

# System-Programmierung

## 5: Prozess-Lebenszyklus

CC BY-SA, Thomas Amberg, FHNW  
(Soweit nicht anders vermerkt)

Slides: [tmb.gr/syspr-5](https://tmb.gr/syspr-5)

# Überblick

Diese Lektion behandelt den *Prozess Lebenszyklus*.

Wie ein Child-Prozess aus einem Parent entsteht.

Ausführen von, und warten auf neue Programme.

Was im Speicher und mit offenen Files geschieht.

# Prozess-Lebenszyklus

Mit *fork()* erstellt ein Prozess einen Child-Prozess:

```
pid_t fork(void); // PID bzw. 0, od. -1, errno
```

Mit *exit()* beendet sich ein (Child-)Prozess selbst:

```
void exit(int status); // status & 0377
```

Mit *wait()* wartet ein Parent auf Child-Prozesse:

```
pid_t wait(int *status); // PID od. -1, errno
```

*execve()* führt ein Programm aus: `int execve(...);`

# Prozess kreieren mit *fork()*

Der *fork()* System Call erlaubt einem Prozess (*Parent*) einen neuen Prozess (*Child*) zu erzeugen. Dazu wird eine fast exakte Kopie des Parent-Prozesses gemacht:

```
pid_t fork(void); // Child PID bzw. 0, oder -1
```

Der Child-Prozess bekommt Kopien der Text-, Daten-, Heap- und Stack-Segmente des Parent-Prozesses.

Ein *fork()* führt zu einer Gabelung, Verzweigung.

# Prozess beenden mit *exit()*

Die Library Funktion *exit()* beendet einen Prozess und gibt dessen Ressourcen — Speicher, File Deskriptoren, ... — frei. Der Status kann mit *wait()* gelesen werden.

```
void exit(int status); // Child terminiert sich
```

Der C Standard definiert Konstanten für *status* Werte:

```
#define EXIT_SUCCESS 0 // siehe stdlib.h
```

```
#define EXIT_FAILURE -1 // bzw. != 0
```

# Auf Child-Prozess warten mit *wait()*

Der *wait()* System Call suspendiert den Prozess, bis einer seiner Child-Prozesse *exit()* aufruft, und gibt die Child-Prozess PID sowie den Status von *exit()* zurück:

```
pid_t wait(int *status); // PID oder -1, errno
```

Auf Child-Prozesse warten, bis keiner mehr übrig ist:

```
while(wait(NULL) != -1) {} // -1 und ECHILD  
if (errno != ECHILD) { ... } // andere Fehler
```

# Ablauf aus Prozess Sicht

A0: ...

A1: `int pid = fork();`

A2: `if (pid == 0) {`

A3: ...

A4: `exit(0);`

A5: `} else {`

A6: ... `// parent`

A7: `wait(&status);`

A8: `} // status = 0`

A1: `int pid = fork();`

A2: `if (pid == 0) {`

A3: ... `// child`

A4: `exit(0);`

# Hands-on, 15': *fork()*

*fork.!c*

Schreiben Sie ein Programm *my\_fork.c*, das "forkt".  
Nutzen Sie die System Calls *fork()*, *exit()* und *wait()*.

Das Programm soll den folgenden Output ausgeben,  
mit konkreten Prozess IDs für *pid*, *pid\_c* und *pid\_p*:

I'm parent *pid* of child *pid\_c*

I'm child *pid* of parent *pid\_p*



# Programm ausführen mit *exec()*

Der *exec()* System Call\* lädt ein neues Programm in den Speicher des Prozesses. Dieser Call kommt nicht zurück. Der vorherige Programmtext wird verworfen. Daten-, Heap- & Stack-Segmente werden neu erstellt:

```
int execve(const char *filepath, // -1, errno  
    char *const argv[], // letztes Element = NULL  
    char *const envp[]); // letztes Elem. = NULL
```

\*Es gibt Varianten von *exec()*, z.B. *execve()*, *execlp()*. 9

# Ablauf aus Prozess Sicht

A0: ... // program A

A1: int pid = fork();

A2: if (pid == 0) {

A3: ...

A4: execve("./B", ...);

A5: } else {

A6: ... // parent

A7: wait(&status);

A8: } // status = 0

A1: int pid = fork();

A2: if (pid == 0) {

A3: ... // child

A4: execve("./B", ...);

---

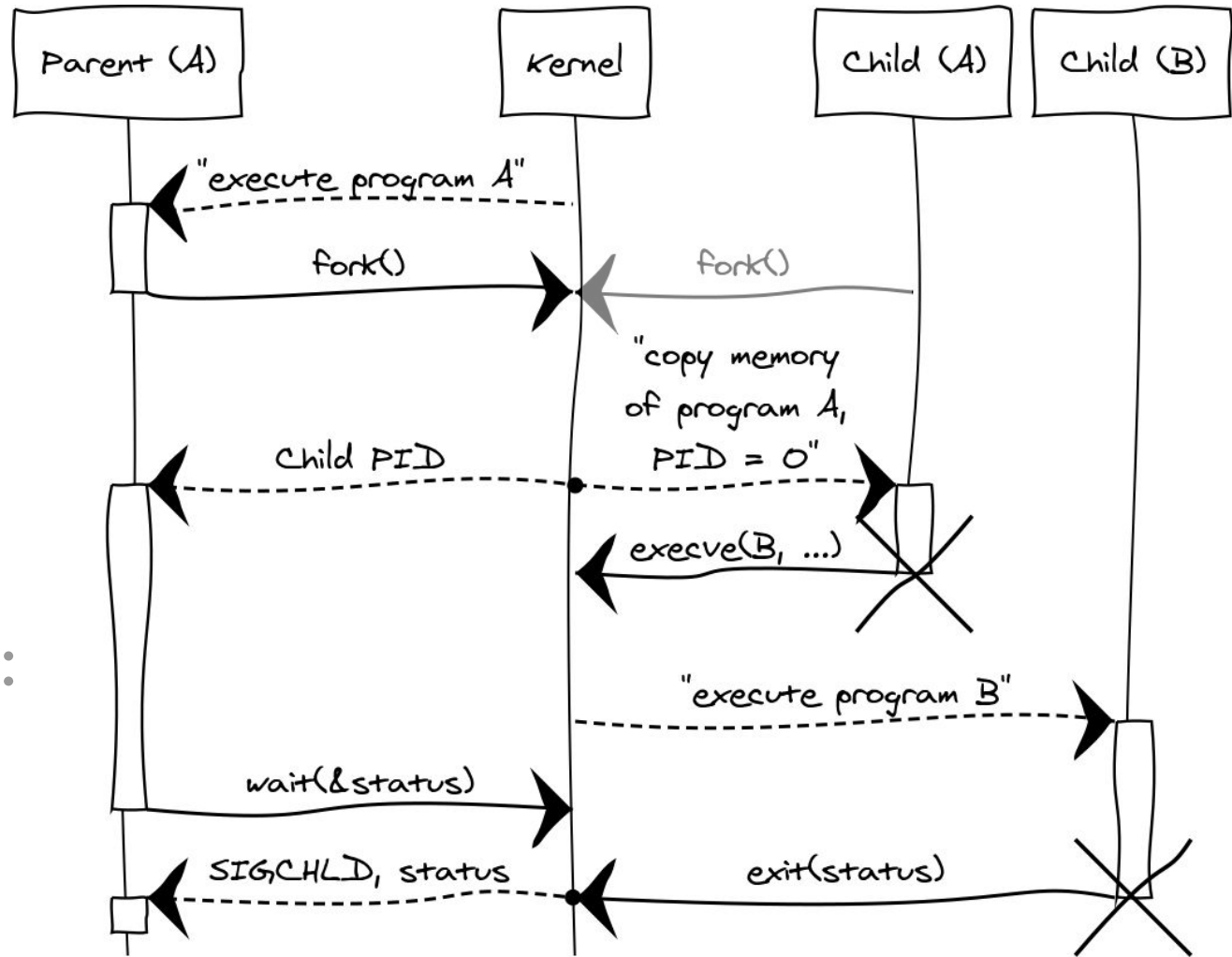
B0: ... // child, prog. B

B1: exit(0);

# Ablauf

*fork(),  
exec(),  
wait(),  
exit().*

Alternative:  
*Child<sub>A</sub> ruft  
exit() auf.*



# Shell Kommando ausführen mit *system()*

Die *system()* Funktion kreiert einen Child-Prozess der Shell Kommandos einfach und bequem ausführt:

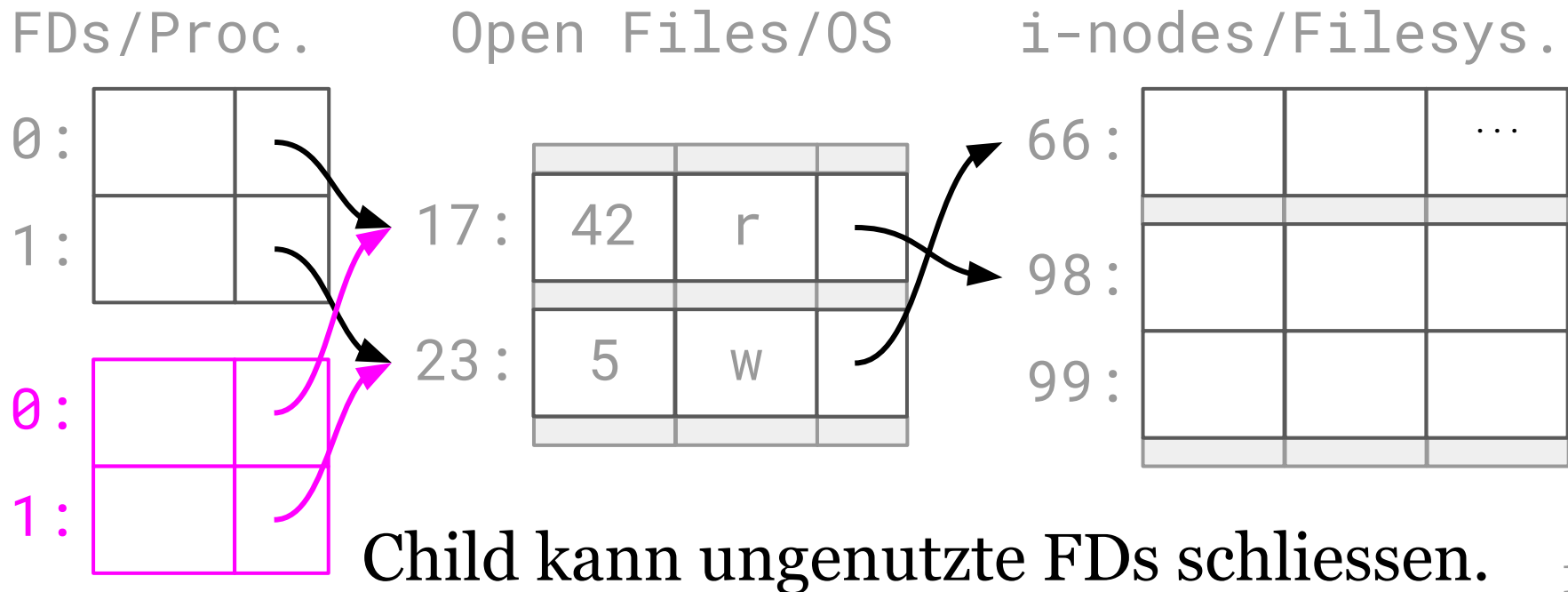
```
int system(const char *cmd); // z.B. "ls | wc"
```

Versteckt Details von *fork()*, *exec()*, *wait()* und *exit()*.

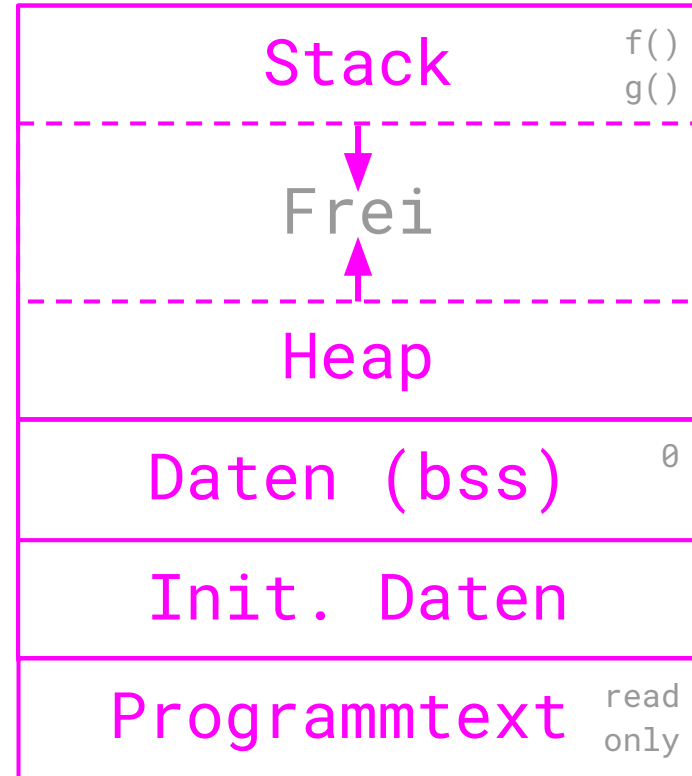
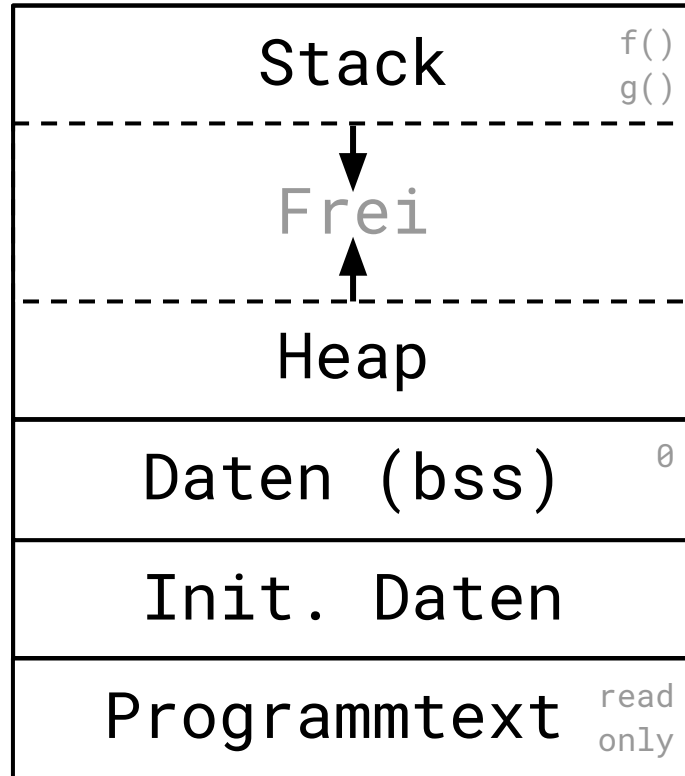
Fehler- und Signal-Handling werden übernommen.

# File Deskriptoren `fork_file_sharing.c`<sup>TLPI</sup>

File Deskriptoren werden bei *fork()* mit *dup()* kopiert:



# Speicher Layout nach *fork()*



# Speicher Semantik von *fork()*

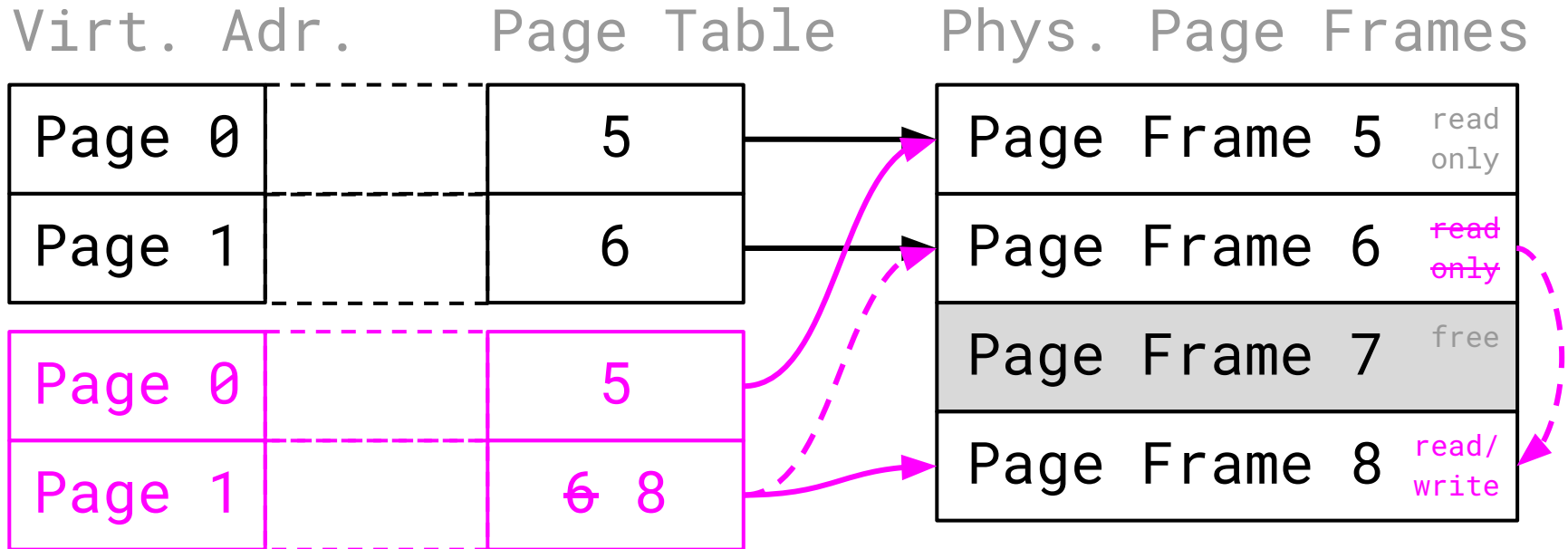
Virtuellen Speicher kopieren wäre verschwenderisch, denn auf einen *fork()* System Call folgt oft ein *exec()*.

Die Programmtext-Segmente von Parent und Child zeigen auf eine physische Page, die *read-only* ist.

Für Daten-, Heap- und Stack-Segmente des Parents verwendet der Kernel *copy-on-write* Semantik: Erst sind Pages *read-only*, ein Trap bei *write* kopiert sie.

# Prozess Page Table *copy-on-write*

Bei *copy-on-write* wird erst beim *write()* kopiert, die physischen Page Frames werden dann schreibbar:





# Funktion in *fork()* wrappen **footprint.c**<sup>TLPI</sup>

Wenn *f()* Speicher verliert, oder Heap fragmentiert:

```
int pid = fork(); // Child Start, Heap kopiert
if (pid == 0) {
    int status = f(); // problematische Funktion
    exit(status);
} // Child Ende, Ressourcen werden freigegeben
wait(&status); // Wartet auf exit() des Childs
if (status == -1) { ... } // Resultat von f()
```

# Race Conditions `fork_whos_on_first.c`<sup>TLPI</sup>

Nach *fork()* können Parent oder Child zuerst laufen\* und auf Mehrprozessorsystemen auch beide parallel:

```
$ ./fork_whos_on_first 10000 > fork.txt
```

```
$ ./fork_whos_on_first.count.awk fork.txt
```

Robuster Code muss mit jeder Reihenfolge zurecht kommen, um Race Conditions auszuschliessen.

\*[Hier](#) eine Email von Linus Torvalds zum Thema.

# Synchronisation

`fork_sig_sync.c`<sup>TLPI</sup>

Signale helfen, Race Conditions zu verhindern, wenn einer der Prozesse auf den anderen warten muss.

Hier wird *SIGUSR1* verschickt, vom Child zum Parent, der *sigaction()* Call setzt einen *handler*, wie *signal()*.

Weil *SIGUSR1* geblockt wurde, bleibt es *pending*, mit *sigsuspend()* wird das Signal *SIGUSR1* entblockt und atomar auf Signale gewartet, wie bei *pause()*.

# Prozess beenden mit `_exit()`

Ein Prozess terminiert *abnormal*, durch ein Signal, oder *normal*, durch Aufruf des `_exit()` System Calls:

```
void _exit(int status);
```

Das *status* Argument kann via `wait()` gelesen werden, wobei nur die unteren 8 Bits des *int* verfügbar sind.

Ein *status* Wert `!= 0` bedeutet, es gab einen Fehler. Meistens wird der `exit()` Library Call verwendet.

# Prozess beenden mit *exit()*

Der *exit()* Library Call macht mehr, als nur *\_exit()*:

```
void exit(int status);
```

Exit Handler, registriert mit *atexit()* und *on\_exit()*, werden in umgekehrter Reihenfolge aufgerufen.

Die *stdio* Stream Buffer werden mit *fflush()* geleert.

Der *\_exit()* System Call wird mit *status* aufgerufen.

# Prozess beenden in *main()*

Ein Prozess kann auch am Ende von *main()* enden:

Explizit, durch *return n* was äquivalent ist zu *exit(n)*, weil die run-time Funktion den Wert in *exit()* steckt.

Oder implizit, indem das Programm unten rausfällt.

Das Resultat ist in C89 undefiniert, in C99 *exit(0)*.

# Prozess Lebensdauer

Parent- und Child-Prozess leben oft verschieden lang:

"Verwaiste" Child-Prozesse bekommen *init* als Parent.

Oder ein Parent ruft *wait()* auf, um den Terminations-Status zu lesen, obwohl der Child-Prozess zu Ende ist.

Der Kernel bewahrt solche, bereits terminierten, aber noch nicht mit *wait()* erwarteten *Zombie*-Prozesse auf.

# Zombie-Prozesse

Der Kernel führt für Zombie-Prozesse eine Liste mit PID, Terminations-Status, und Ressourcen-Statistik. Zombies können mit keinem Signal beendet werden.

Wenn der Parent *wait()* noch aufruft, gibt der Kernel den Status zurück und entfernt den Zombie-Prozess.

Falls der Parent-Prozess *wait()* nicht aufruft, verwaist der Zombie, und der *init*-Prozess ruft *wait()* auf.



# Hands-on, 15': Zombie-Prozesse `zombie.c`

Schreiben Sie Code, der für 1 Sekunde einen Zombie-Prozess erzeugt, mit `exit()`, `fork()`, `sleep()` und `wait()`.

```
$ ./my_zombie &  
[1] 1001  
$ ps aux | grep my_zombie  
... 1001 ... ./my_zombie  
... 1002 ... [my_zombie] <defunct>
```

Hinweis: *<defunct>* bedeutet Zombie-Prozess.

# Das *SIGCHLD* Signal

Immer wenn ein Child-Prozess terminiert, wird das *SIGCHLD* Signal zum Parent-Prozess gesendet.

Ein Handler kann dann *wait()* rechtzeitig aufrufen:

```
int result = signal(SIGCHLD, handle);  
void handle(int sig) { int pid = wait(NULL); }  
// für > 1 Child, wait() in Loop bis -1, ECHILD
```

Explizites Ignorieren des Signals verhindert Zombies:

```
int result = signal(SIGCHLD, SIG_IGN);
```

# Selbststudium: Threads

Als Vorbereitung auf die nächste Lektion, lesen Sie <https://hpc-tutorials.llnl.gov/posix/> bis *Pthread Exercise 1*.

# Feedback oder Fragen?

Gerne in Teams, oder per Email an

[thomas.amberg@fhnw.ch](mailto:thomas.amberg@fhnw.ch)

Danke für Ihre Zeit.