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

25. Juni 2025

Dustin Nguyen, Maximilian Ott & Phillip Raffeck

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

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

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. 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!

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)*

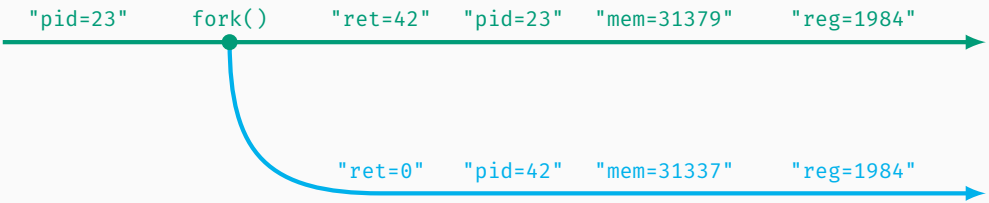
Neue Systemaufrufe für Prozessverwaltung

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



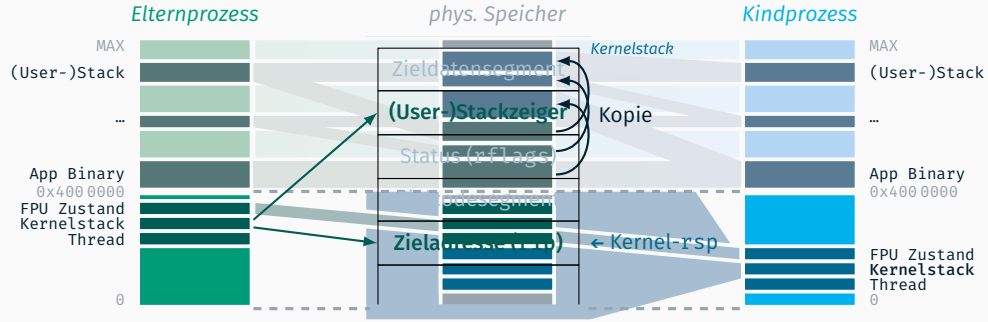
Zu duplizierender Zustand eines Prozesses

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

→ **kompletten Userspace-Teil des virtuellen Speichers duplizieren**

Duplizieren eines Prozesses



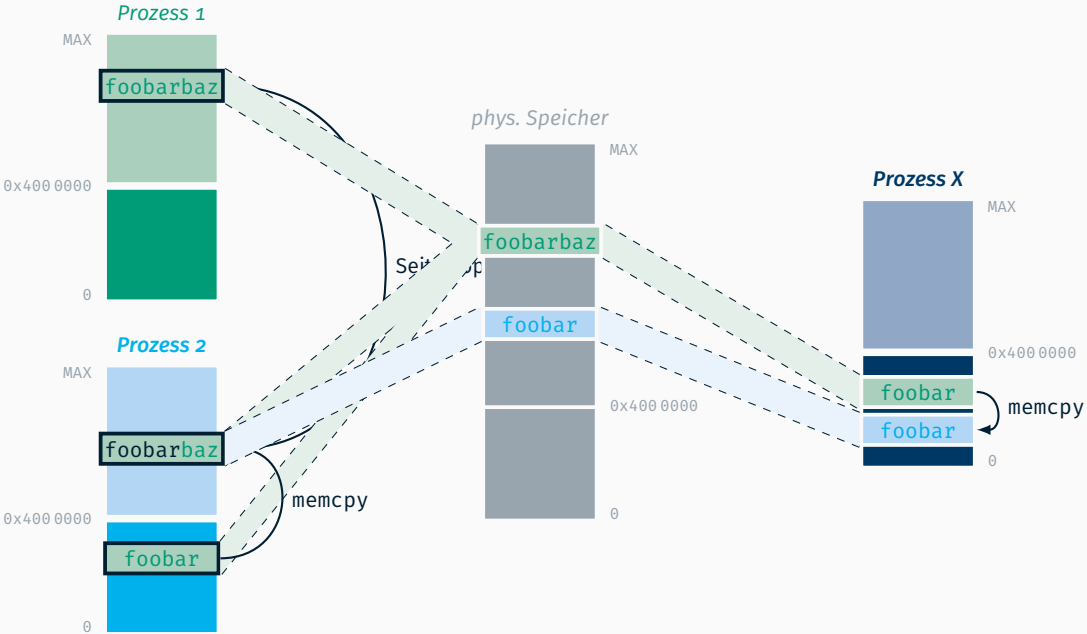
Kindprozess erstellen

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

Virtuellen Adressraum vorbereiten

Adressraumübergreifendes Kopieren

generisches deep copy



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
  - 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!
Denkt an die Evaluation!