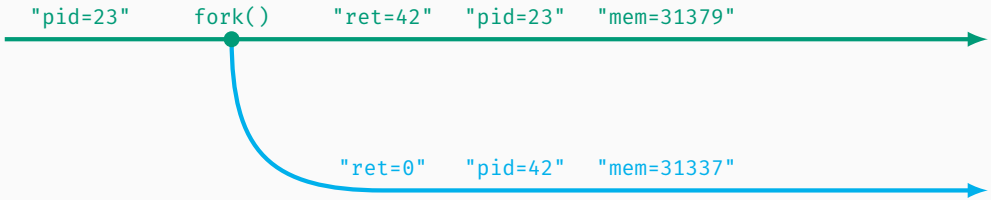
Beispiel

```
cerr << "pid=" << getpid() << endl;

volatile long mem = 1337;
int ret = fork();
cerr << "ret=" << ret << endl;
cerr << "pid=" << getpid() << endl;
mem += 30000 + ret;
cerr << "mem=" << mem << endl;
```



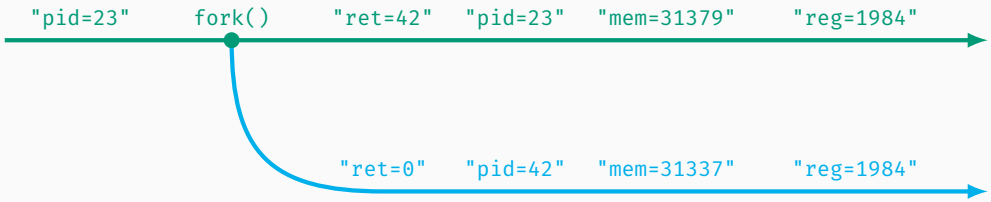
Beispiel

```
cerr << "pid=" << getpid() << endl;
register long reg asm("rbx") = 1984;
volatile long mem = 1337;
int ret = fork();
cerr << "ret=" << ret << endl;
cerr << "pid=" << getpid() << endl;
mem += 30000 + ret;
cerr << "mem=" << mem << endl;
cerr << "reg=" << reg << endl;
```



Beispiel

```
cerr << "pid=" << getpid() << endl;
register long reg asm("rbx") = 1984;
volatile long mem = 1337;
int ret = fork();
cerr << "ret=" << ret << endl;
cerr << "pid=" << getpid() << endl;
mem += 30000 + ret;
cerr << "mem=" << mem << endl;
cerr << "reg=" << reg << endl;
```



Zu duplizierender Zustand eines Prozesses

Zu duplizierender Zustand eines Prozesses

- (User-)Stack

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register
  - *callee-save*/*non-scratch* Register müssen von uns gesichert werden

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register
  - *callee-save*/*non-scratch* Register müssen von uns gesichert werden

im Kernel

beim Einsprung in die Systemaufruf-behandlung (im Assemblerteil)

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register
  - *callee-save*/*non-scratch* Register müssen von uns gesichert werden

im Kernel	im Userspace
beim Einsprung in die Systemaufrufbehandlung (im Assemblerteil)	zu Beginn im Aufrufstumpf auf Stack pushen, bei Rückkehr wieder popen

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register
  - *callee-save*/*non-scratch* Register müssen von uns gesichert werden

im Kernel	im Userspace
beim Einsprung in die Systemaufrufbehandlung (im Assemblerteil)	zu Beginn im Aufrufstumpf auf Stack pushen, bei Rückkehr wieder popen
+ (theoretisch) sauberer Ansatz	– Abhängigkeit im Userspace

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register
  - der Übersetzer kümmert sich bereits um *calle**r**-save*/*scratch* Register
  - *calle**e**-save*/*non-scratch* Register müssen von uns gesichert werden

im Kernel	**im Userspace**
beim Einsprung in die Systemaufrufbehandlung (im Assemblerteil)	zu Beginn im Aufrufstumpf auf Stack `push`en, bei Rückkehr wieder `pop`en
+ (theoretisch) sauberer Ansatz	– Abhängigkeit im Userspace
– komplex[er]e Implementierung	+ sehr einfache Implementierung
– ggf. permanenter Overhead	+ kein unnötiger Overhead

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register
  - *callee-save*/*non-scratch* Register müssen von uns gesichert werden

im Kernel	im Userspace
beim Einsprung in die Systemaufrufbehandlung (im Assemblerteil)	zu Beginn im Aufrufstumpf auf Stack `push`en, bei Rückkehr wieder `pop`en
+ (theoretisch) sauberer Ansatz	– Abhängigkeit im Userspace
– komplex[er]e Implementierung	+ sehr einfache Implementierung
– ggf. permanenter Overhead	+ kein unnötiger Overhead
→ Ansatz von Linux usw.	→ empfohlene Variante für **StuBSmi**

Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register → in **StuBSmI** auf (User-)Stack gesichert
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register
  - *callee-save*/*non-scratch* Register müssen von uns gesichert werden

im Kernel	im Userspace
beim Einsprung in die Systemaufrufbehandlung (im Assemblerteil)	zu Beginn im Aufrufstumpf auf Stack `push`en, bei Rückkehr wieder `pop`en
+ (theoretisch) sauberer Ansatz	– Abhängigkeit im Userspace
– komplex[er]e Implementierung	+ sehr einfache Implementierung
– ggf. permanenter Overhead	+ kein unnötiger Overhead
→ Ansatz von Linux usw.	→ empfohlene Variante für **StuBSmI**

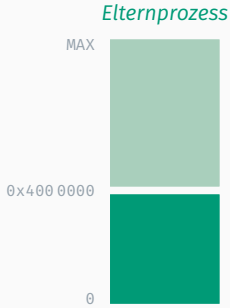
Zu duplizierender Zustand eines Prozesses

- (User-)Stack
- Daten (statisch und dynamisch allokiert)
- Code (insb. falls selbstverändernd)
- Register → in **StuBSmI** auf (User-)Stack gesichert
  - der Übersetzer kümmert sich bereits um *caller-save*/*scratch* Register
  - *callee-save*/*non-scratch* Register müssen von uns gesichert werden

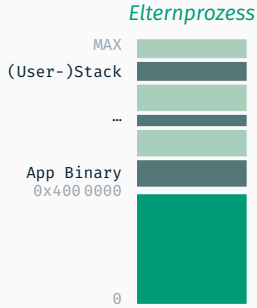
im Kernel	im Userspace
beim Einsprung in die Systemaufrufbehandlung (im Assemblerteil)	zu Beginn im Aufrufstumpf auf Stack pushen, bei Rückkehr wieder popen
+ (theoretisch) sauberer Ansatz	– Abhängigkeit im Userspace
– komplex[er]e Implementierung	+ sehr einfache Implementierung
– ggf. permanenter Overhead	+ kein unnötiger Overhead
→ Ansatz von Linux usw.	→ empfohlene Variante für **StuBSmI**

→ kompletten Userspace-Teil des virtuellen Speichers duplizieren

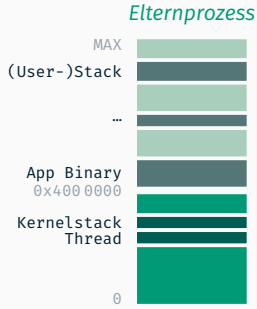
Duplizieren eines Prozesses



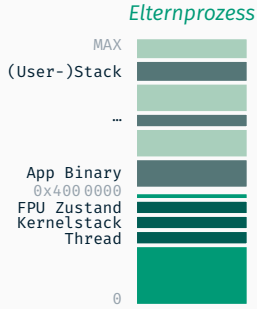
Duplizieren eines Prozesses



Duplizieren eines Prozesses

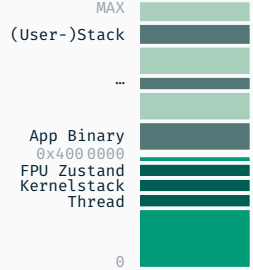


Duplizieren eines Prozesses



Duplizieren eines Prozesses

Elternprozess



Kindprozess

Kindprozess erstellen

Duplizieren eines Prozesses

Kindprozess erstellen

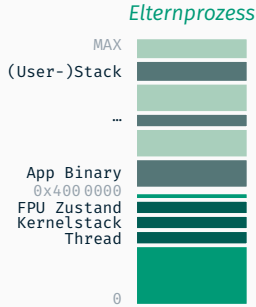
- neues `Thread`-Objekt

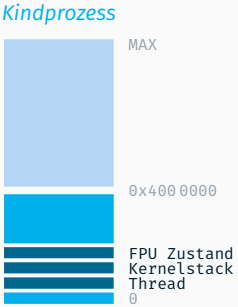
Duplizieren eines Prozesses

Kindprozess erstellen

- neues `Thread`-Objekt
- Kernelstack allokieren
- ggf. Speicher für FPU Zustand reservieren

Duplizieren eines Prozesses

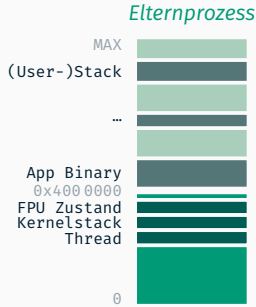


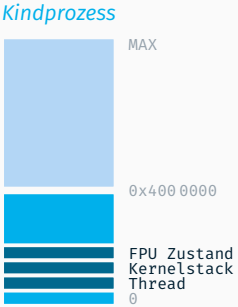


Kindprozess erstellen

- neues `Thread`-Objekt
- Kernelstack allokieren
- ggf. Speicher für FPU Zustand reservieren
- neuen virtuellen Adressraum erstellen

Duplizieren eines Prozesses





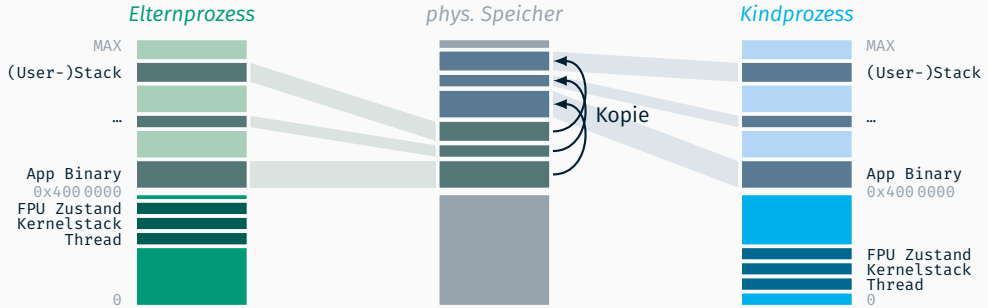
Virtuellen Adressraum vorbereiten

Duplizieren eines Prozesses

Virtuellen Adressraum vorbereiten

- Userspace-Seiten kopieren & einblenden

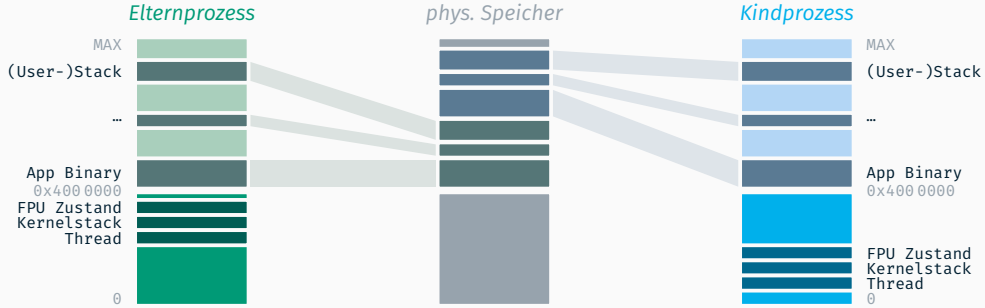
Duplizieren eines Prozesses



Virtuellen Adressraum vorbereiten

- Userspace-Seiten kopieren & einblenden
  - tiefe Kopie (*deep copy*), d.h. Inhalte der Seiten

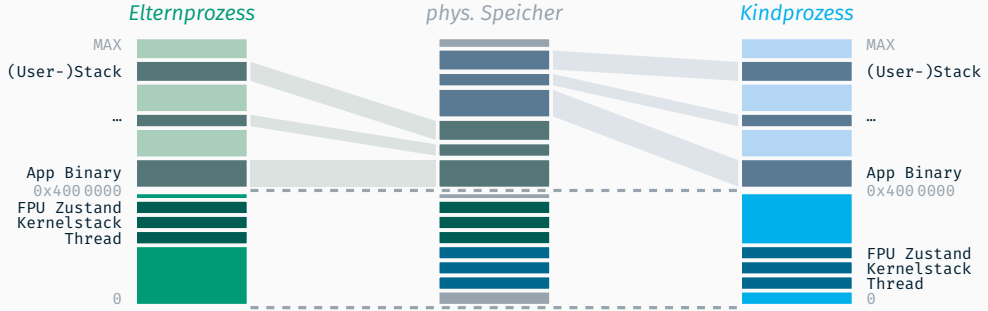
Duplizieren eines Prozesses



Virtuellen Adressraum vorbereiten

- Userspace-Seiten kopieren & einblenden
  - tiefe Kopie (*deep copy*), d.h. Inhalte der Seiten
  - Optimierung folgt in Aufgabe 7 (*copy-on-write*)

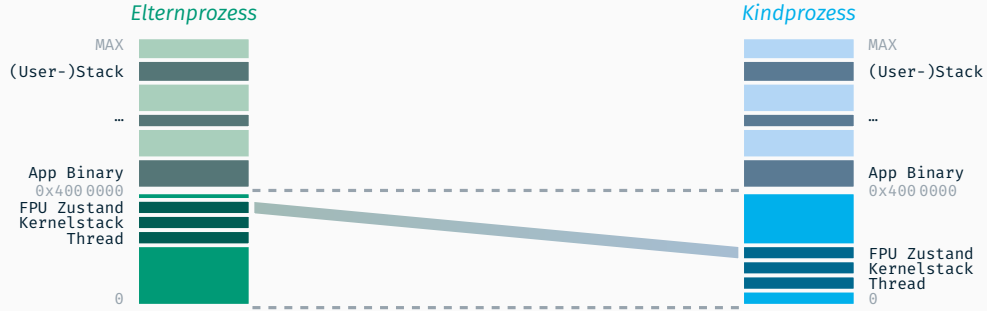
Duplizieren eines Prozesses



Virtuellen Adressraum vorbereiten

- Userspace-Seiten kopieren & einblenden
  - tiefe Kopie (*deep copy*), d.h. Inhalte der Seiten
  - Optimierung folgt in Aufgabe 7 (*copy-on-write*)
- Kernelspace ist identitätsabgebildet

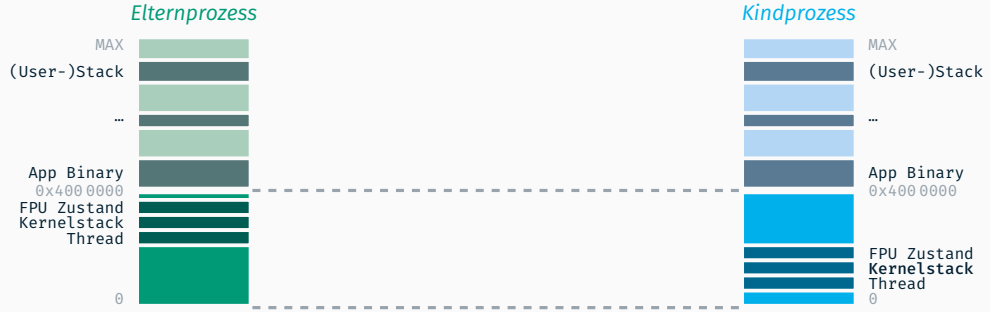
Duplizieren eines Prozesses



Virtuellen Adressraum vorbereiten

- Userspace-Seiten kopieren & einblenden
  - tiefe Kopie (*deep copy*), d.h. Inhalte der Seiten
  - Optimierung folgt in Aufgabe 7 (*copy-on-write*)
- Kernelspace ist identitätsabgebildet
  - ggf. FPU Zustand kopieren

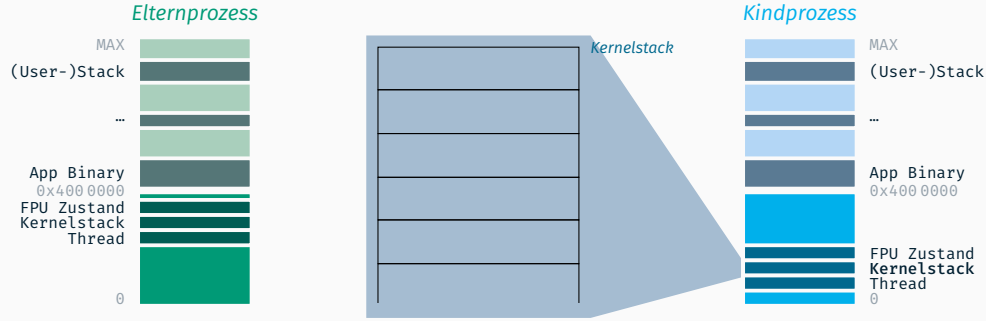
Duplizieren eines Prozesses



Virtuellen Adressraum vorbereiten

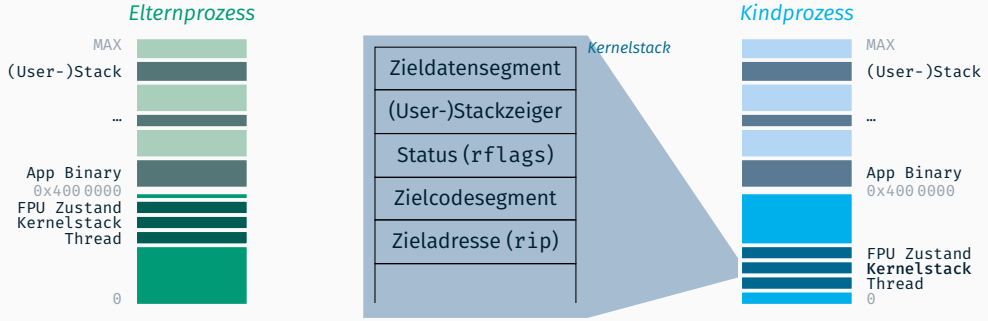
- Userspace-Seiten kopieren & einblenden
  - tiefe Kopie (*deep copy*), d.h. Inhalte der Seiten
  - Optimierung folgt in Aufgabe 7 (*copy-on-write*)
- Kernelspace ist identitätsabgebildet
  - ggf. FPU Zustand kopieren
  - Kernelstack des Kindes wird händisch (neu) aufgesetzt

Duplizieren eines Prozesses



Kernelstack des Kindes präperieren

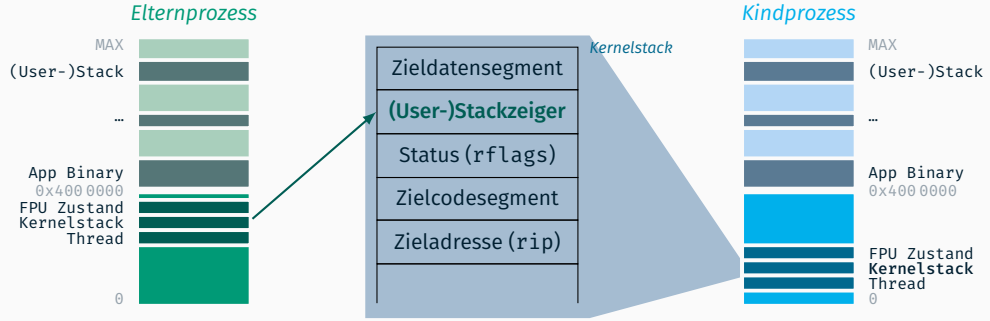
Duplizieren eines Prozesses



Kernelstack des Kindes präperieren

- analog zu initialem Wechseln nach Ring 3 (*siehe Aufgabe 1*)

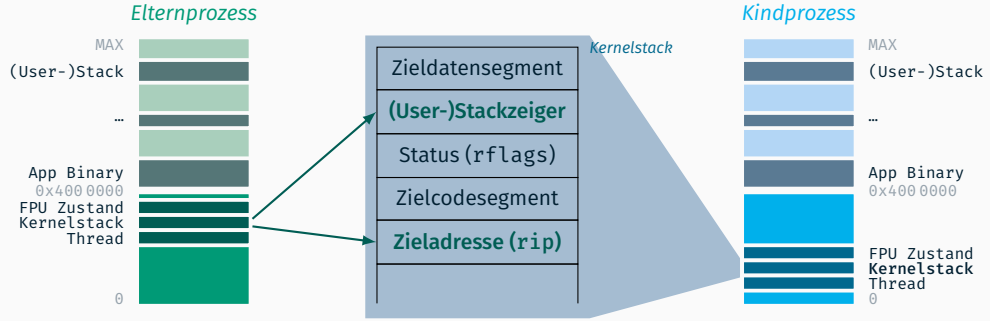
Duplizieren eines Prozesses



Kernelstack des Kindes präperieren

- analog zu initialem Wechseln nach Ring 3 (*siehe Aufgabe 1*)
- aber mit Userspace-Stackpointer wie Elternprozess

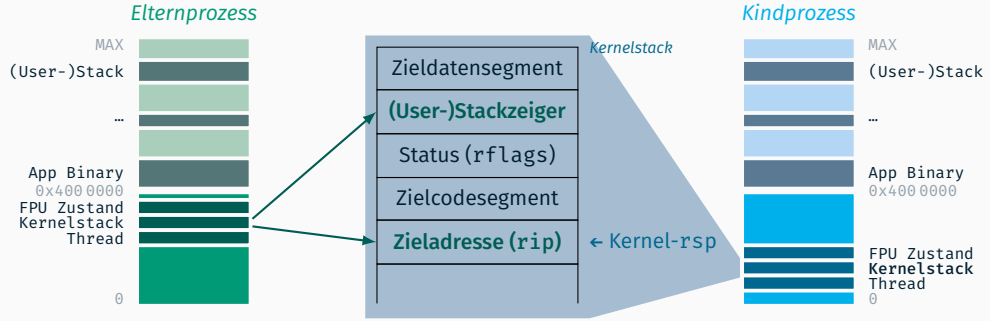
Duplizieren eines Prozesses



Kernelstack des Kindes präperieren

- analog zu initialem Wechseln nach Ring 3 (*siehe Aufgabe 1*)
- aber mit Userspace-Stackpointer wie Elternprozess
- ebenso auch Userspace-Rücksprungadresse

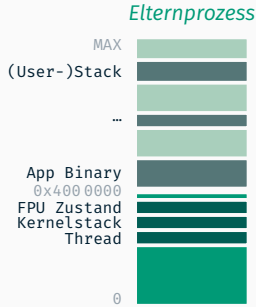
Duplizieren eines Prozesses

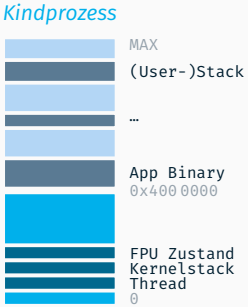


Kernelstack des Kindes präperieren

- analog zu initialem Wechseln nach Ring 3 (*siehe Aufgabe 1*)
- aber mit Userspace-Stackpointer wie Elternprozess
- ebenso auch Userspace-Rücksprungadresse
- Kernel-Stackpointer in Thread entsprechend setzen

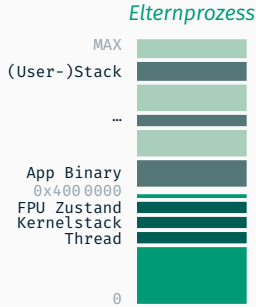
Duplizieren eines Prozesses

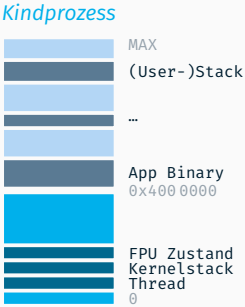




und Kindprozess als bereit markieren

Duplizieren eines Prozesses

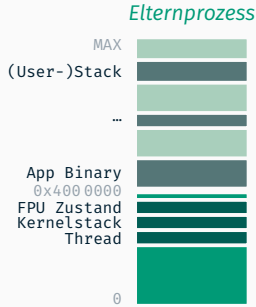


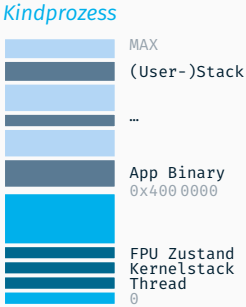


und Kindprozess als bereit markieren

- in Scheduler einlasten

Duplizieren eines Prozesses





und Kindprozess als bereit markieren

- in Scheduler einlasten
- im Elternprozess zurückkehren

Duplizieren eines Prozesses

und Kindprozess als bereit markieren

- in Scheduler einlasten
- im Elternprozess zurückkehren (mit ID des Kindprozesses)

Duplizieren eines Prozesses

und Kindprozess als bereit markieren

- in Scheduler einlasten
- im Elternprozess zurückkehren (mit ID des Kindprozesses)

Rückgabewert von `fork()` muss bei Kindprozess `0` sein

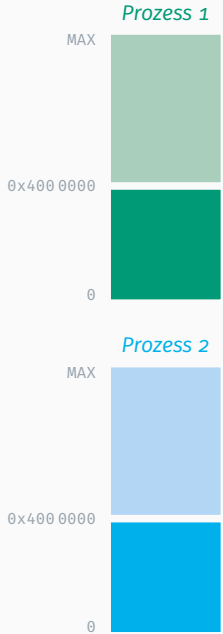
Duplizieren eines Prozesses

und Kindprozess als bereit markieren

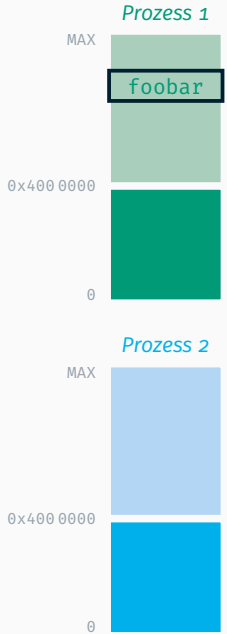
- in Scheduler einlasten
- im Elternprozess zurückkehren (mit ID des Kindprozesses)

Rückgabewert von `fork()` muss bei Kindprozess `0` sein
→ bei Einsprung in Ring 3 Register für Rückgabewert (`rax`) anpassen

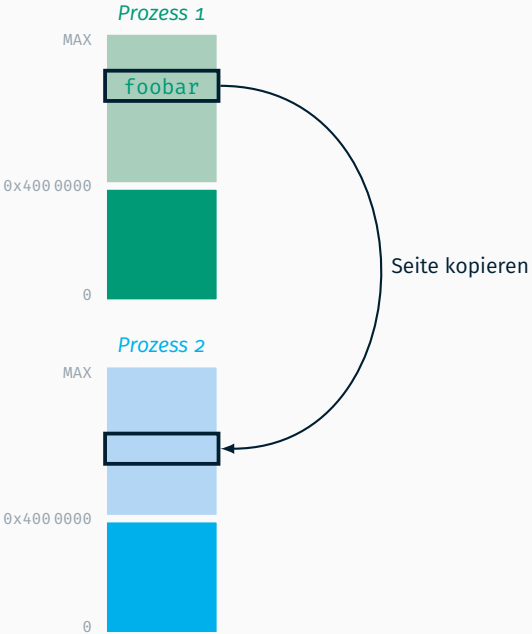
Adressraumübergreifendes Kopieren



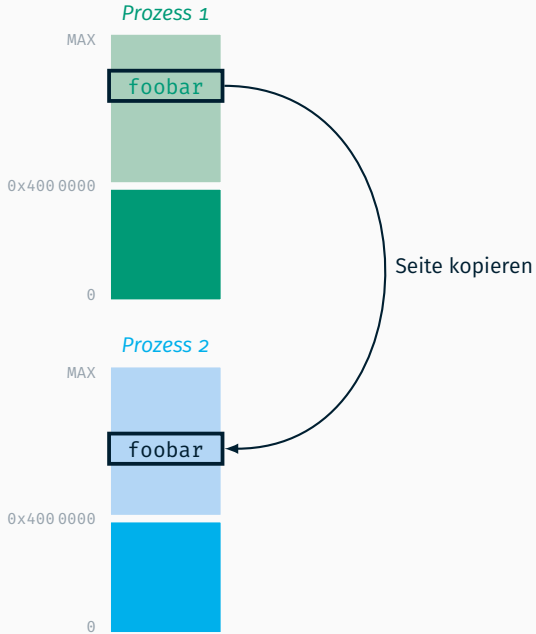
Adressraumübergreifendes Kopieren



Adressraumübergreifendes Kopieren

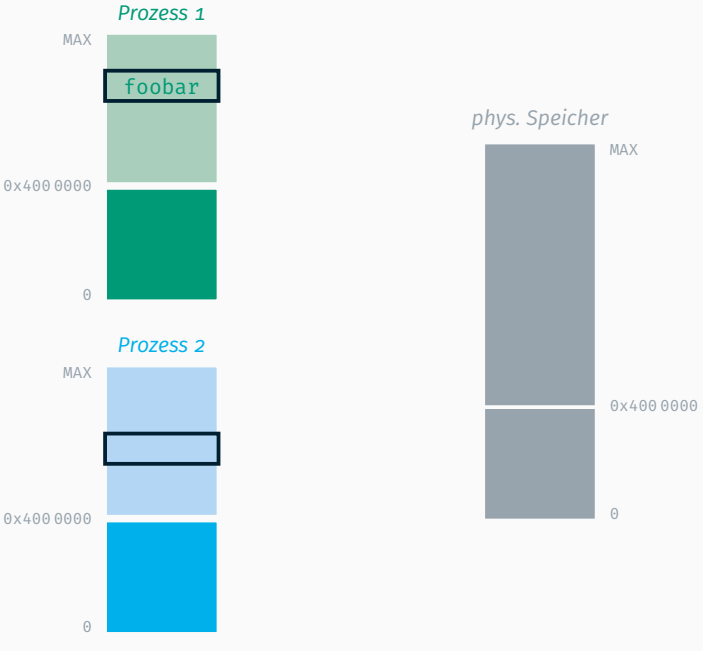


Adressraumübergreifendes Kopieren



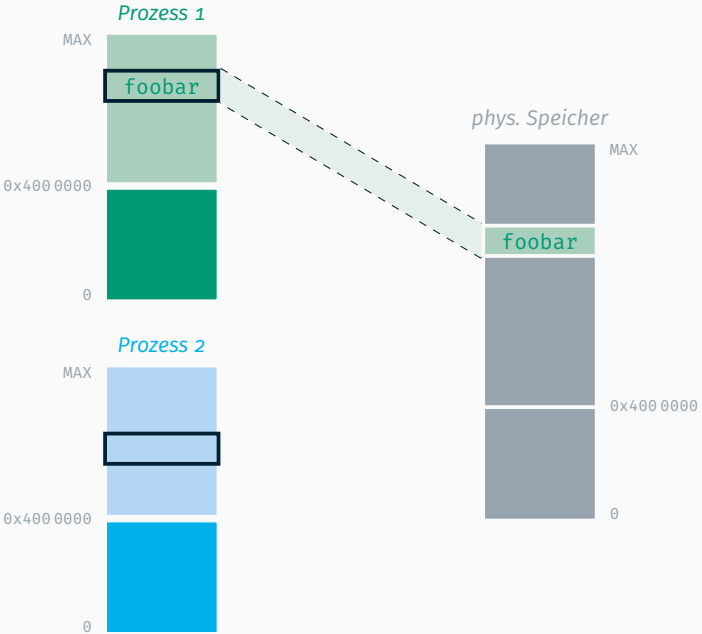
Adressraumübergreifendes Kopieren

shallow copy / Referenz



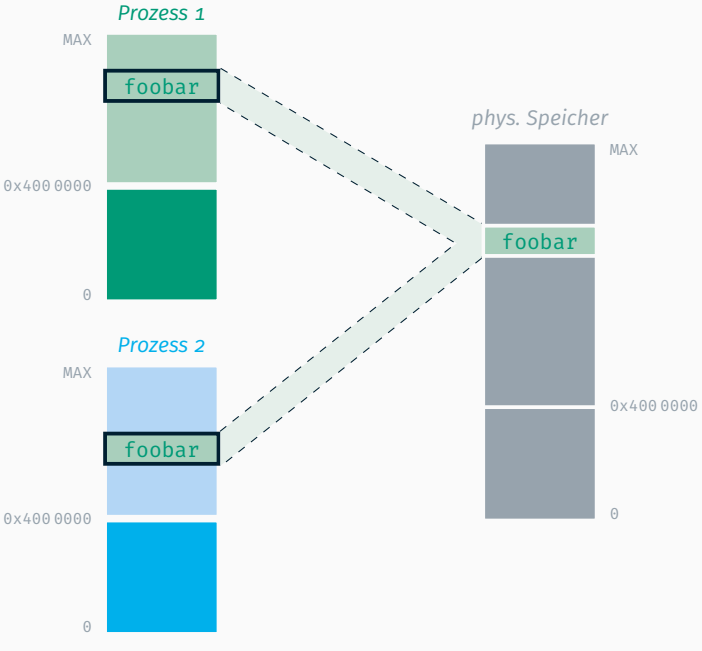
Adressraumübergreifendes Kopieren

shallow copy / Referenz



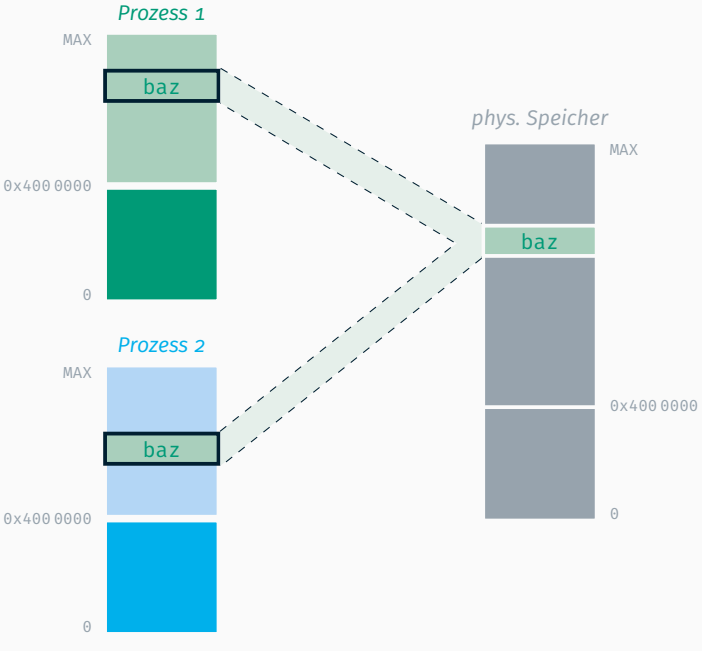
Adressraumübergreifendes Kopieren

shallow copy / Referenz



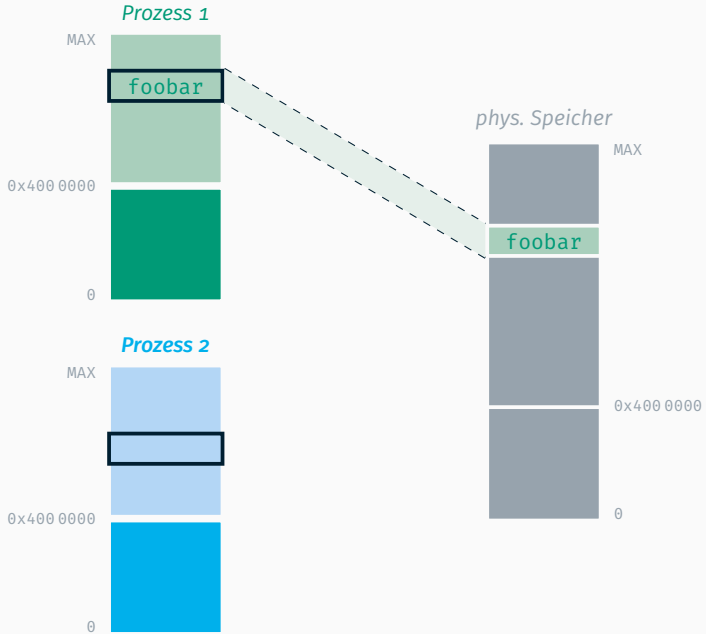
Adressraumübergreifendes Kopieren

shallow copy / Referenz



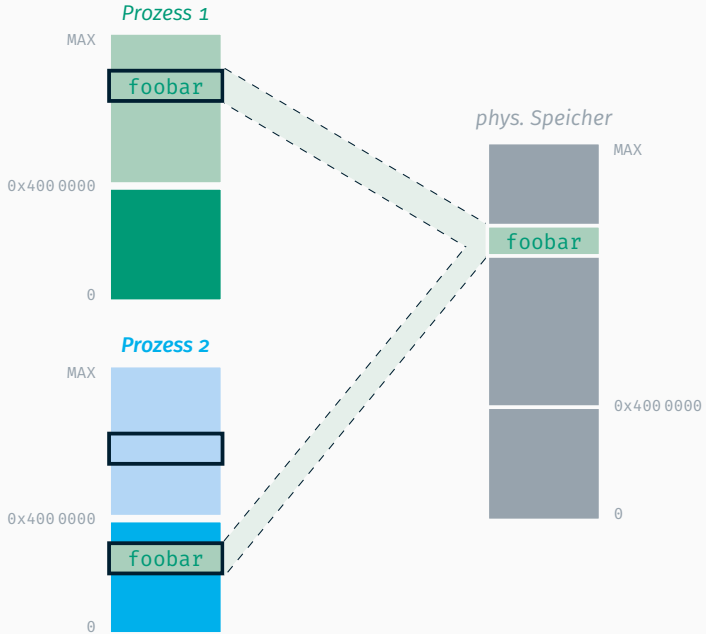
Adressraumübergreifendes Kopieren

deep copy



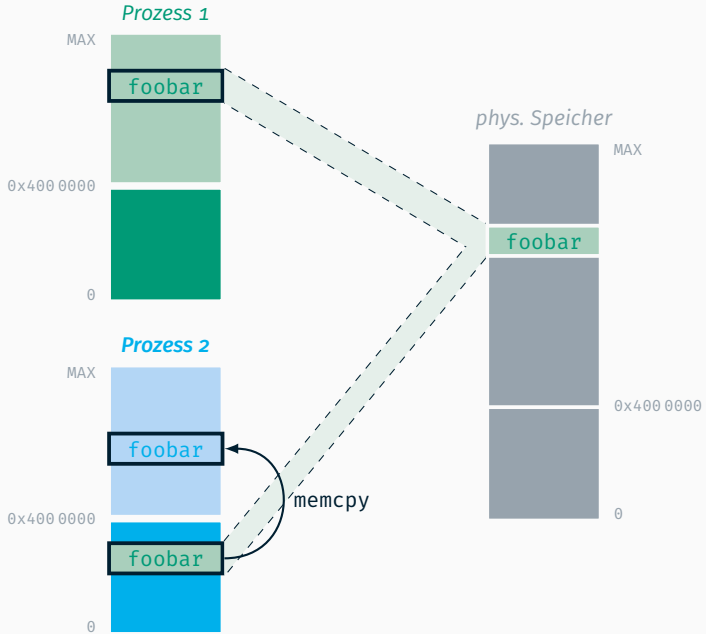
Adressraumübergreifendes Kopieren

deep copy



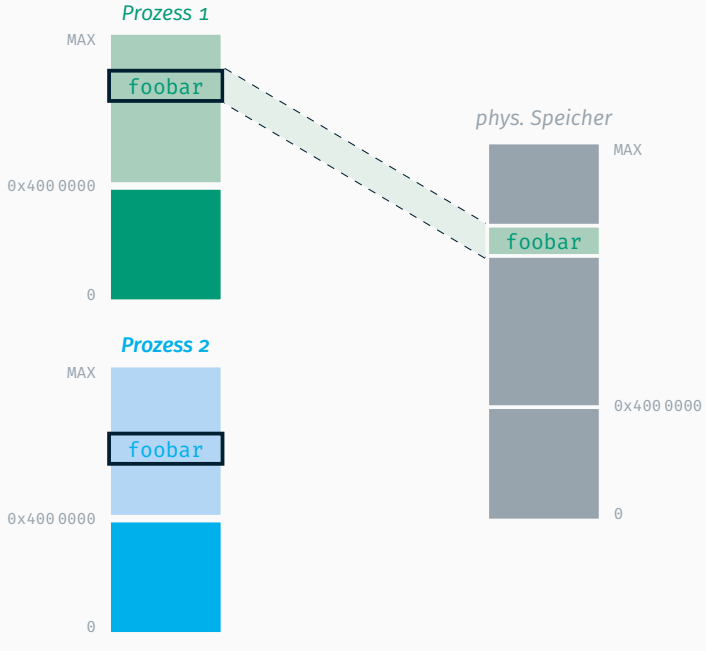
Adressraumübergreifendes Kopieren

deep copy



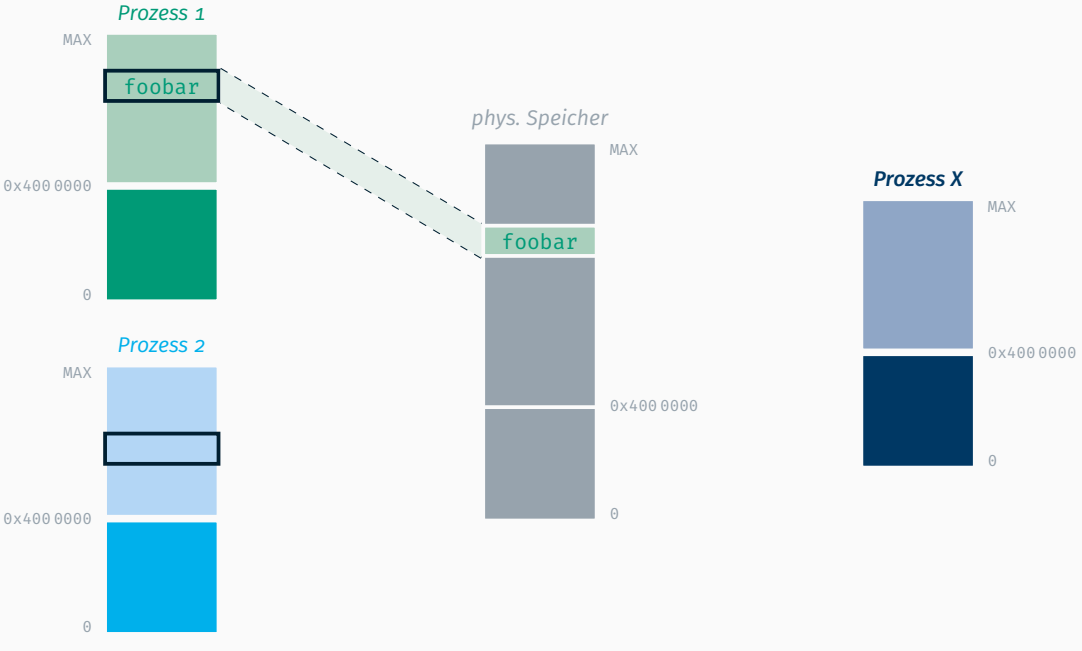
Adressraumübergreifendes Kopieren

deep copy



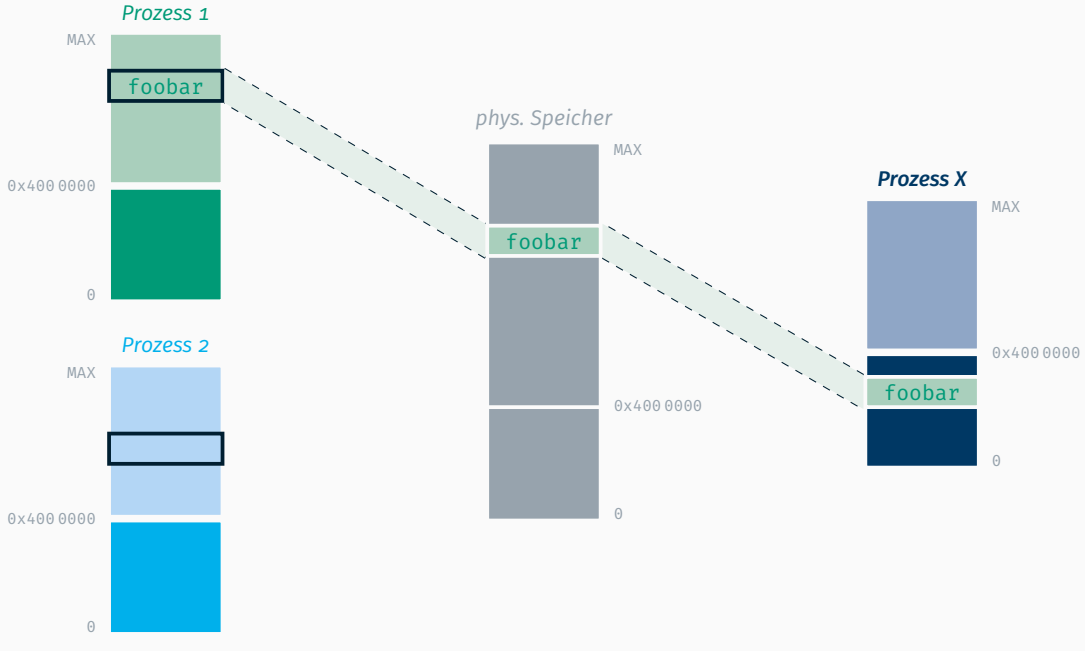
Adressraumübergreifendes Kopieren

generisches deep copy



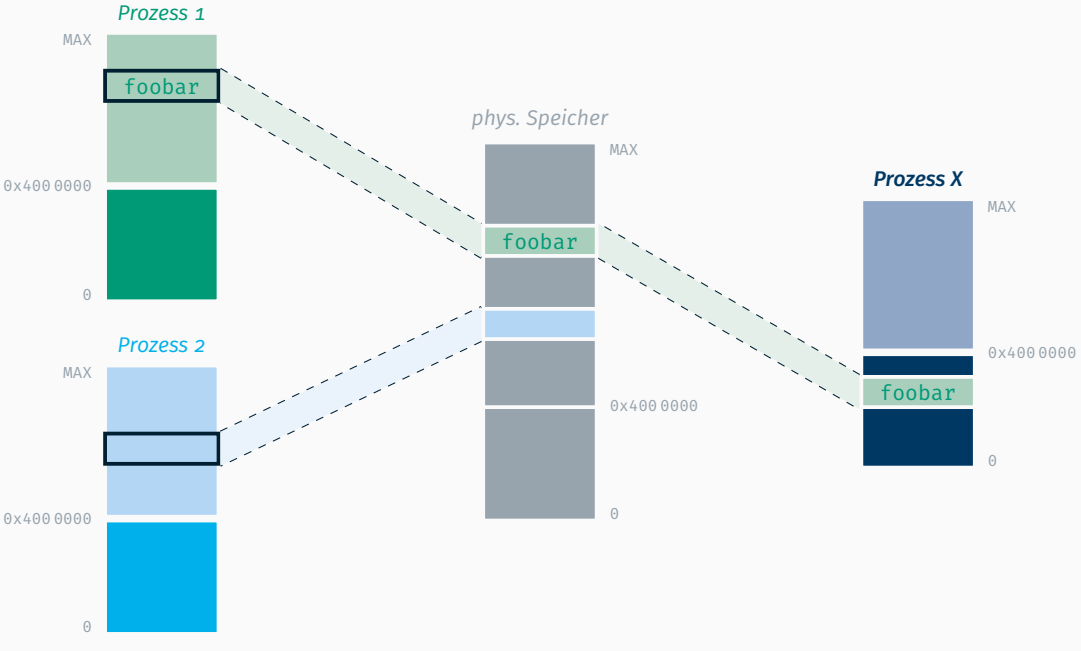
Adressraumübergreifendes Kopieren

generisches deep copy



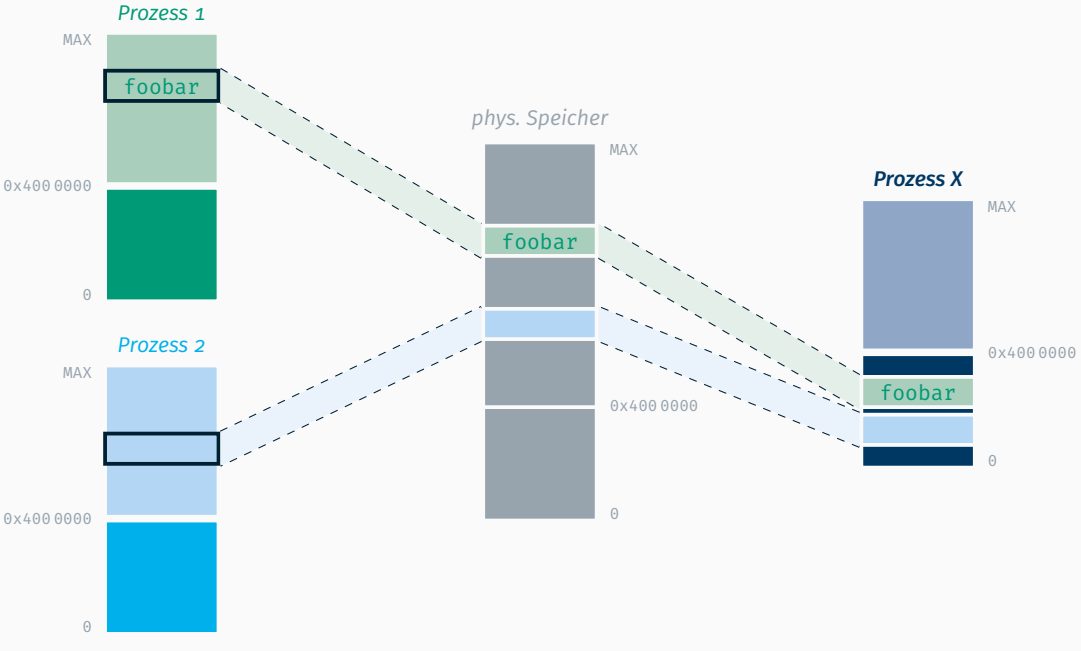
Adressraumübergreifendes Kopieren

generisches deep copy



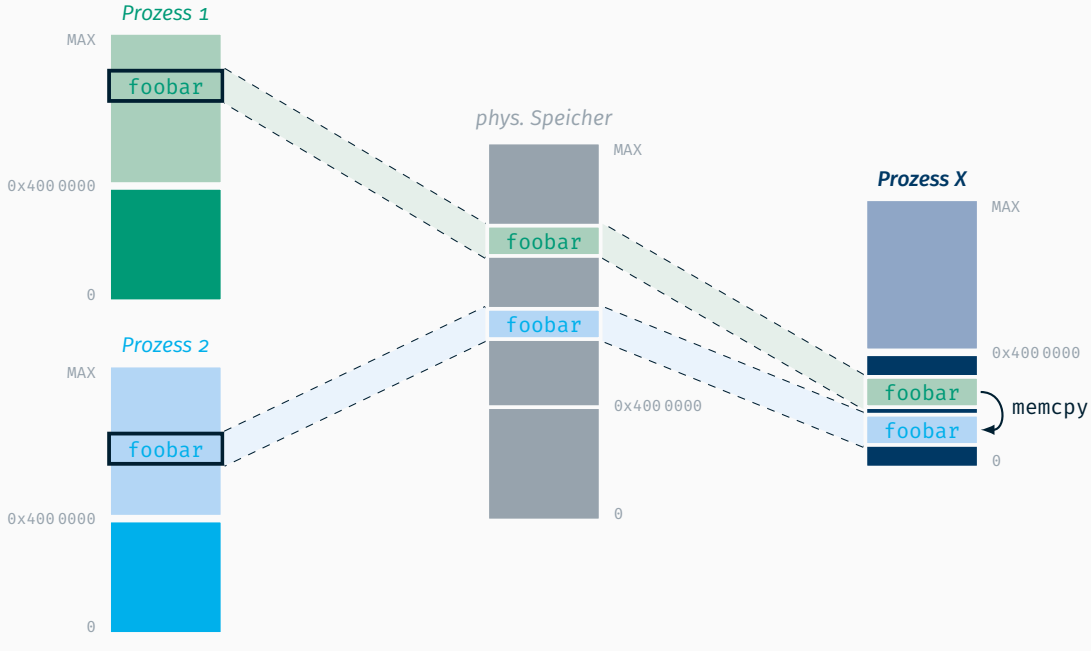
Adressraumübergreifendes Kopieren

generisches deep copy



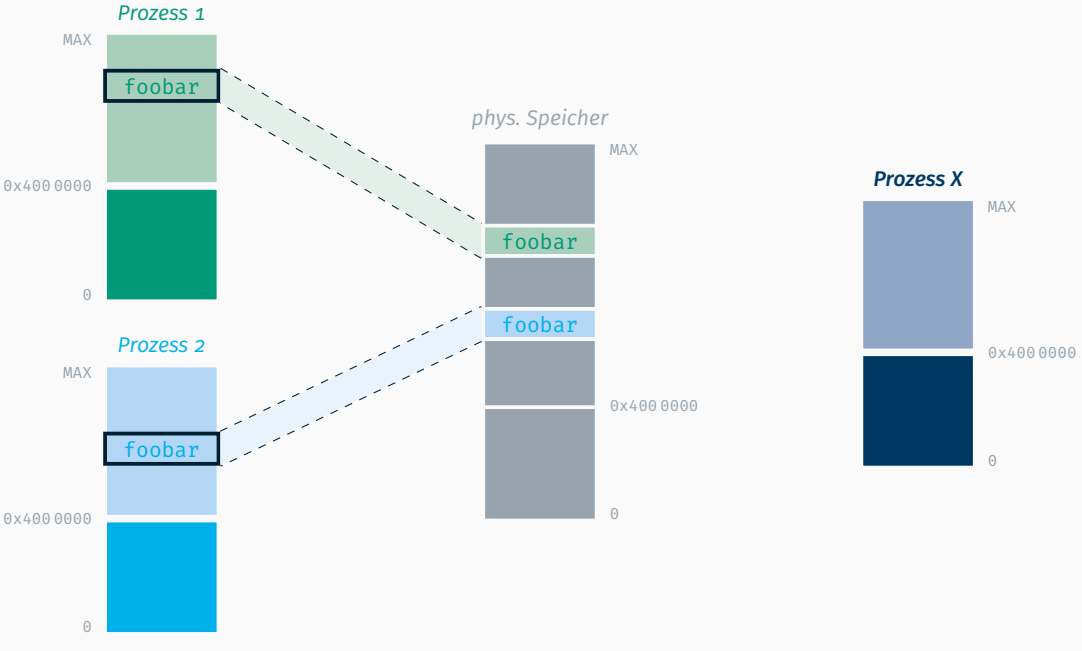
Adressraumübergreifendes Kopieren

generisches deep copy



Adressraumübergreifendes Kopieren

generisches deep copy



Nach Änderungen an der Seitenzuordnung (Mapping) unbedingt den TLB invalidieren!

Nach Änderungen an der Seitenzuordnung (Mapping) unbedingt den TLB invalidieren!

Vorzugsweise mit der Instruktion `invlpg`:

```
asm volatile("invlpg (%0) \n\t"
             :
             : "r" (virt_address)
             : "memory"
);
```

Dynamische Speicherverwaltung *(nur 7.5 ECTS)*

Neue Systemaufrufe für die Speicherverwaltung

- `void* map(void* addr, size_t size)`
  blendet (mindestens) `size` Bytes an genulltem Speicher im Userspace an der angegebenen virtuellen Adresse ein

Neue Systemaufrufe für die Speicherverwaltung

- `void* map(void* addr, size_t size)`
  blendet (mindestens) `size` Bytes an genulltem Speicher im Userspace an der angegebenen virtuellen Adresse ein
  - die Einblendung erfolgt seitenweise
    (`addr` muss jedoch nicht an einer Seitengrenze ausgerichtet sein!)

Neue Systemaufrufe für die Speicherverwaltung

- `void* map(void* addr, size_t size)`
  blendet (mindestens) `size` Bytes an genulltem Speicher im Userspace an der angegebenen virtuellen Adresse ein
  - die Einblendung erfolgt seitenweise
    (`addr` muss jedoch nicht an einer Seitengrenze ausgerichtet sein!)
  - bei `addr = NULL` wird ein passender Adressbereich ausgewählt
    → *program break* Zeiger (analog zu `brk`/`sbrk`) nützlich

Neue Systemaufrufe für die Speicherverwaltung

- `void* map(void* addr, size_t size)`
  blendet (mindestens) `size` Bytes an genulltem Speicher im Userspace an der angegebenen virtuellen Adresse ein
  - die Einblendung erfolgt seitenweise
    (`addr` muss jedoch nicht an einer Seitengrenze ausgerichtet sein!)
  - bei `addr = NULL` wird ein passender Adressbereich ausgewählt
    → *program break* Zeiger (analog zu `brk`/`sbrk`) nützlich
  - der Rückgabewert ist die virtuelle Adresse des Speichers

Neue Systemaufrufe für die Speicherverwaltung

- `void* map(void* addr, size_t size)`
  blendet (mindestens) `size` Bytes an genulltem Speicher im Userspace an der angegebenen virtuellen Adresse ein
  - die Einblendung erfolgt seitenweise
    (`addr` muss jedoch nicht an einer Seitengrenze ausgerichtet sein!)
  - bei `addr = NULL` wird ein passender Adressbereich ausgewählt
    → *program break* Zeiger (analog zu `brk`/`sbrk`) nützlich
  - der Rückgabewert ist die virtuelle Adresse des Speichers
  - falls die Adresse bereits eingeblendet/verwendet wird oder außerhalb des Userspace liegt, so wird ein Fehler (`NULL`) zurückgegeben

Neue Systemaufrufe für die Speicherverwaltung

- `void* map(void* addr, size_t size)`
  blendet (mindestens) `size` Bytes an genulltem Speicher im Userspace an der angegebenen virtuellen Adresse ein
  - die Einblendung erfolgt seitenweise
    (`addr` muss jedoch nicht an einer Seitengrenze ausgerichtet sein!)
  - bei `addr = NULL` wird ein passender Adressbereich ausgewählt
    → *program break* Zeiger (analog zu `brk`/`sbrk`) nützlich
  - der Rückgabewert ist die virtuelle Adresse des Speichers
  - falls die Adresse bereits eingeblendet/verwendet wird oder außerhalb des Userspace liegt, so wird ein Fehler (`NULL`) zurückgegeben
- `void exit()`
  beendet die Anwendung und gibt alle Resourcen wieder frei

Neue Systemaufrufe für die Speicherverwaltung

- `void* map(void* addr, size_t size)`
  blendet (mindestens) `size` Bytes an genulltem Speicher im Userspace an der angegebenen virtuellen Adresse ein
  - die Einblendung erfolgt seitenweise
    (`addr` muss jedoch nicht an einer Seitengrenze ausgerichtet sein!)
  - bei `addr = NULL` wird ein passender Adressbereich ausgewählt
    → *program break* Zeiger (analog zu `brk`/`sbrk`) nützlich
  - der Rückgabewert ist die virtuelle Adresse des Speichers
  - falls die Adresse bereits eingeblendet/verwendet wird oder außerhalb des Userspace liegt, so wird ein Fehler (`NULL`) zurückgegeben
- `void exit()`
  beendet die Anwendung und gibt alle Resourcen wieder frei
  - zum Testen sollen vor dem Start und nach dem Beenden die Zahl der freien Seiten ausgegeben werden (und gleich bleiben!)

Fragen?

**Die Bearbeitung von Aufgabe 6 kann direkt im Anschluss beginnen,
Detaillierte Implementierungstipps in der Tafelübung nächste Woche!**