

System-Programmierung

5: Prozess-Lebenszyklus

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



Ablauf heute

⅓ Vorlesung,
⅔ Hands-on,
Feedback.

Slides, Code & Hands-on: tmb.gr/syspr-5



Prozess-Lebenszyklus System Calls

Mit *fork()* erstellt ein Prozess einen neuen Prozess:

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

exit() beendet einen Prozess, gibt Ressourcen frei:

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

wait() wartet auf eine Prozess-Zustandsänderung:

```
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()* ist eine "Verzweigung" in zwei Kopien.



Prozess beenden mit *exit()*

Die Library Funktion *exit()* beendet einen Prozess, und gibt alle Ressourcen (Speicher, File Deskriptoren etc.) frei. Das Status Argument wird dem *wait()* Call übergeben, nachdem der Child-Prozess beendet ist:

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

Der C Standard definiert Konstanten für *status* Werte:

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

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



Zustandsänderung abwarten mit *wait()*

Der *wait()* System Call suspendiert den Prozess, bis einer seiner Child-Prozesse *exit()* aufruft, und gibt den *status* des Child-Prozesses im Argument zurück:

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

Als *return*-Wert liefert *wait()* die Child-Prozess PID:

```
while(wait(NULL) != -1) {} // mehrere abwarten  
if (errno != ECHILD) { ... } // ECHILD => fertig
```



Programm ausführen mit *execve()*

Der *execve()* 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:

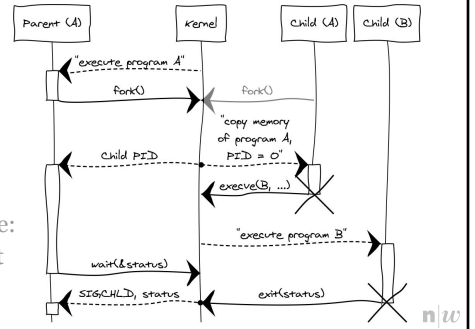
```
int execve(const char *filepath, // -1, errno
char *const argv[], // letztes Element = NULL
char *const envp[]); // letztes Elem. = NULL
```

Es gibt Varianten, allgemein *exec()* Calls genannt. n|w

Ablauf

fork(),
execve(),
wait(),
exit().

Alternative:
Child_A ruft
exit() auf.



Ablauf aus Prozess Sicht

Parent:

```
A0: ... // Programm A
A1: int pid = fork();
A2: if (pid == 0) {
A6: } else { // != 0
A7:   pid_c = pid;
A8:   pid = getpid();
A9:   wait(&status);
A10: } // status = 0
```

Child:

```
A1: int pid = fork();
A2: if (pid == 0) {
A3:   pid = getpid();
A4:   pid_p = getppid();
A5:   execve("./B", ...);
B1: ... // Programm B
B2: exit(0);
```

n|w

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

fork().!c

Schreiben Sie ein Programm *my_fork.c*, das "forkt". Nutzen Sie die online System Call Dokumentation.

Das Programm soll den folgenden Output ausgeben, mit konkreten PID Werten für *pid*, *pid_c* und *pid_p*:
I'm parent *pid* of child *pid_c*
I'm child *pid* of parent *pid_p*

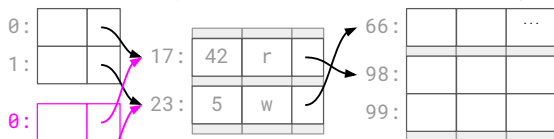
Entspricht der Output ihren Erwartungen? Wieso?

n|w

File Deskriptoren *fork_file_sharing.c*^{TLPI}

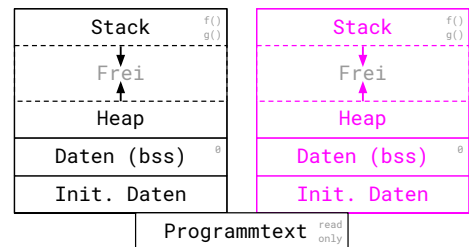
File Deskriptoren werden bei *fork()* mit *dup()* kopiert:

FDs/Proc. Open Files/OS i-nodes/Filesys.



Child kann ungenutzte FDs schliessen. n|w

Speicher Layout nach *fork()*



n|w

Speicher Semantik von *fork()*

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

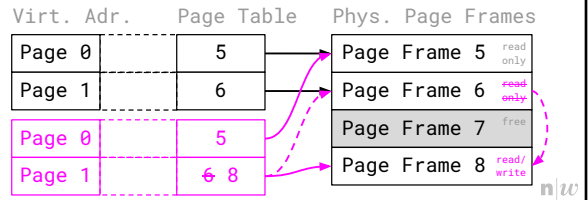
Die Programmtext-Segmente von Parent und Child zeigen auf eine physische Page, die *read-only* ist.

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.

n|w

Prozess Page Table *copy-on-write*

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



n|w

Funktion in *fork()* wrappen [footprint.c](#)^{TLPI}

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()
```

n|w

Race Conditions [fork_whos_on_first.c](#)^{TLPI}

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

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

Auf Linux kann die Reihenfolge beeinflusst* werden:

```
$ cat /proc/sys/kernel/sched_child_runs_first
Child-first kann das Kopieren von Pages minimieren,
hier eine Email von Linus Torvalds zum Thema.

```

n|w

Synchronisation [fork_sig_sync.c](#)^{TLPI}

Signale helfen, Race Conditions zu verhindern, wenn einer der Prozesse auf den anderen warten muss, z.B. wird hier *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()*.

n|w

Hands-on, 45': Signal Masken [sync.c](#)

Schreiben Sie ein Programm *my_sync.c*, das den Code von [fork_sig_sync.c](#)^{TLPI} wie folgt erweitert:

Der neue Code soll ohne *tlpi_hdr.h* und *curr_time.h* builden und Signal Masken für *blocked* und *pending* Signale ausgeben, dort im Code, wo sich was ändert.

Hinweis: Dokumentation von *sigprocmask()* lesen, *sigpending()* und *printf("%032x", ...)* nutzen.

n|w

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 $\neq 0$ bedeutet, es gab einen Fehler. Meistens wird der `exit()` Library Call verwendet.

n|w

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.

n|w

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)`.

n|w

Prozess Termination im Detail

Bei normaler und abnormaler Prozess Termination werden die offenen File Deskriptoren geschlossen, und File Locks und Speicher-Mappings freigegeben, sowie weitere Ressourcen im Zusammenhang mit Shared Memory, Semaphoren und Prozessgruppen.

Manchmal will man selber Ressourcen aufräumen, mit mehr Kontrolle, dazu gibt es Exit Handler.

n|w

Exit Handler, `atexit()` `exit_handlers.c`^{TLPI}

Die *glibc* Library erlaubt, Exit Handler zu registrieren:
`int atexit(void (*h)(void)); // != 0 => Error`
`void cleanup(void) { ... } // Beispiel Handler`

Handler werden in eine Liste eingefügt, und bei `exit()` in umgekehrter Registrationsreihenfolge aufgerufen.

Der Handler bekommt den Exit *status* nicht mit, und sollte selber `exit()` auch nicht nochmal aufrufen.

n|w

Exit Handler, `on_exit()`

glibc bietet eine Alternative, Handler zu registrieren:
`int on_exit(void (*h)(int, void *), void *arg);`
`void cleanup(int status, void * arg) { ... }`

Das *arg* Argument wird beim Registrieren übergeben, und wird nur vom Handler interpretiert bzw. gecastet.

Die Funktion ist nicht Standard, d.h. nicht portabel.

Im Fehlerfall liefert `on_exit()` einen Wert $\neq 0$.

n|w

Hands-on, 15': `exit()` `fork_stdio_buf.c`!pdf

Finden Sie heraus, wieso sich in `fork_stdio_buf.c`^{TLPI} der Output dieser beiden Aufrufe unterscheidet:

```
$ ./fork_stdio_buf  
$ ./fork_stdio_buf > file && cat file
```

Wieso wird ein Teil des Outputs doppelt ausgegeben?

Wieso wird nur im einen Fall der Output verdoppelt?

Hinweis: Was passiert bei `fork()` im Speicher?

n|w

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.

n|w

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.

n|w

Hands-on, 15': `make_zombie.c`^{TLPI}

Lassen Sie den Beispiel-Code `make_zombie.c` laufen.

Senden Sie dem Zombie-Child ein `SIGKILL` Signal.

Was macht der `system()` Aufruf im Source Code?

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

n|w

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);
```

n|w

Programm ausführen `t_execve.c`^{TLPI}

Der `execve()` Call ersetzt das laufende Programm:

```
int result = execve(filepath, argv, envp);
```

Das Programm `filepath` startet normal, mit `main()`.

Die PID des ausführenden Prozesses bleibt dieselbe.

Der `return`-Wert kann nur -1 sein, sonst kein `return`.

Zum Beispiel oben braucht's noch `envargs.c`^{TLPI}

n|w

`exec()` Library Funktionen

`execlp.c`

Von `exec()` gibt es einige Varianten, z.B. `execlp()`:

```
int execlp( // aus der exec() Familie
    const char *file, // statt filepath
    const char *arg, ... /* (char *) NULL */);
```

Das `l` bedeutet, dass die Argument-Liste "offen" ist:

```
execlp("curl", "-v", "tmb.gr", (char *) NULL);
```

Das `p` bedeutet, dass das `file` im `$PATH` gesucht wird, wie in der Shell, wenn man keine `'/'` verwendet.

`n|w`

File Deskriptoren und `exec()`

Per default bleiben File Deskriptoren bei `exec()` offen, die Shell nutzt dieses Verhalten, um I/O umzuleiten:

```
$ ls /tmp > dir.txt
```

Shell forked, Child öffnet `dir.txt` mit `fd = 1`, als `stdout`:

```
fd=open(...); dup2(fd, STDOUT_FILENO); close(fd);
```

Das Child lässt `ls` laufen mit `exec()`, Output auf `stdout`.

Manche Shell-Kommandos sind eingebaut, z.B. `cd`. `n|w`

Das `close-on-exec` Flag `closeonexec.c`^{TLPI}

Manchmal möchte man Files schliessen, vor `exec()`.

Man könnte `close()` aufrufen, aber das Flag ist besser:

```
int flags = fcntl(fd, F_GETFD); // get flags
flags |= FD_CLOEXEC; // add close-on-exec
fcntl(fd, F_SETFD, flags); // set flags
```

So wird ein File Deskriptor nur geschlossen, wenn der `exec()` Aufruf erfolgreich ist, nicht im Fehlerfall.

`n|w`

Signale und `exec()`

Bei einem `exec()` wird der Programmtext verworfen.

Dabei verschwinden potentiell auch Signal Handler.

Deshalb setzt der Kernel alle Handler auf `SIG_DFL`, ausser denen, die mit `SIG_IGN` Signale ignorieren.

Die Prozess Signal Maske und das Set von `pending` Signalen bleiben beide intakt während dem `exec()`.

`n|w`

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"
```

Details von `fork()`, `exec()`, `wait()`, and `exit()` versteckt.

Fehler- und Signal-Handling werden übernommen.

Der `system()` Call nutzt die Shell, wie "von Hand".

`n|w`

Hands-on, 15': `simple_system.c`^{TLPI}

Implementieren Sie eine eigene `system()` Funktion.

Nutzen Sie dazu das `sh` Kommando mit Argument `-c`:

```
$ sh -c "ls | wc"
```

Vereinfachung: Gerüst von `my_system.c` im Repo.

`n|w`

Signals und *system()*

system.c^{TLPI}

Das Beispiel *system.c* zeigt eine robustere Variante.

Der Caller sollte *SIGINT* und *SIGQUIT* während dem Kommando mit *SIG_IGN* behandeln, das Child mit *SIG_DFT*, um verwirrendes Verhalten zu vermeiden.

Beim Warten wird hier *waitpid()* benutzt, um auf den Child-Prozess zu warten, den wir gestartet haben.

n|w

Warten mit *waitpid()*

child_status.c^{TLPI}

Der *waitpid()* Call erlaubt, auf einen konkreten Child-Prozess zu warten. Im Gegensatz zu *wait()* blockiert der Call nicht, wenn noch kein Child beendet wurde:

```
pid_t waitpid(pid_t pid, // Child PID, 0, ...
              int *status, // Makro WIFEXITED(status), ...
              int options); // WNOHANG => non-blocking, ...
```

Beispiel *child_status* verwendet *print_wait_status.c*:

```
$ ./child_status 23 # exit() mit Status 23
```

n|w

Selbststudium, 3h: Topic

Als Vorbereitung auf die nächste Lektion, lesen Sie <https://computing.llnl.gov/tutorials/threads/> bis *Pthread Excercise 1*.

Falls Zeit übrig bleibt, beginnen Sie mit Repetieren der bisherigen Lektionen für das Assessment.

n|w

Feedback?

Gerne im [Slack](#) oder an thomas.amberg@fhnw.ch

Programmierfragen am besten schriftlich.

Sprechstunde auf Voranmeldung :)

Slides, Code & Hands-on: tmb.gr/syspr-5

