2. Prozeßverwaltung

2.1 Einführung

Ein Prozeß (process, task) ist ein in Ausführung befindliches Programm bestehend aus

- einer Folge von Maschinenbefehlen, die durch das ausgeführte Programm festgelegt sind (program code, text section) ,
- dem Inhalt des Stapel-Speichers (stack), auf dem temporäre Variablen und Parameter für Funktionsaufrufe verwaltet werden,
- dem Inhalt des Speichers, in dem die globale Daten des Prozesses gehalten werden (data section) und
- Verwaltungsinformationen, die den aktuellen Bearbeitungszustand (Kontext) beschreiben.

Bemerkung: Der aktuelle Bearbeitungszustand wird u.a. durch den Programmzähler und die Registerinhalte des Prozessors beschrieben (engl. internal state).

Prozeßverwaltung:

- Die Verwaltungsinformationen speichert das Betriebssystem in der Prozeßtabelle, die pro Prozeß einen Eintrag enthält.
- Ein Eintrag in der Prozeßtabelle heißt Prozeßkontrollblock oder auch Prozeßleitblock (engl. process control block).

Ein Prozeßkontrollblock beinhaltet:

- Prozeßidentifikationsnummer (pid)
- Platz für Registerinhalte des Prozessors (CPU)
- Platz für Programmzähler
- Zeiger auf Text-, Daten-, Stacksegment
- Daten über Ein-/ Ausgabegeräte, die dem Prozeß zugeordnet sind
- Daten über geöffnete Dateien und Netzwerkverbindungen
- Prozeßzustand
- Daten, die die Bearbeitungsfolge regeln (Prioritäten)
- Accounting-Daten über benutzte Betriebsmittel/Ressourcen (z.B. die bisher verbrauchte CPU-Zeit)
- Parameter, die die Zugriffsrechte des Prozesses beschreiben

Process Control Block (PCB)

Definition: OS data structure which contains the information needed to manage a process (one PCB per process)

Process identifiers	• IDs of process, parent process, and user

CPU state	• User-visible registers
	• Control and status registers:
	  • Stack pointer (SP)
	  • Program counter (PC)
	  • Processor status word (PSW)

Control information	• Scheduling information:
	  • Process state, priority, awaited event
	• Accounting information:
	  • Amount of memory used, CPU time elapsed
	• Memory management:
	  • Location and access state of all user data
	• I/O management:
	  • Devices currently opened (files, sockets)

Quelle: J. Schiller, FU Berln

Man unterscheidet:
Hardwarekontext/CPU State: Registerinhalte, Befehlszähler, ...
Softwarekontext: Prozeß-ID, Prozeßzustand, ...

Datenstruktur für die Realisierung der Prozeßtabelle: meist als doppelt verkettete Liste

Definition 12:

Ein Prozeß befindet sich in genau einem der folgenden Zustände:

rechnend: Der Prozessor ist dem Prozeß zugeteilt.

rechenbereit: Der Prozeß ist ausführbar und wartet auf Prozessorzuteilung.

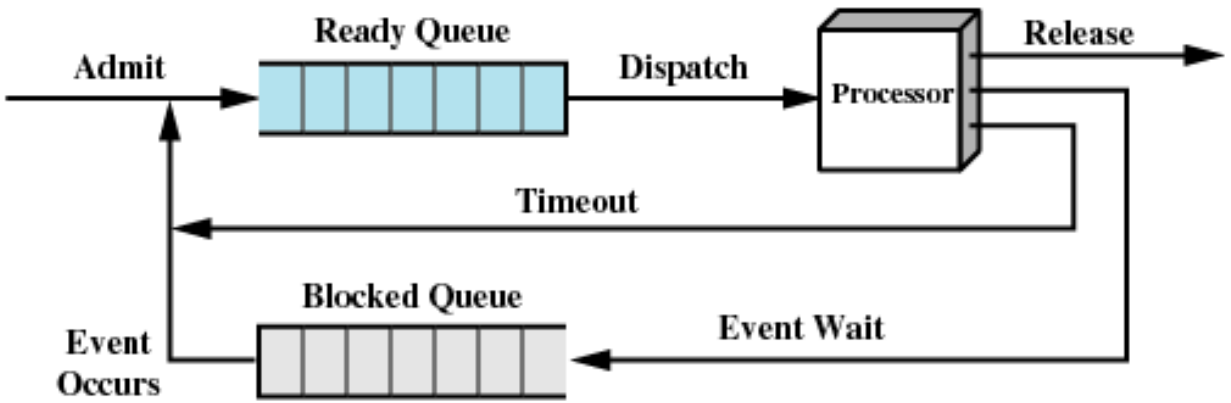
blockiert: Der Prozeß wartet auf ein Ereignis, z.B. auf Eingabe, Ausgabe oder ein Semaphor.

Implementation of Process States

Assign process to different queues based on state of required resources

Two queues:

- Ready processes (all resources available)
- Blocked processes (at least one resource busy)



Quelle: J. Schiller, FU Berln

Abnormales Prozeßverhalten

Definition 13:

Ein Prozeß befindet sich in einem:

Deadlock := Der Prozeß befindet sich im Zustand blockiert und es tritt kein Zustandswechsel mehr ein.

Livelock := Der Prozeß wechselt zwar zwischen den Zuständen, kommt aber in seiner Programmausführung nicht weiter.

Starvation := Man sagt ein Prozeß verhungert, falls er sich im Zustand rechenbereit befindet und es tritt kein Zustandswechsel ein.

Unterbrechungen

Unterbrechungen sind das Schlüsselkonzept zur effizienten Ablaufsteuerung.

Definition 14:

1. Hardware Interrupts (auch asynchrone Unterbrechung) werden von externen Ereignissen außerhalb des laufenden/rechnenden Prozesses verursacht.
2. Software Interrupts (auch: Traps, Program Interrupts) (synchrone Unterbrechung) werden von dem laufenden/rechnenden Prozeß selber verursacht.

Beispiele:

1. Hardware Interrupts (asynchrone Unterbrechung)
   - I/O-Controller (Platte, Tastatur, Network Interface Card (NIC))
   - Timer: ermöglicht dem Betriebssystem periodisch bestimmte Arbeiten durchzuführen
2. Software Interrupts (auch: Traps, Program Interrupts) (synchrone Unterbrechung)
   - Systemdienstaufruf (System Call): `read`, `write`, `open`, `close`, `...`
   - Division durch Null
   - ungültige Speicheradresse

E. W. Dijkstra: *My recollections of operating system design*, Operating Systems review, Vol. 39, Number 2, 2005.

Definition 15:

Der Unterbrechungsvektor enthält für jeden Unterbrechungstyp (Disk-Interrupt, Terminal-Interrupt, ...) die Adresse einer zugehörigen Unterbrechungsbehandlungsroutine.

Beispiel für eine asynchrone Unterbrechung:

Prozeß A rechnet, Prozeß B wartet auf Eingabe von der Platte.

Es erfolgt ein Disk-Interrupt, wenn die Daten für Prozeß B eingelesen sind.

Ablauf:

1. Programmzähler etc. von Prozeß A werden auf den Stack geschrieben (Hardware).
2. Unterbrechungsbehandlungsroutine wird geladen.
3. Unterbrechungsbehandlungsroutine schreibt die aktuellen Registerinhalte in den Prozeßleitblock, der Stack wird rekonstruiert (Prozeßzustand von A := rechenbereit) (Assembler).
4. C-Routine setzt Prozeß B auf rechenbereit.
5. Scheduler wird aktiv.

Erzeugung von Prozessen

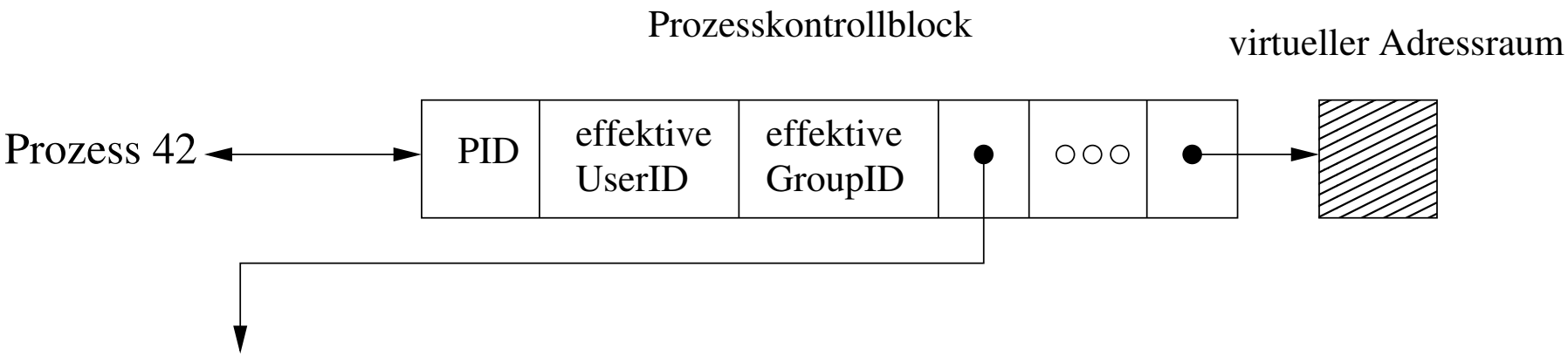
- Der erste Prozeß wird beim Systemstart vom Betriebssystem erzeugt.
- Neue Prozesse können von einem existierenden Prozeß durch spezielle Systemaufrufe erzeugt werden.
- POSIX-Systeme benutzen hierzu den `fork()` Systemaufruf. Der neu erzeugte Prozeß (child) ist eine **Kopie** des erzeugenden Prozesses (parent), der nebenläufig zum erzeugenden Prozeß läuft.
- Da beliebige Prozesse neue Prozesse erzeugen können, entsteht eine Hierarchie von Prozessen.

Ablauf von `fork()`:

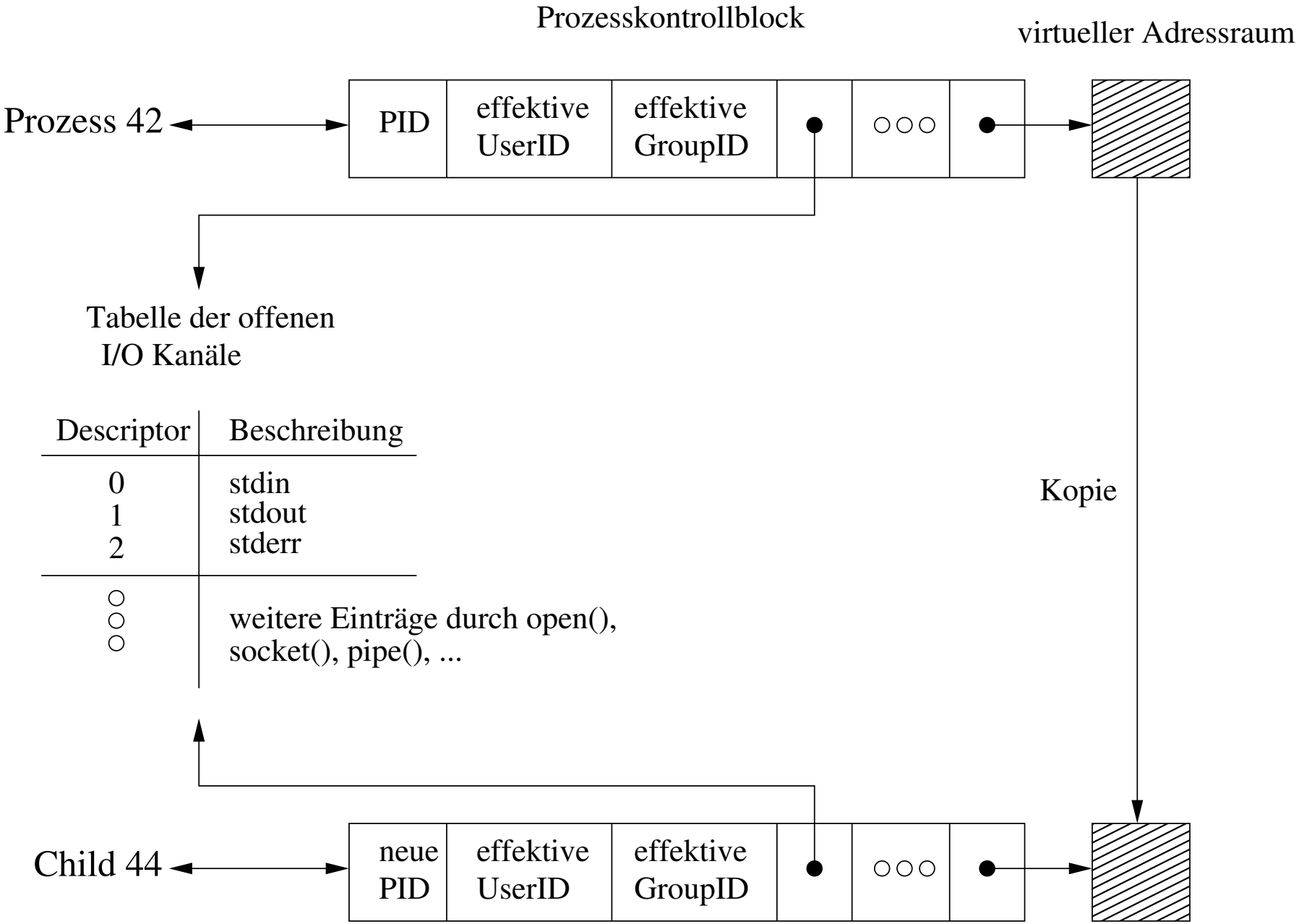
1. Neuen Prozeßkontrollblock erstellen:
   - Teil der Einträge werden neu gesetzt:
     PID, PPID, ...
   - Teil der Einträge werden auf Null gesetzt:
     CPU-Verbrauch, Timer, ...
   - Teil der Einträge werden vom Elternprozeß kopiert:
     Zeiger auf offene Dateideskriptoren, Prozeßgruppe,
     Signalzustand, Nice-Wert, ...
2. Der Adressraum des Elternprozesses wird mittel `vm_fork()` kopiert.
3. Der Kindprozeß wird in die Warteschlange der rechenbereiten Prozesse (Runqueue) eingereiht.

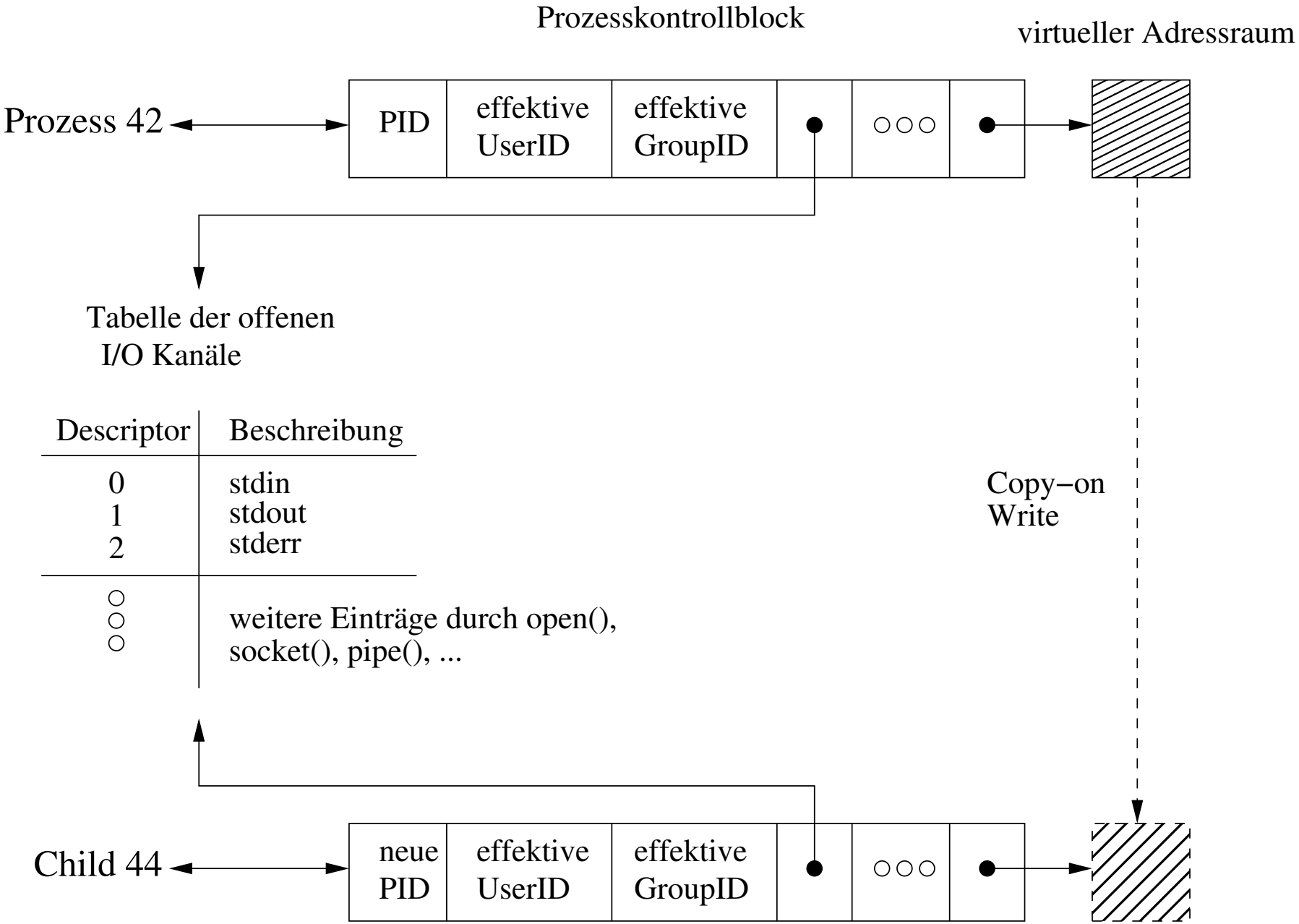
Anekdote: Das B-S-D Maskottchen Beastie

Hier eine Version des BSD Maskottchen von John Lasseter (Chief Creator Officer bei Pixar). Es tauchte erstmals 1988 auf dem Buch-Cover auf.

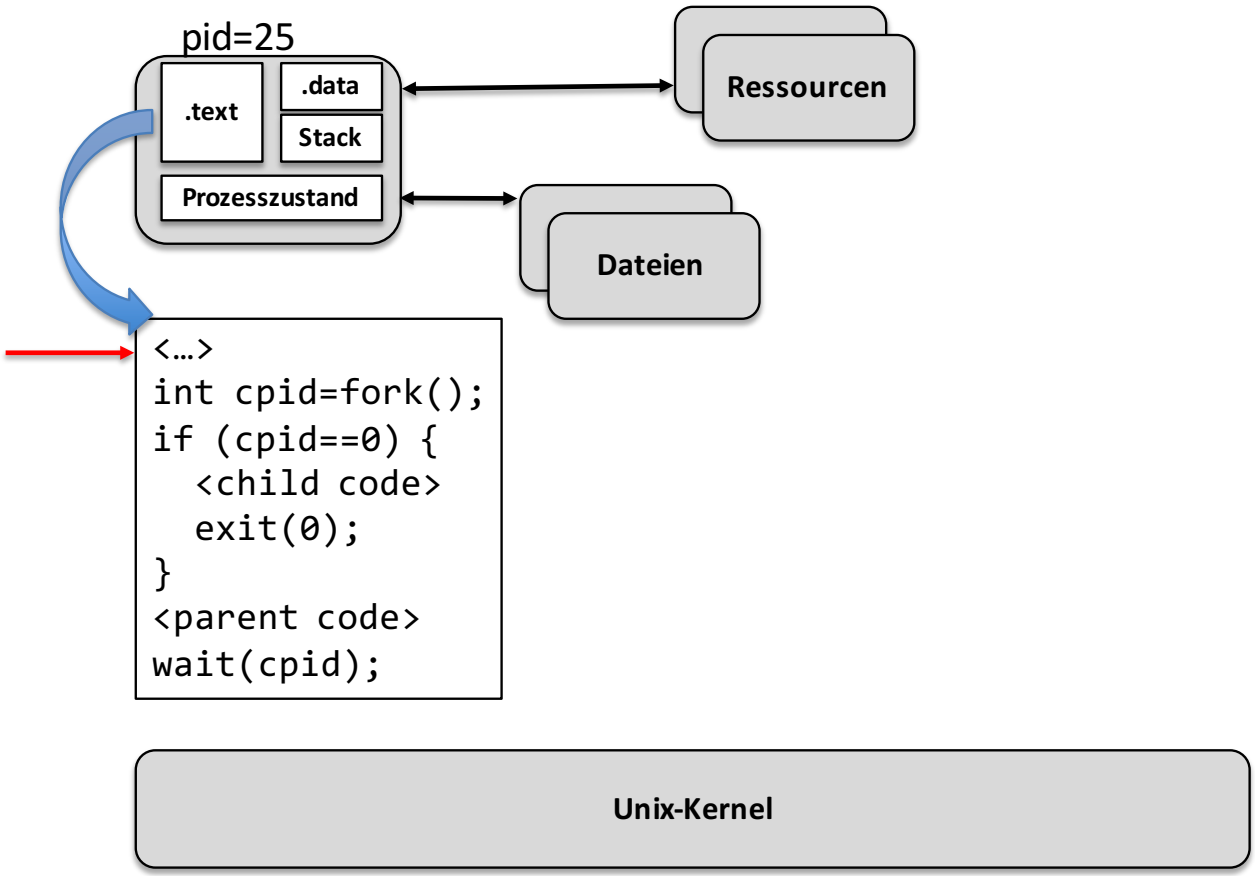


Descriptor	Beschreibung
0	stdin
1	stdout
2	stderr
○ ○ ○	weitere Einträge durch open(), socket(), pipe(), ...


Prozesskontrollblock
virtueller Adressraum
Prozess 42
PID
effektive UserID
effektive GroupID
○○○
Tabelle der offenen I/O Kanäle
Descriptor
Beschreibung
0
1
2
stdin
stdout
stderr
○○○
weitere Einträge durch open(), socket(), pipe(), ...
Kopie
Child 44
neue PID
effektive UserID
effektive GroupID
○○○

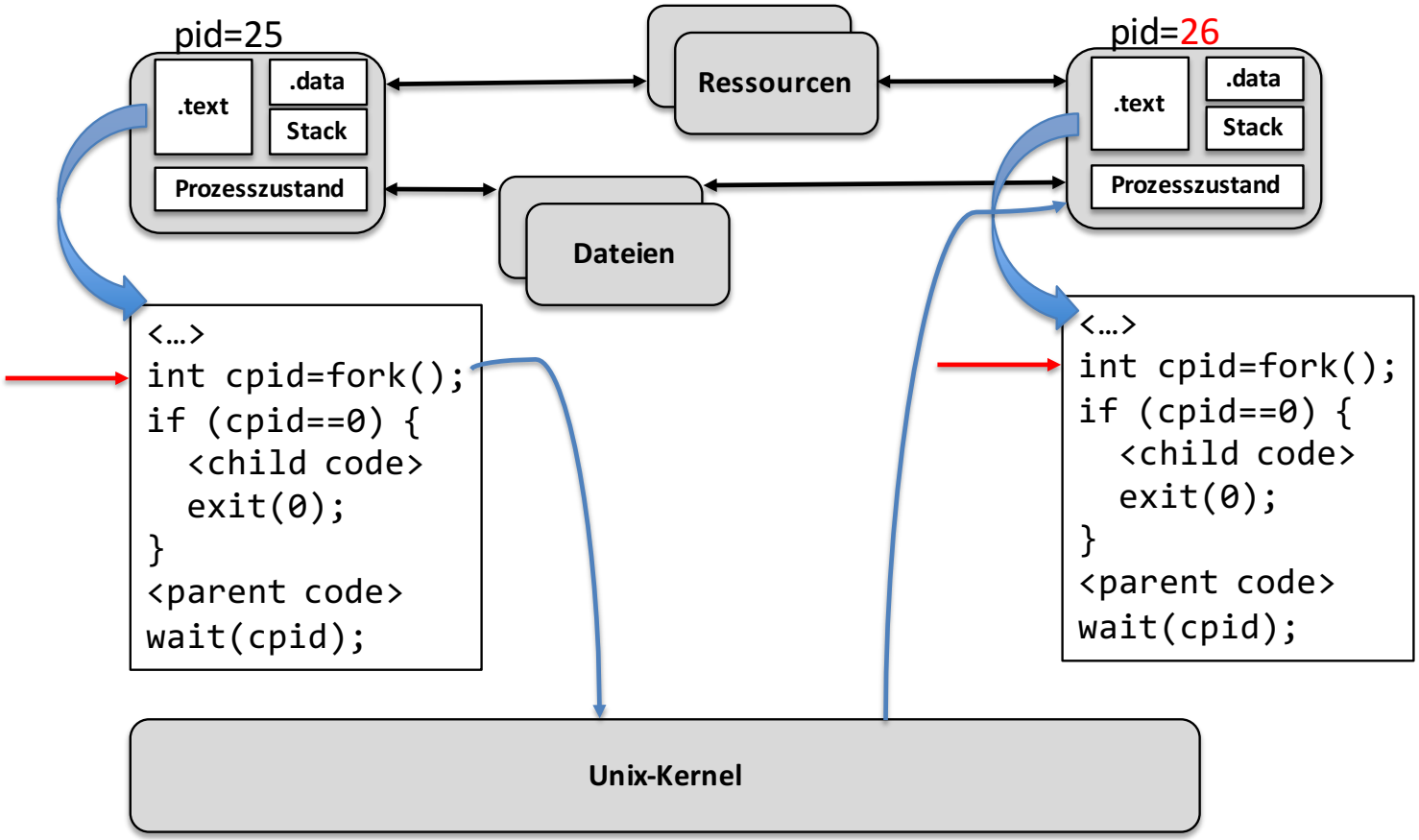

Prozesskontrollblock
virtueller Adressraum
Prozess 42
PID
effektive UserID
effektive GroupID
○○○
Tabelle der offenen I/O Kanäle
Descriptor
Beschreibung
0
1
2
stdin
stdout
stderr
○
○
○
weitere Einträge durch open(), socket(), pipe(), ...
Copy−on Write
Child 44
neue PID
effektive UserID
effektive GroupID
○○○

Ablauf von fork[9]:



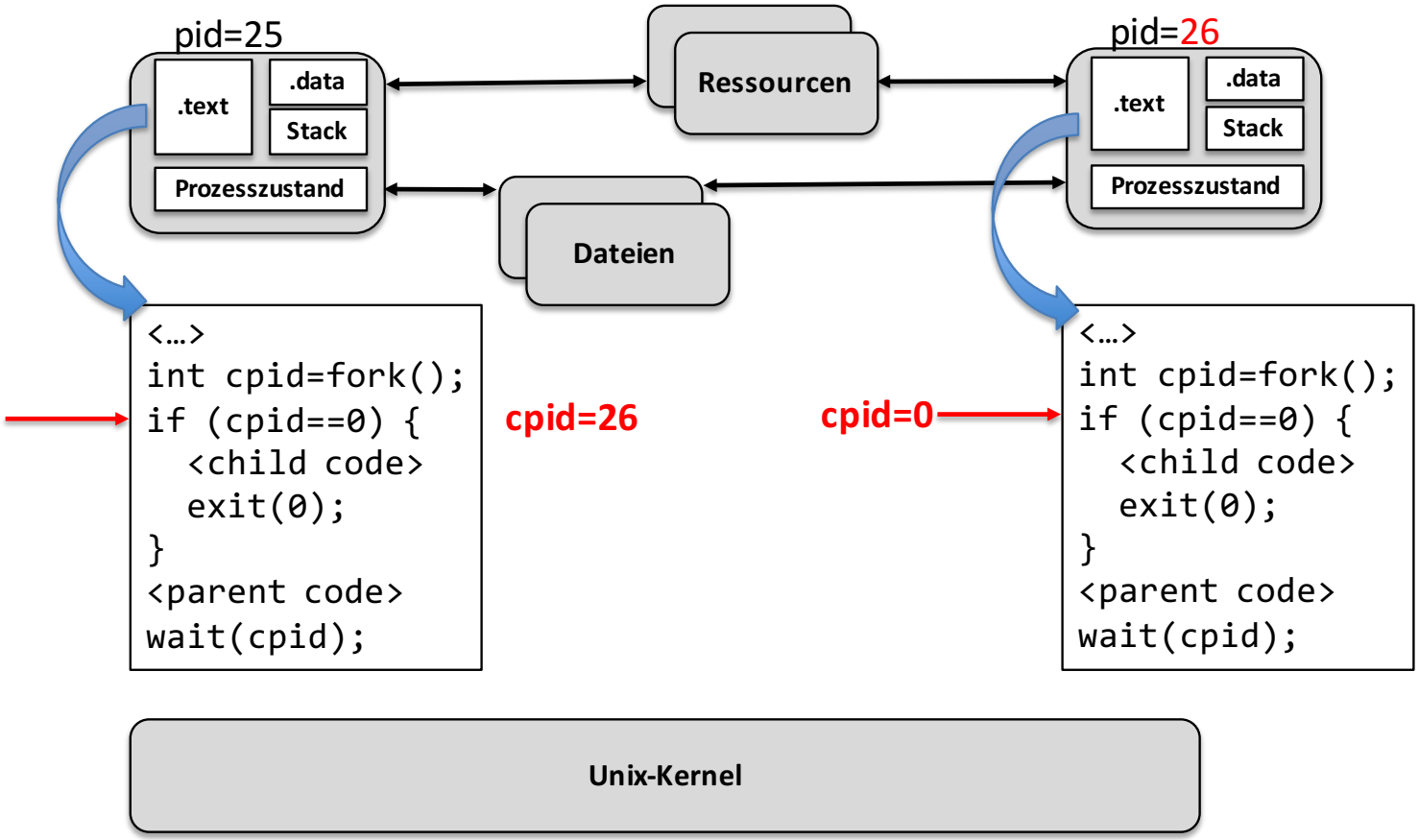
[9] Einfaches Beispiel von Michael Engel ohne execve() im <child code>

Ablauf von fork:



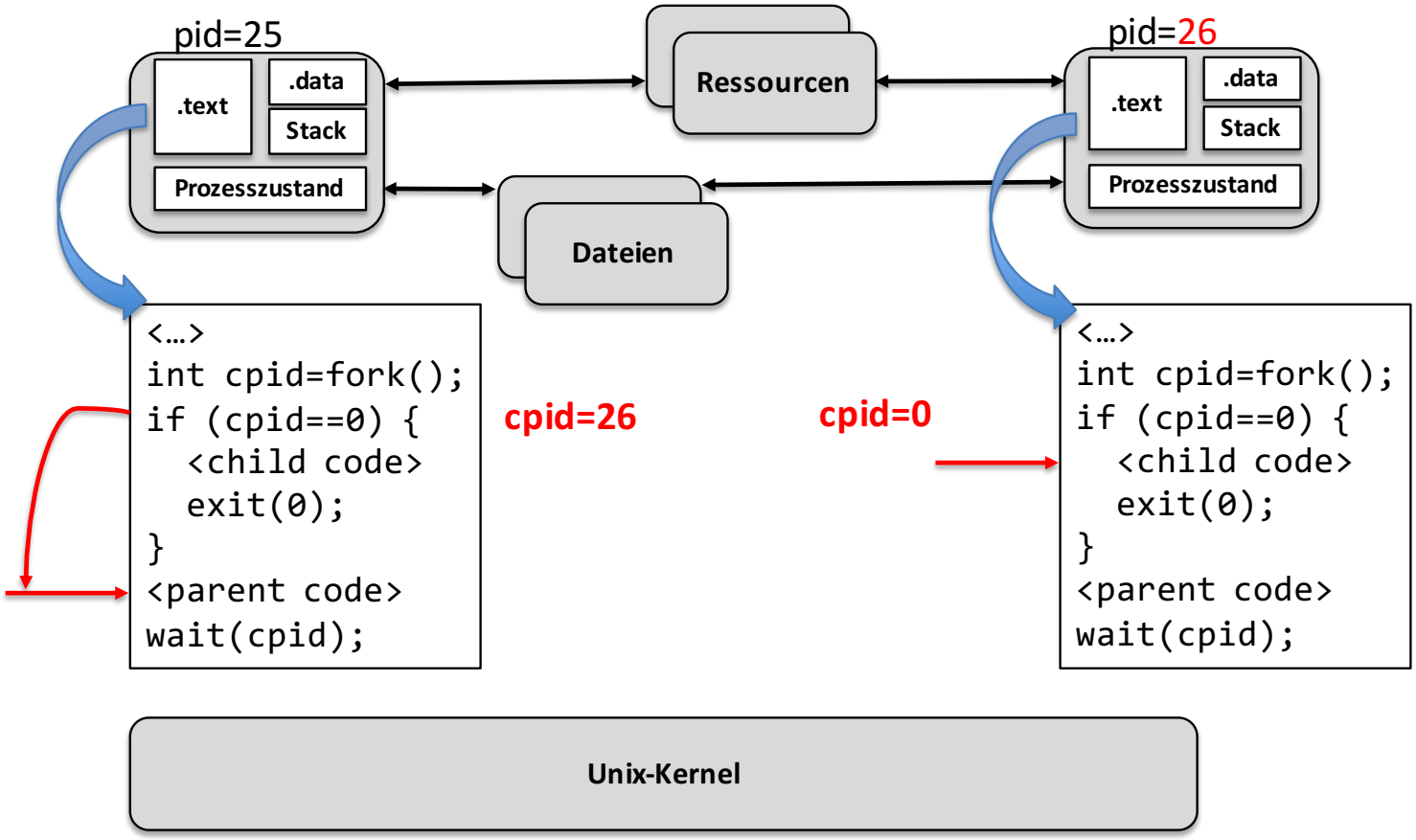
Der Prozeß mit ID = 26 ist eine Kopie vom Parent-Prozeß. Die Programmzähler stehen an derselben rot-markierten Stelle (im fork).

Ablauf von fork:

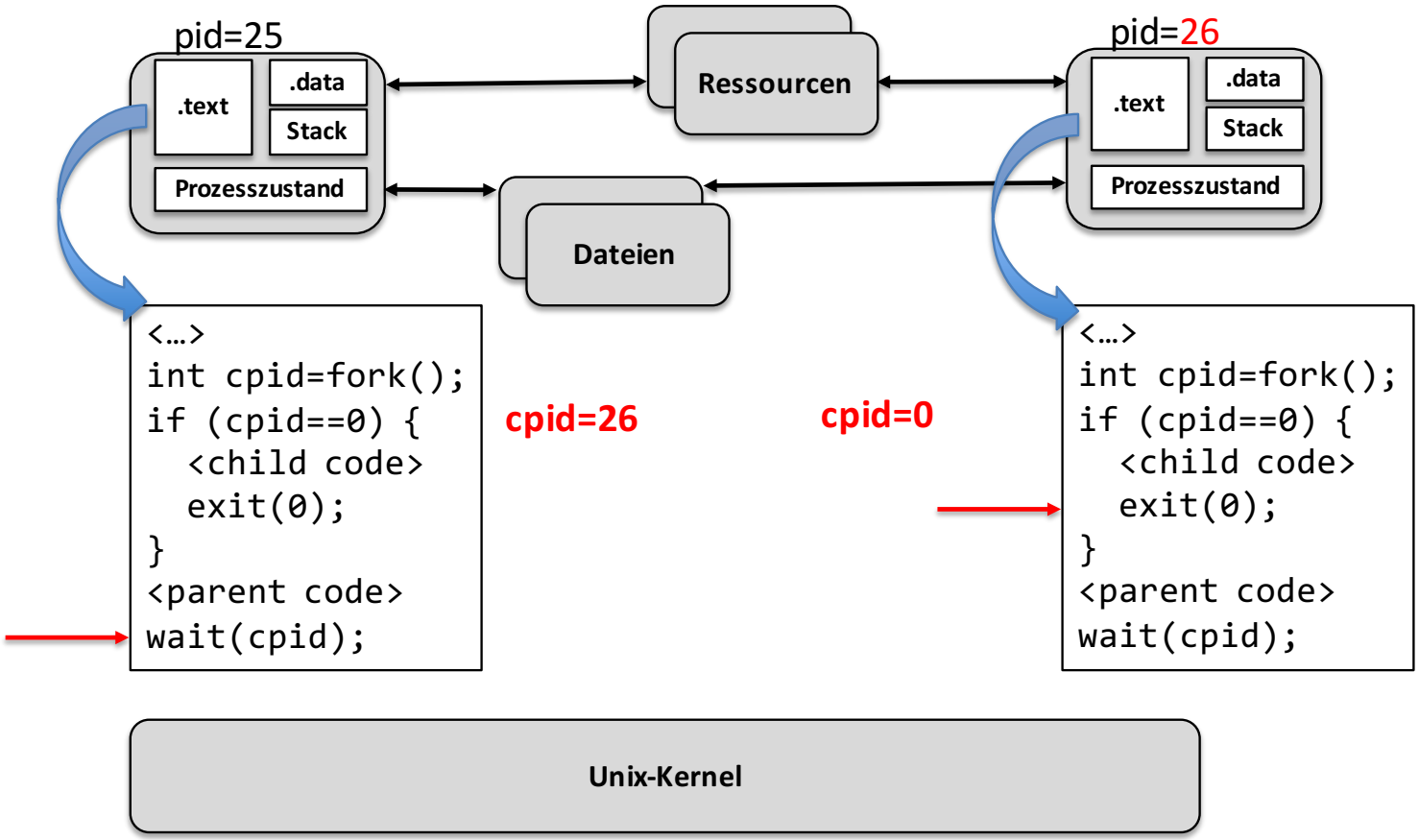


Beide Prozesse kehren aus dem `fork`-Aufruf zurück.
Unterschied: verschiedene Rückgabewerte!

Ablauf von fork:

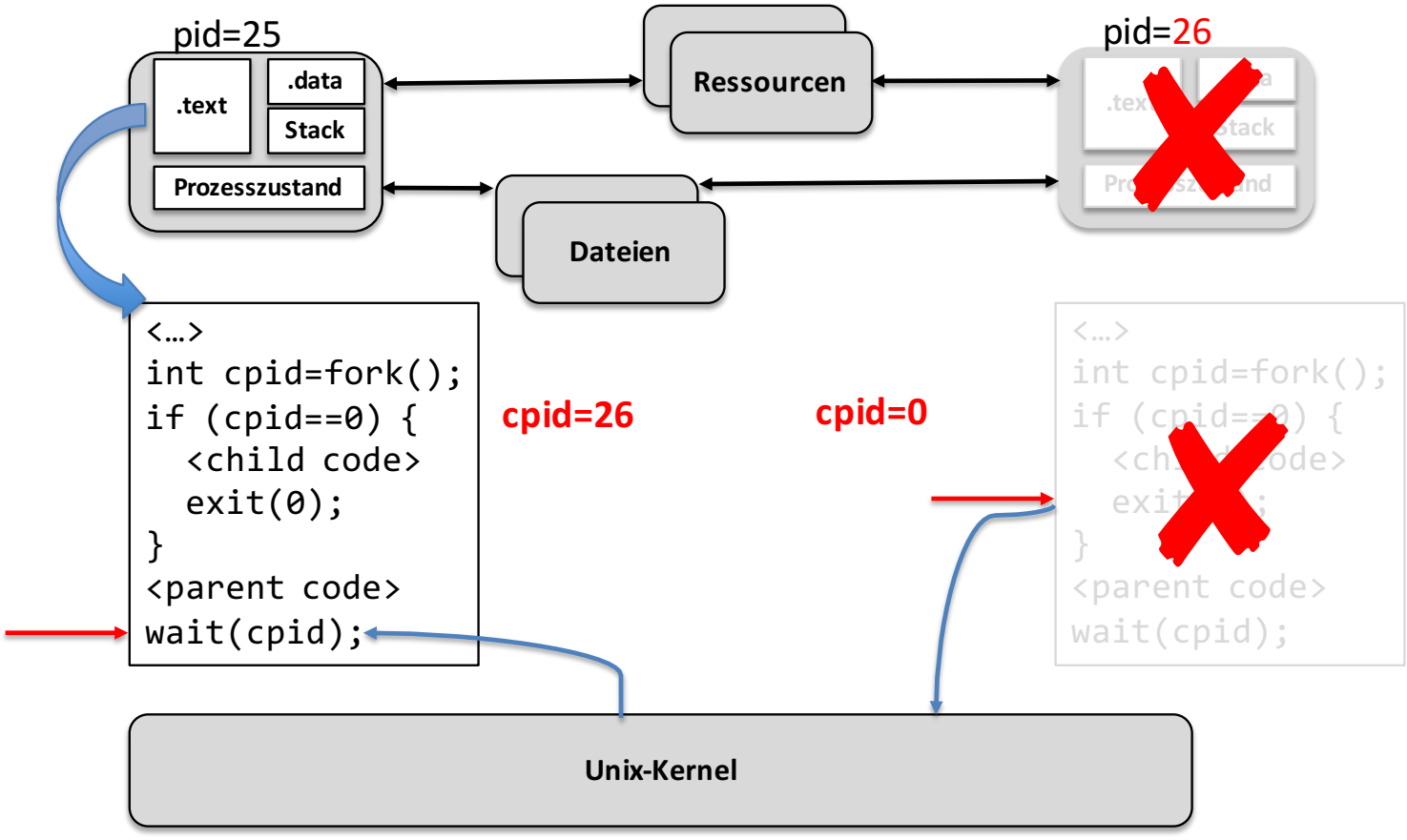


Ablauf von fork:



Der Parent-Prozeß wartet, bis der Kindprozeß sich beendet.

Ablauf von fork:



Beenden von Prozessen

- Prozesse können sich selbst durch den `exit()` Systemaufruf beenden und damit aus dem System entfernen.
- Der Ergebniswert (return code) eines sich beendenden Prozesses wird an den erzeugenden Prozeß (Parent) weitergeleitet.
- Auf die Beendigung eines erzeugten Prozesses kann in POSIX-Systemen mit Hilfe des `wait()` Systemaufrufs gewartet werden.
- Ein Prozeß kann die von ihm erzeugten Prozesse beenden, falls
  $\rightarrow$ ein Prozeß nicht länger gebraucht wird oder falls
  $\rightarrow$ der erzeugende Prozeß sich beendet und es nicht gewünscht/erlaubt ist, daß die erzeugten Prozesse unabhängig weiter existieren (cascading termination).
- Prozesse können in POSIX-Systemen mit Hilfe des `execve()` Systemaufrufs das ausgeführte Programm durch ein neues ersetzen.

Ablauf `execve()`:

1. Speicherplatz für neues Programm reservieren (Text, Data, Stack)
2. Falls Reservieren klappt, wird der alte Adressraum freigegeben
3. Neue Programmargumente und die Umgebungsvariablen werden auf den Stack geschrieben
4. Register initialisieren (Programmzähler, Stack Pointer, ...)

Bei Erfolg kehrt `execve()` nicht zurück, sondern Text- und Data-Section, sowie Stack werden von dem neu geladenen Programm überschrieben.

Beispiel: Einfacher Kommandointerpreter

```
while (1) { /* repeat forever */
   type_prompt(); /* display prompt on screen */
   read_command(); /* read input from the terminal */
   pid = fork(); /* create a new process */
   if (pid < 0) { /* repeat if system call failed */
      perror ("fork");
      continue;
   }
   if (pid != 0)  { /* parent process*/
      waitpid(pid, &status, 0); /* parent process */
      } else { /* child process */
      execve(command, params, 0); /* execute command */
   }
}
```

Beispiel: Linux implementiert `fork()` on top of:

```
NAME
       clone, __clone2, clone3 - create a child process
```

By contrast with fork(2), these system calls provide more precise control over what pieces of execution context are shared between the calling process and the child process. For example, using these system calls, the caller can control whether or not the two processes share the virtual address space, the table of file descriptors, and the table of signal handlers. These system calls also allow the new child process to be placed in separate namespaces(7).

Bem.: POSIX spezifiziert:

`posix_spawn` – spawn a process

The posix_spawn() function is used to create a new child process that executes a specified file.
The posix_spawn() function provides the functionality of a combined fork(2) and exec(3), with some optional housekeeping steps in the child process before the exec(3).
These functions are not meant to replace the fork(2) and execve(2) system calls. In fact, they provide only a subset of the functionality that can be achieved by using the system calls.

Quelle: https://man7.org/linux/man-pages/man3/posix_spawn.3.html