Inhalt

- Unix Prozesse

Prozesse

Eigenschaften

- Nebenläufigkeit wird mit Prozessen (tasks) oder threads umgesetzt.
- Unix-Kommandos starten typischerweise einen Prozess.
- Prozesse haben einen eigenen Adressraum, der gegenüber anderen Prozessen geschützt ist.
- Prozesse haben einen Zustand.
- Prozesse haben einen eigenen Kontext.
- Prozesse können andere Prozesse erzeugen
  → der Kontext wir vererbt

Prozesskontext

Der Kontext umfasst alle Informationen, die einen Prozess ausmachen:

- einen eindeutigen Identifier, process-id: **PID**
- den Programm- und Datenbereich
- den Speicherbereich (Adressraum)
- program-counter und Registerinhalte
- die Liste der Umgebungsvariablen
- die Rechte (permissions)
- die Priorität und das Schedulingverfahren
- zugeordnete Dateien und Geräte
- sonstige Informationen: Resourcenverbrauch, ...

Erzeugung von Prozessen

a) aus einem Kommandoprozessor
(shell, z.B. bash, *Bourne-again-shell*)

- eine ausführbare Datei wird per **Name** gestartet
- es können **Parameter** angegeben werden

Beispiele:
```
$ ls -l
$ cp ./file1.txt ./file2.txt
```

- ein C-Programm hat über die Parameter des main Zugriff auf die *Kommandozeilenparameter*:

`int argc` : Anzahl der Parameter
`char* argv[]` : enthält die Parameter als strings, `argv[0]` enthält den eigenen Programmnamen

- die shell ist **parent** des gestarteten Prozesses
  ➔ diverse Kontexteigenschaften werden vererbt
- Prozesse können im Hintergrund gestartet werden (*non-blocking call*):

```
$ machWatt &
```

- die IO-Deskriptoren sind standardmäßig konfiguriert:
  - **STDIN_FILENO** (Wert 0) ➔ Tastatur/Terminal
  - **STDOUT_FILENO** (Wert 1) ➔ Bildschirm/Terminal
  - **STDERR_FILENO** (Wert 2) ➔ Bildschirm/Terminal

POSIX Konstanten in **unistd.h**

- die Deskriptoren können mit der shell umgeleitet werden:

```
$ machWatt < in.dat > out.txt
```

- die shell kann *stdin* und *stdout* von Prozessen mit **pipes** synchronisiert verbinden:

```
$ machWatt | machNochWatt
```

Einige prozessrelevante shell-Kommandos:

- Prozessstatus anzeigen:

```
$ ps -el
```

- Prozess beenden:

```
$ kill -SIGKILL 4242
```

9 — PID

ultimativ beenden,
keine „geordnete“ Terminierung,
Dateien nicht geschlossen ...

```
$ kill -SIGTERM 4242
```

15

sichere Standard-Terminierung,
„Aufforderung“ zur Terminierung

- auf die Terminierung eines Prozesses warten:

```
$ wait 4242; machWeiter
```

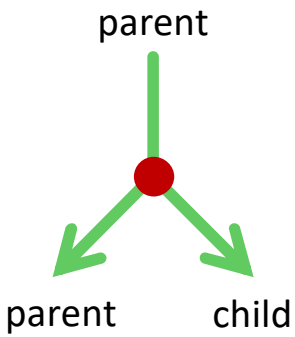
➔ auf Terminierung des Prozesses mit PID 4242 warten, dann den Prozess **`machWeiter`** starten

b) Prozesserzeugung aus einem C-Programm

1) mit der `stdlib`-Funktion **`system(...)`**

- **`system`** startet Prozesse wie ein shell-Kommando
- **`system`** ist ein blockierender Aufruf

```
system("ls -l");
```



2) mit dem system-call **`fork()`**

- der aufrufende Prozess wird dupliziert, insbesondere der Code!
- der neue Prozess ist child und erbt den Kontext
- der neue Prozess bekommt eine neue PID
  (Zugriff per **`getpid()`**)
- der neue Prozess kann seine PID und die seines parents (PPID) abfragen: **`getpid()`** und **`getppid()`**
- **`fork()`** liefert als return-Wert:

  - `0` im child-Prozess
  - **`child-PID`** im parent-Prozess
  - `-1` im Fehlerfall und **`errno`** wird gesetzt

fork Beispiel

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    printf("Hier ist der parent.\n");
    fflush(stdout);

    int ergFork = fork();

    printf( "Hier sind wir beide.\n");
    fflush(stdout);

    if(ergFork == 0)
    {
        printf("Hier ist der child.\n");
        fflush(stdout);
    }

    printf("Hier sind schon wieder beide.\n");
    fflush(stdout);

    wait(NULL); // auf Terminierung eines beliebigen childs warten
    // waitpid(ergFork, NULL, 0);
    return 0;
}
```

Ausgabe:

```
Hier ist der parent.
Hier sind wir beide.
Hier sind schon wieder beide.
Hier sind wir beide.
Hier ist der child.
Hier sind schon wieder beide.
```

Reihenfolge der Ausgaben = f(scheduling)

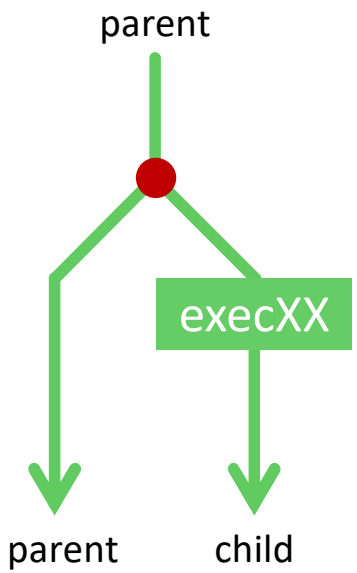
alternativ auf bestimmten child-Prozess warten

zombies und **orphans:**

Korrekterweise müssen parents auf die Terminierung aller childs warten. Sonst:

- Terminiert ein child, ohne dass der parent auf ihn gewartet hat wird er zu einem sog. **zombie**.
  - Zombies verbleiben zunächst im System und belegen die Prozesstabelle - der parent könnte immer noch auf sie warten.
  - Ständig erzeugte zombies können die Prozesstabelle so auffüllen, dass keine weiteren Prozesse erzeugt werden können.
- Terminiert der parent vor seinen childs, werden sie zu sog. **orphans** (Waisen).
  - Orphans werden vom Systemprozess init (PID 1) adoptiert.
  - init terminiert nicht, es entstehen keine „orphan-orphans“.
  - init wartet periodisch auf orphans, um sie ordnungsgemäß aus dem System zu entfernen.

3) Prozesserzeugung mit `fork()` und `execXX()`



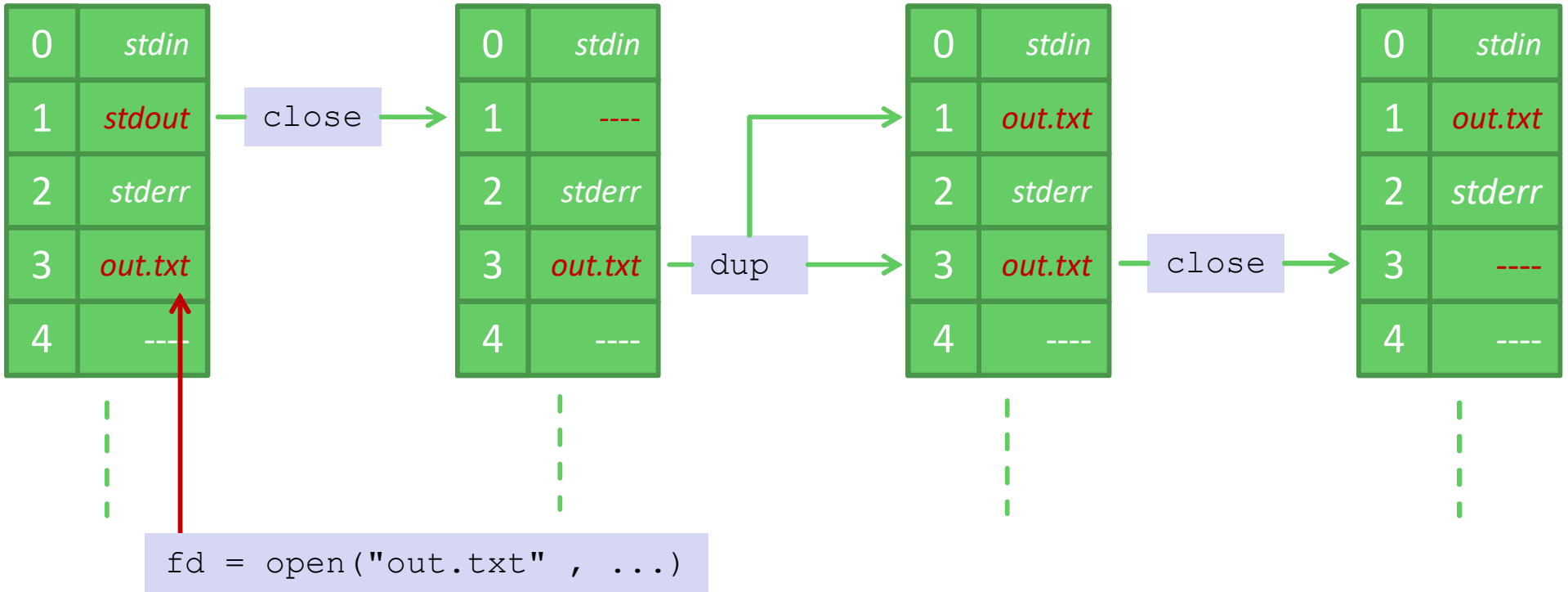
Auch der parent könnte `execXX` ausführen.

- Die **`execXX()`**-Funktionen laden ein neues Programm,
  → das process-image wird vollständig **überschrieben**.
- Die **`execXX()`**-Funktionen kehren nur im Fehlerfall zurück (Ergebnis ist -1 und **`errno`** wird gesetzt).
- Es kann ein binäres image (*executable*) oder ein Interpreter-file geladen werden.
- Die eigentliche exec-Funktion lautet: **execve(…)**
- Die execXX-Varianten unterscheiden sich nur bzgl. der Parameterlisten:
  - **`execl`**, **`execlp`**, **`execle`**: die Argumente des Prozesses werden als Auflistung von **`n`** strings übergeben. Die Liste muss mit einem **`(char*)NULL`**-Zeiger beendet werden.
  - **`execv`**, **`execvp`**, **`execve`**: die Argumente des Prozesses werden als Array von **`n`** strings übergeben. Das letzte Array-Element ist NULL.

Hinweis: **`fork/execXX`**-Konstrukte können teilweise durch **`posix_spawnX()`** ersetzt werden.

IO-Umleitungen mit `dup()`

- File-Deskriptoren sind Indices der sog. File-Descriptor-Tabelle.
- Programme können die Einträge der Tabelle ändern und so IO-Umleitungen erstellen.
- Die Funktion **`dup()`** dupliziert einen Tabelleneintrag.
  ➔ Dabei wird der Platz mit dem niedrigsten freien Index belegt.



Beispiel

Umleitung `stdout` → Datei

```
int flags = O_WRONLY | O_CREAT | O_TRUNC;
mode_t permission = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

int fdOut = open("dup.txt", flags, permission);

close(STDOUT_FILENO);

int fdNew = dup(fdOut);

close(fdOut);

printf("Moin dup\n");// <-- schreibt in Datei dup.txt

FILE* fStreamNew = fdopen(fdNew, "w+");
fflush(fStreamNew);

close(fdNew);
```

Etwas unsichere Kurzversion:

```
int fdOut = open("dup.txt", flags, permission);
close(STDOUT_FILENO);
dup(fdOut);
printf("Moin dup\n");
fflush(stdout);
```