

# Technische Informatik 3 – Embedded Systems

## Kapitel 5: Linux - Prozesse

Prof. Dr. Benjamin Kormann

Fakultät für Elektro- und Informationstechnik

22.05.2023



# Das Konzept des Funktionszeigers

## Zeiger auf Funktionen

### Syntax des Funktionszeigers anhand eines Beispiels

```
int (*func_ptr) (int);
```

Diagramm zur Syntax des Funktionszeigers:

- Rückgabewert:** `int`
- Name des Funktionszeigers:** `func_ptr`
- Parameter der Funktion:** `int`

### Verwendungszweck von Funktionszeigern

- Übergabe von aufzurufenden Funktionen als Funktionsparameter
  - Vergleiche lambda Funktionen als Übergabe an höhere Funktionen
  - Möglichkeit für sog. Callback Funktionen
- Generalisierung von Aufrufen von Funktionen identischer Signatur
  - Durchführung verschiedener Berechnungen auf identischen Daten

```
#include <stdio.h>

int add(int a, int b)
{
    return a+b;
}

int sub(int a, int b)
{
    return a-b;
}

int main()
{
    // Definition eines Funktionszeigers
    int (*op)(int, int);

    // Anwendung des Funktionszeigers
    op = &add;
    printf("%d ", op(5, 3));
    op = &sub;
    printf("%d ", op(5, 3));

    printf("\n");

    // Funktionszeiger in einem Array
    int (*op_arr[])(int, int) = {&sub, &add};
    for(int i = 0; i < 2; i++)
    {
        printf("%d ", (*op_arr[i])(7, 3));
    }

    return 0;
}
```

Diagramm zur Verwendung von Funktionszeigern:

- Deklaration:** `int (*op)(int, int);`
- Zuweisung:** `op = &add;`
- Anwendung:** `op(5, 3);`
- Deklaration:** `int (*op_arr[])(int, int) = {&sub, &add};`
- Initialisierung:** `{&sub, &add};`
- Anwendung:** `(*op_arr[i])(7, 3);`

# Linux Prozesse - Prozessinformationen

## Rekapitulation zum Beenden eines Linux Prozesses

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    // Abfrage der Prozess Identifikation
    pid_t pid = getpid();
    printf("PID: %d\n", pid);

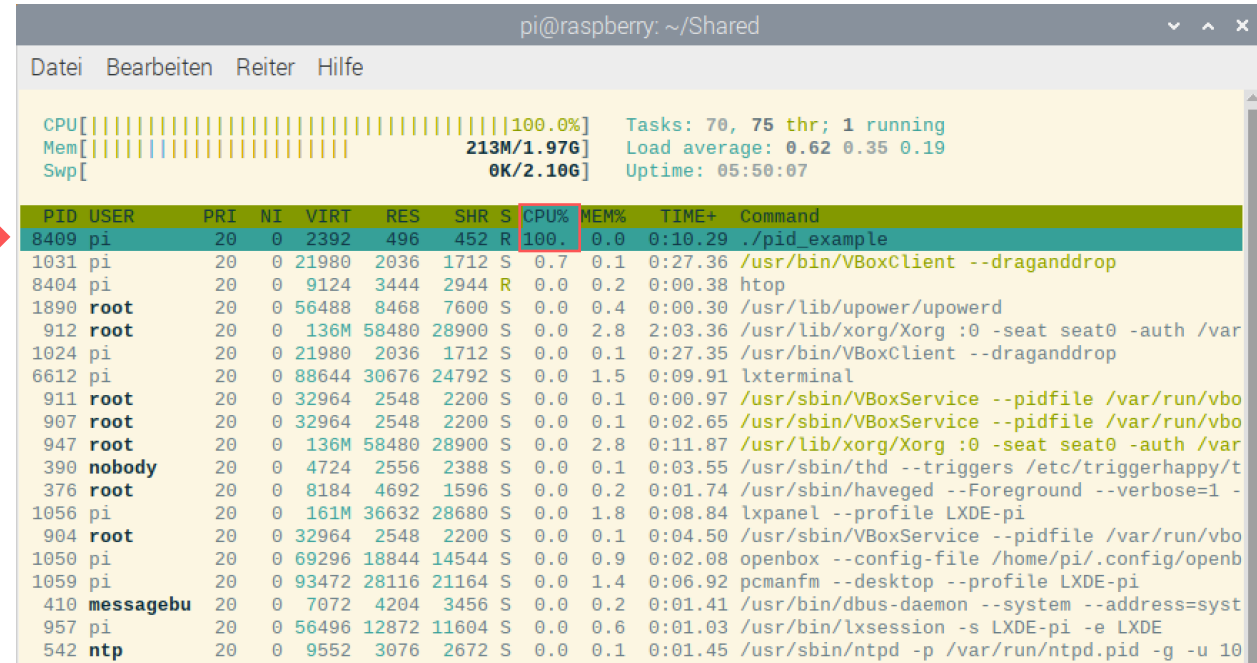
    // Endlosschleife
    while(1)
    {
        ;
    }

    // Programmende
    return 0;
}
```

Konsole

```
$ gcc pid-example.c -o pid_example
$ ./pid_example
PID: 8409
Beendet
```

### Ausschnitt aus htop



pi@raspberrypi: ~/Shared

Datei Bearbeiten Reiter Hilfe

CPU[|||||||||||||||||||||||||||||||||||||||||100.0%] Tasks: 70, 75 thr; 1 running  
Mem[|||||||||||||||||||||||||||||||||213M/1.97G] Load average: 0.62 0.35 0.19  
Swp[|||||0K/2.10G] Uptime: 05:50:07

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
8409	pi	20	0	2392	496	452	R	100.	0.0	0:10.29	./pid_example
1031	pi	20	0	21980	2036	1712	S	0.7	0.1	0:27.36	/usr/bin/VBoxClient --draganddrop
8404	pi	20	0	9124	3444	2944	R	0.0	0.2	0:00.38	htop
1890	root	20	0	56488	8468	7600	S	0.0	0.4	0:00.30	/usr/lib/upower/upowerd
912	root	20	0	136M	58480	28900	S	0.0	2.8	2:03.36	/usr/lib/xorg/Xorg :0 -seat seat0 -auth /var
1024	pi	20	0	21980	2036	1712	S	0.0	0.1	0:27.35	/usr/bin/VBoxClient --draganddrop
6612	pi	20	0	88644	30676	24792	S	0.0	1.5	0:09.91	lxterminal
911	root	20	0	32964	2548	2200	S	0.0	0.1	0:00.97	/usr/sbin/VBoxService --pidfile /var/run/vbo
907	root	20	0	32964	2548	2200	S	0.0	0.1	0:02.65	/usr/sbin/VBoxService --pidfile /var/run/vbo
947	root	20	0	136M	58480	28900	S	0.0	2.8	0:11.87	/usr/lib/xorg/Xorg :0 -seat seat0 -auth /var
390	nobody	20	0	4724	2556	2388	S	0.0	0.1	0:03.55	/usr/sbin/thd --triggers /etc/triggerhappy/t
376	root	20	0	8184	4692	1596	S	0.0	0.2	0:01.74	/usr/sbin/haveged --Foreground --verbose=1 -
1056	pi	20	0	161M	36632	28680	S	0.0	1.8	0:08.84	lxpanel --profile LXDE-pi
904	root	20	0	32964	2548	2200	S	0.0	0.1	0:04.50	/usr/sbin/VBoxService --pidfile /var/run/vbo
1050	pi	20	0	69296	18844	14544	S	0.0	0.9	0:02.08	openbox --config-file /home/pi/.config/openb
1059	pi	20	0	93472	28116	21164	S	0.0	1.4	0:06.92	pcmanfm --desktop --profile LXDE-pi
410	messagebu	20	0	7072	4204	3456	S	0.0	0.2	0:01.41	/usr/bin/dbus-daemon --system --address=syst
957	pi	20	0	56496	12872	11604	S	0.0	0.6	0:01.03	/usr/bin/lxsession -s LXDE-pi -e LXDE
542	ntp	20	0	9552	3076	2672	S	0.0	0.1	0:01.45	/usr/sbin/ntpd -p /var/run/ntpd.pid -g -u 10

Konsole

Signal  
SIGTERM

```
$ kill 8409
$
```

Beendet den Prozess  
pid\_example (PI: 8409)

# Linux Prozesse - Signalkonzept

## Einrichten eines Signal Handlers

Linux Kernel mitteilen, was beim Empfang eines Signal geschehen soll

```
#include <signal.h>

sig_t signal(int sig, sig_t func);
```

└──────────> typedef void (\*sig\_t) (int);

### Erläuterung der Parameter

- `sig`: legt das Signal fest, auf das bei Empfang reagiert werden soll
- `func`: Funktionszeiger zu einem Signal Handler (ähnliche Idee wie bei Interrupt Service Routinen)

### Reagieren auf empfangene Signale

- Auf die Signale `SIGKILL` und `SIGSTOP` kann nicht reagiert werden. Das Betriebssystem kann damit einen Prozess “ohne Rückfrage” beenden bzw. anhalten
- Das Standardsignal beim Aufruf des Konsolenkommandos `kill` ist `SIGTERM`

# Linux Prozesse - Signalkonzept

## Übersicht der Linux Signale

ID	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	<b>SIGKILL</b>	terminate process	kill program (cannot be caught / ignored)
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	<b>SIGTERM</b>	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket

ID	Name	Default Action	Description
17	SIGSTOP	stop process	stop (cannot be caught / ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from ctrl.
22	SIGTTOU	stop process	background write attempted to ctrl.
23	SIGIO	discard signal	I/O is possible on a descriptor
24	SIGXCPU	terminate process	cpu time limited exceeded
25	SIGXFSZ	terminate process	file size limited exceeded
26	SIGVTALRM	terminate process	virtual time alarm
27	SIGPROF	terminate process	profiling timer alarm
28	SIGWINCH	discard signal	window size changed
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	user defined signal 1
31	SIGUSR2	terminate process	user defined signal 2

# Linux Prozesse - Signalkonzept

## Abfangen eines Signals

```
$ gcc signal_test.c -o signal_test
$ ./signal_test
PID: 98058
```

```
Signal 15 empfangen...
Programm beendet sich trotzdem nicht.
```

```
Signal 15 empfangen...
Programm beendet sich trotzdem nicht.
```

```
$ zsh: killed ./signal_test
$
```

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

// Signal Handler für SIGTERM
void ignore(int sig)
{
    printf("Signal %d empfangen...\n", sig);
    printf("Programm beendet sich trotzdem nicht.\n");
    fflush(stdout);
}

int main()
{
    // PID abfragen
    pid_t pid = getpid();
    printf("PID: %d\n", pid);

    // Signal Handler registrieren
    signal(SIGTERM, ignore);

    // Endlosschleife
    while(1)
    {
        ;
    }

    // Programmende wird so nie erreicht
    return 0;
}
```

```
$ kill 98058
$ kill -TERM 98058
$ kill -KILL 98058
$
```

# Linux Prozesse - Prozesserstellung

## Erzeugen eines neuen Prozesses

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    // Erzeugen eines neuen Prozesses
    fork();

    // Abfrage der Prozess Identifikation
    pid_t pid = getpid();
    printf("PID: %d\n", pid);

    // Programmende
    return 0;
}
```

↓ Konsole

```
PID: 65759
PID: 65760
```

### Beschreibung der Funktion

- `pid_t fork()`
  - Erzeugt einen neuen Prozess durch Duplikation des aufrufenden Prozesses
  - Der neue Prozess heißt `child` Prozess
  - Der aufrufende Prozess heißt `parent` Prozess
  - Die beiden Prozesse laufen in verschiedenen Speicherbereichen und besitzen nach dem Aufruf von `fork()` den identischen Inhalt
- Rückgabewerte
  - `>0` := PID des `child` Prozesses für den `parent` Prozess
  - `0` := Rückgabewert für den `child` Prozess
  - `-1` := Fehler im `parent` Prozess und kein `child` Prozess erzeugt

# Linux Prozesse - Prozesserzeugung

## Entwicklung der Prozesse bei Anwendung von `fork()`

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    // Erzeugen neuer Prozesse
    fork();
    fork();
    fork();

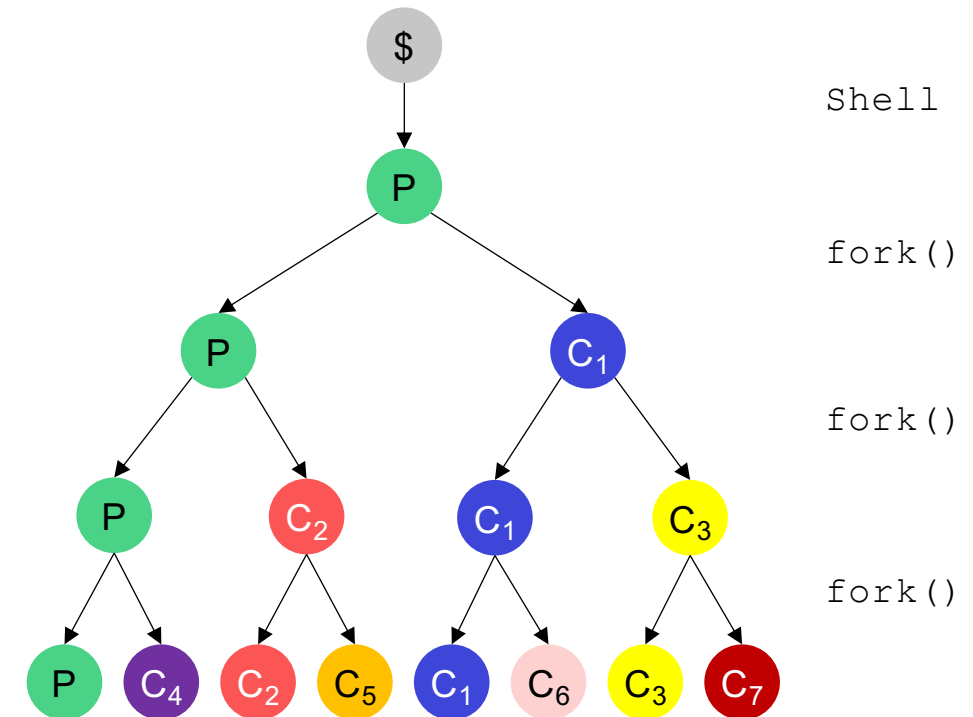
    // Abfrage der Prozess Identifikation
    pid_t pid = getpid();
    printf("PID: %d\n", pid);

    // Programmende
    return 0;
}
```

Konsole

```
$ ./fork-example
PID: 72401
PID: 72404
PID: 72403
PID: 72402
PID: 72407
PID: 72406
PID: 72405
PID: 72408
$
```

### Entwicklung der Prozesse





# Linux Prozesse - Prozesserstellung

## Typischer Ablauf des Prozess-Handlings mit `fork()`

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

int main()
{
    // Prozess-ID zur Fallunterscheidung
    pid_t pid = 0;

    // Fehler bei der Prozesserstellung
    if ( (pid = fork()) < 0 )
    {
        fprintf(stderr, "error creating process");
        return 0;
    }

    // Neuer Prozess (child)
    else if ( pid == 0 )
    {
        printf("Child: PID(%d), PPID(%d)\n", getpid(), getppid());
        exit(0); // Prozess beenden
    }

    // Aktueller Prozess (parent)
    else
    {
        printf("Parent: PID(%d), PPID(%d)\n", getpid(), getppid());
        int status;
        // Warten auf Ende Child-Prozesses (Zombie Gefahr)
        while(wait(&status)!=pid);
    }

    // Programmende
    return 0;
}
```

- Falls `fork()` fehlschlägt wird -1 zurückgegeben und es wird kein Prozess erzeugt.
- Auch wenn dieser Fall selten eintreten kann, sollte dieser immer geprüft werden.
- Der erzeugte `child` Prozess erhält von `fork()` den Rückgabewert 0.
- Der Prozess kann mit `exit()` beendet werden.
- Der `parent` Prozess erhält als Rückgabe von `fork()` die Prozess-ID des erzeugten `child` Prozesses.
- Die Funktion `wait()` unterbricht die Ausführung bis einer der `child` Prozesse beendet wird.
- Der Rückgabewert von `wait()` entspricht der PID des `child` Prozesses, der beendet wurde.

Konsole



```
$ echo $$
64088
$ ./fork-template
Parent: PID(72827), PPID(64088)
Child: PID(72828), PPID(72827)
```

# Linux Prozesse - Prozesserstellung

## Typische Anwendungen von `fork()`

Die Prozesserstellung mit `fork()` findet meist in den folgenden beiden Fällen eine Anwendung

- 1) Ein Programm soll zu einem Zeitpunkt zwei verschiedene Codeteile ausführen
  - Server in Netzwerkkommunikation kann Anfragen von Clients entgegennehmen
  - Zur Bearbeitung einer Client Anfrage muss dieser in einem separaten Prozess bearbeitet werden, da sonst der Server für die Dauer der Bearbeitung der Client Anfrage nicht mehr reagiert
- 2) Ein Prozess möchte ein weiteres Programm (Prozess) ausführen. Dazu ruft der Kind Prozess nach der Erstellung (`fork()`) die sog. `exec()` Funktion auf
  - Neuer Prozess übernimmt die Prozess-ID des Kindprozesses
  - Die vier Speichersegmente werden übernommen

```
#include <stdio.h>
#include <unistd.h> // Header für exec()
#include <sys/wait.h>

int main()
{
    // Prozess erzeugen
    pid_t pid;
    if ( (pid = fork()) < 0 ) // Fehlerfall
    {
        perror("process couldn't be created\n");
        return 1;
    }
    else if ( pid == 0 ) // Kindprozess
    {
        printf("Child: %d", getpid());
        fflush(stdout);
        // Externes Programm starten
        char* args[4] = { "ls", "-l", "/usr/bin", "NULL"};
        execv("/bin/ls", args);
        // die nun folgenden Zeilen werden nicht mehr ausgeführt
        printf("Child: %d", getpid());
        fflush(stdout);
    }
    else // Elternprozess
    {
        // auf das Ende des Kindprozesses warten
        int status;
        while(wait(&status) != pid);
    }

    // Programmende
    return 0;
}
```

**Kindprozess puffert Ausgabe bis zum Prozessende**

**Wird nicht mehr ausgeführt, da `execv` anschließend den Prozess beendet**

# Dokumentationssystem unter Linux

## Manpages

**Linux hat ein etabliertes Dokumentationssystem (Handbuch) für Kommandos etc.**

- Aufruf der Manpage erfolgt in der Konsole mit Hilfe des Kommandos `man`
  - Beispiel: `$ man ls`
- Die Manpages sind in verschiedene Sections unterteilt
  - 1) General commands
  - 2) System calls
  - 3) Library functions, covering in particular the C standard library
  - 4) Special files (usually devices, those found in `/dev`) and drivers
  - 5) file formats and conventions
  - 6) Games and screensavers
  - 7) Miscellaneous
  - 8) System administration commands and daemons

# Dokumentationssystem unter Linux

## Manpage für exec (man 3 exec)

1/X

```
EXEC(3)                                Library Functions Manual                                EXEC(3)

NAME
    execl, execl, execlp, execv, execvp, execvp - execute a file

LIBRARY
    Standard C Library (libc, -lc)

SYNOPSIS
    #include <unistd.h>

    extern char **environ;

    int
    execl(const char *path, const char *arg0, ..., /* (char *)0, */);

    int
    execl(const char *path, const char *arg0, ..., /* (char *)0 char *const envp[] */);

    int
    execlp(const char *file, const char *arg0, ..., /* (char *)0, */);

    int
    execv(const char *path, char *const argv[]);

    int
    execvp(const char *file, char *const argv[]);

    int
    execvp(const char *file, const char *search_path, char *const argv[]);
```

# Dokumentationssystem unter Linux

## Manpage für exec (man 3 exec)

2/X

### DESCRIPTION

The exec family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function `execve(2)`. (See the manual page for `execve(2)` for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The `const char *arg0` and subsequent ellipses in the `execl()`, `execvp()`, and `execvP()` functions can be thought of as `arg0`, `arg1`, ..., `argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a NULL pointer.

The `execv()`, `execvp()`, and `execvP()` functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers must be terminated by a NULL pointer.

The `execle()` function also specifies the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the `argv` array with an additional argument. This additional argument is an array of pointers to null-terminated strings and must be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable `environ` in the current process.

Some of these functions have special semantics.

The functions `execlp()`, `execvp()`, and `execvP()` will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash `"/"` character. For `execlp()` and `execvp()`, search path is the path specified in the environment by `"PATH"` variable. If this variable is not specified, the default path is set according to the `_PATH_DEFPATH` definition in `<paths.h>`, which is set to `"/usr/bin:/bin"`. For `execvP()`, the search path is specified as an argument to the function. In addition, certain errors are treated specially.

# Linux Prozesse - Prozesserschöpfung

## Zombie Prozesse

### Definition Zombie Prozess

- Ein bereits (theoretisch) beendeter Prozess, der jedoch weiterhin in der Prozesstabelle des Betriebssystems eingetragen ist
- Es werden weder Rechenkapazität noch Speicherkapazitäten genutzt, jedoch verbleibt der Eintrag in der Prozesstabelle

### Entstehung eines Zombie Prozesses

- Ein gestarteter Prozess wird genau dann zu einem Zombie, wenn der `parent` Prozess nicht auf das Ende des `child` Prozesses gewartet hat
- Beim Ende des `child` Prozesses sendet dieser ein `SIGCHLD` Signal an den `parent` Prozess
- Vermeidung durch Einsatz von `wait()` im `parent`

```
int main()
{
    // Neuer Prozess (child)
    pid_t pid = 0;
    if ( ( pid = fork() ) == 0 )
    {
        printf("stopping child process\n");
        exit(0); // Prozess beenden
    }
    // Aktueller Prozess (parent)
    else if ( pid > 0 )
    {
        printf("parent process running...");
        sleep(60);
        // no wait() call
    }

    // Programmende
    return 0;
}
```

↓ Konsole

```
$ ./zombie
stopping child process
parent process running...
```

```
$ ps aux | grep zombie
pi 8911  0.0  0.0  2392  560 pts/2  S+  17:00   0:00  ./zombie
pi 8912  0.0  0.0    0     0 pts/2  Z+  17:00   0:00  [zombie] <defunct>
```

**<defunct>** identifiziert den Zombie Prozess. Prozess kann nur durch Beenden (oder killen) des parent Prozesses ausgetragen werden.  
\$ pstree -g <zombie-PID> liefert die PID des parent.

# Linux Prozesse - Prozesserzeugung

## Speicherbereiche der Prozesse nach `fork()`

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

int globalVar = 47;
static int staticVar = 11;

int main()
{
    short localVar = 73;
    char* dynLocal = (char*) malloc(16);

    pid_t pid;

    if ( (pid = fork()) > 0 )
    {
        globalVar = localVar = staticVar = 111;
        printf("Parent: %p %p %p %p\n", &globalVar, &staticVar, &localVar, dynLocal);
        printf("Parent: %d %d %d\n", globalVar, localVar, staticVar);
    }
    else if ( pid == 0 )
    {
        sleep(1);
        printf("Child: %p %p %p %p\n", &globalVar, &staticVar, &localVar, dynLocal);
        printf("Child: %d %d %d\n", globalVar, localVar, staticVar);
    }

    // Programmende
    return 0;
}
```

\$ ./fork-memory

Parent:	0x436028	0x43602c	0xbffeeb46	0x10f8160
Parent:	111	111	111	
Child:	0x436028	0x43602c	0xbffeeb46	0x10f8160
Child:	47	73	11	

Unterschiedliche Werte für identische Variablen (unterschiedliche Speicherbereiche).

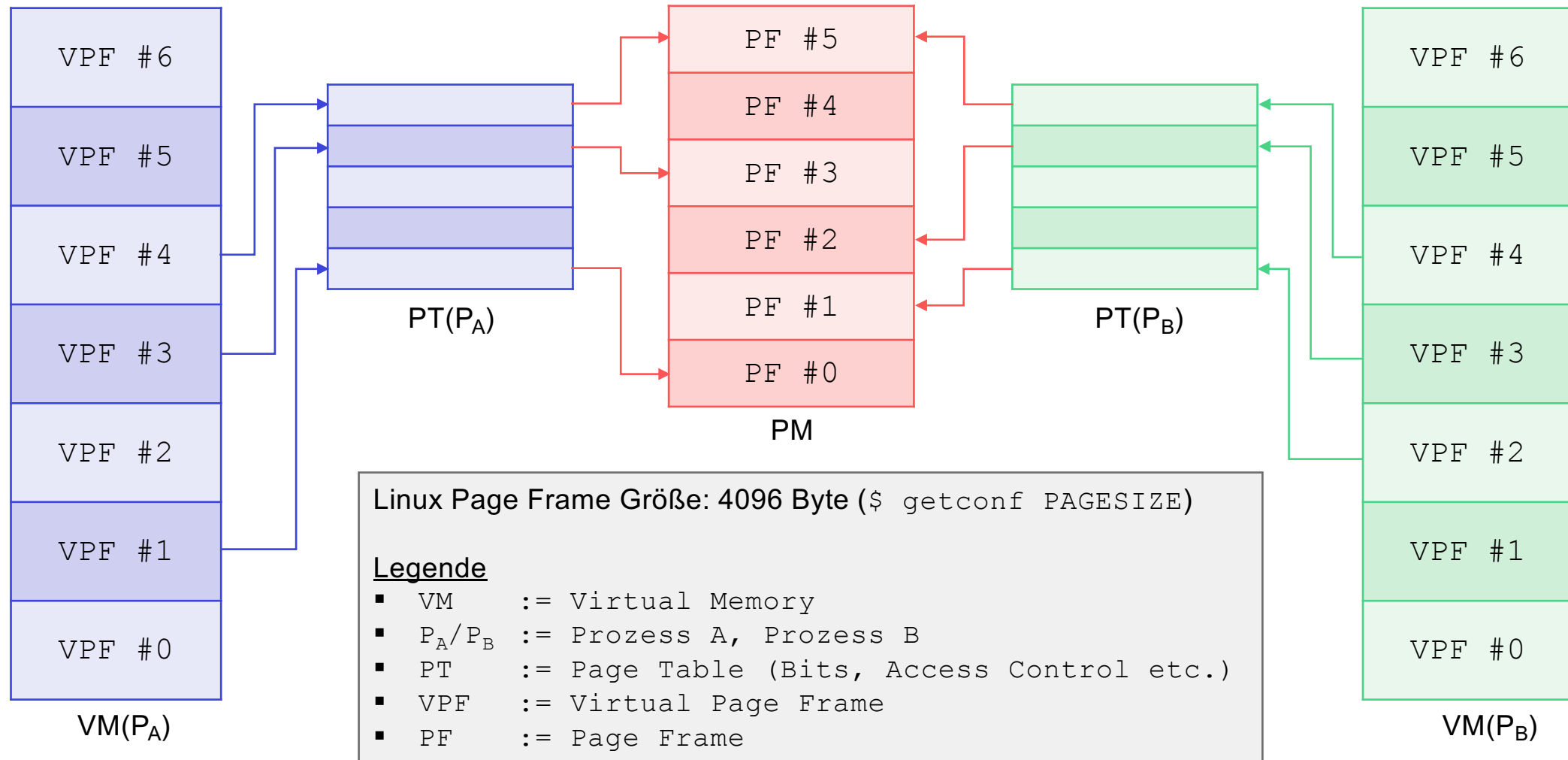
Identische Speicheradressen, obwohl `fork()` Prozesse mit eigenen (unabhängigen) Speicherblöcken erzeugt.

## Virtuelle Speicheradressen

- Linux organisiert den Speicher mit virtuellen Adressen für die ausführenden Prozesse
- Dadurch kann mehr Speicher verwendet bzw. adressiert werden als tatsächlich existiert
- Speicher Management ist Aufgabe des Betriebssystems und erfolgt aus Prozesssicht vollständig transparent
- Virtueller Speicher von Prozessen ist gegenüber anderen Prozessen geschützt

# Linux Prozesse - Prozesserzeugung

## Abstraktes Modell für das Mapping virtuell-physikalisch





# Zusammenfassung

## Funktionszeiger in C

- Übergabe von aufzurufenden Funktionen als Funktionsparameter
- Generalisierung von Aufrufen von Funktionen identischer Signatur
- Anwendbar beim Signalkonzept unter Linux

## Signalkonzept

- Reagieren auf den Empfang von Signalen
- Liste der vorhandenen Signal (auf SIGKILL und SIGSTOP kann nicht reagiert werden)

## Linux Prozesse

- Abfragen und Zusammenhänge von PID und PPID
- Starten und Beenden von Prozessen (fork, exit, kill)
  - Separater Adressraum trotz gleicher Speicheradressen (virtuelles Speichermanagement)
- Besonderheiten beim Aufruf von fork() zur Prozesserzeugung
- Gefahr von Zombie Prozessen