System-Programmierung 4: Prozesse und Signale

Slides: tmb.gr/syspr-4

Überblick

Diese Lektion behandelt *Prozesse und Signale*.

Wie der Kernel laufende Programme verwaltet.

Das Layout des Speichers, mit Stack und Heap.

Wie ein Prozess auf Signale hören kann.

Prozesse und Programme

Prozesse sind Instanzen eines laufenden Programms.

Ein *Programm* ist ein File mit Informationen, wie ein Prozess zur Laufzeit konstruiert werden soll.

Ein Programm kann mehrere Prozesse kreieren, und mehrere Prozesse können dasselbe Programm laufen lassen bzw. instanzieren.

Programm Binary Aufbau

Binärformat	- z.B. Executable & Linking Format (ELF).
Instruktionen	- Das "Programm" in Maschinensprache.
Eintrittspunkt	- Instruktion bei der das Programm startet.
Programmdaten	- Initialwerte, Konstanten, String-Literale.
Relokations- und Symboltabellen	- Position und Name von Funktionen und Variablen im Programm, für Debugging.
Shared Libraries	- Liste der Libraries und Pfad des Linkers.
Weitere Angaben	- Informationen, wie Prozess gebaut wird.

Prozess aus Kernel Sicht

Der Kernel sieht Prozesse als User-Space Speicher mit Programmcode und Initialwerten von Variablen.

Zudem unterhält der Kernel Datenstrukturen, um den Zustand von Prozessen zu managen, z.b. Prozess IDs, Virtual Memory, Offene File Deskriptoren, Signal-Handling, Ressourcenverbrauch und Limiten, das aktuelle Arbeitsverzeichnis und einiges mehr.

Prozess IDs

Jeder Prozess hat eine *Prozess ID* (PID), eine ganze, positive Zahl die den Prozess im System identifiziert.

Der *init* Prozess mit dem Linux startet, hat die PID 1.

Der *getpid()* System Call liefert die PID des Callers:

```
pid_t getpid(void); // geht immer ohne Error
```

Prozess Baumstruktur

pid.c

Jeder Prozess hat einen sogenannten Parent-Prozess:

```
pid_t getppid(void); // geht immer ohne Error
```

Die Prozesse bilden einen Baum, mit *init* als Wurzel:

```
$ pstree # zeigt den aktuellen Prozess-Baum
$ cat /proc/PID/status | grep PPid # PID = …
```

Wenn ein Prozess verwaist, wird er von *init* adoptiert, d.h. die *getppid()* Funktion gibt ab dann 1 zurück.

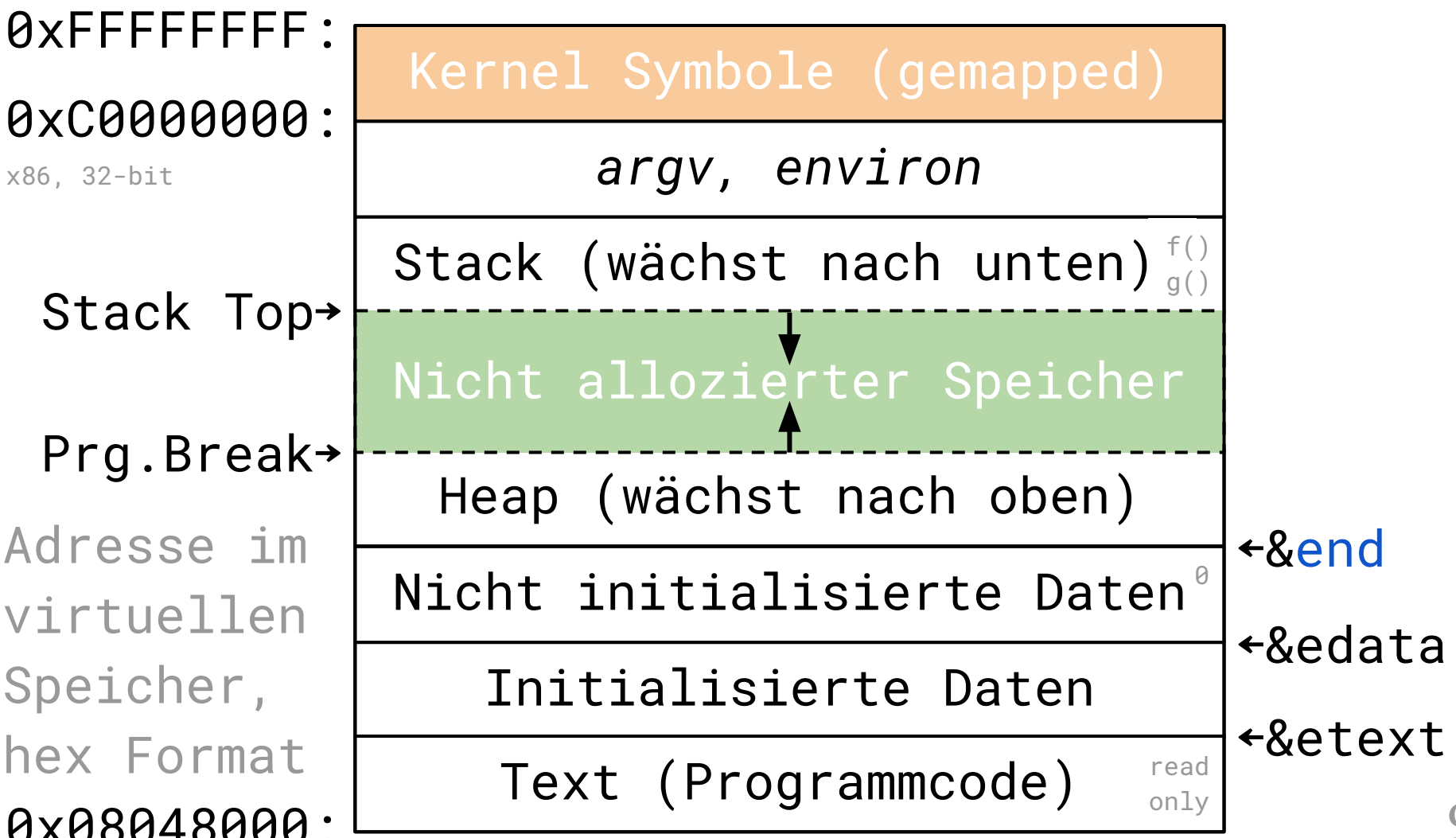
Prozess Speicherlayout segments.c

Der Speicher jedes Prozesses ist in *Segmente* geteilt: Text bzw. Programmcode, initialisierte Daten, nicht initialisierte Daten (bss), Stack und Heap.

Das Beispiel *segments.c* zeigt deren Adressen, Grösse.

Diese findet man auch per Command Line:

```
$ cat /proc/PID/maps # zeigt Segment Adressen
$ size segments # Grösse von text, data & bss
```


0xFFFFFFFF:
0xC0000000:
x86, 32-bit
Kernel Symbole (gemapped)
argv, environ
Stack (wächst nach unten)
f()
g()
Stack Top→
Nicht allozierter Speicher
Prg.Break→
Heap (wächst nach oben)
Nicht initialisierte Daten
0
Initialisierte Daten
Text (Programmcode)
read only
←&end
←&edata
←&etext
Adresse im virtuellen Speicher, hex Format
0x08048000:

Virtueller Speicher

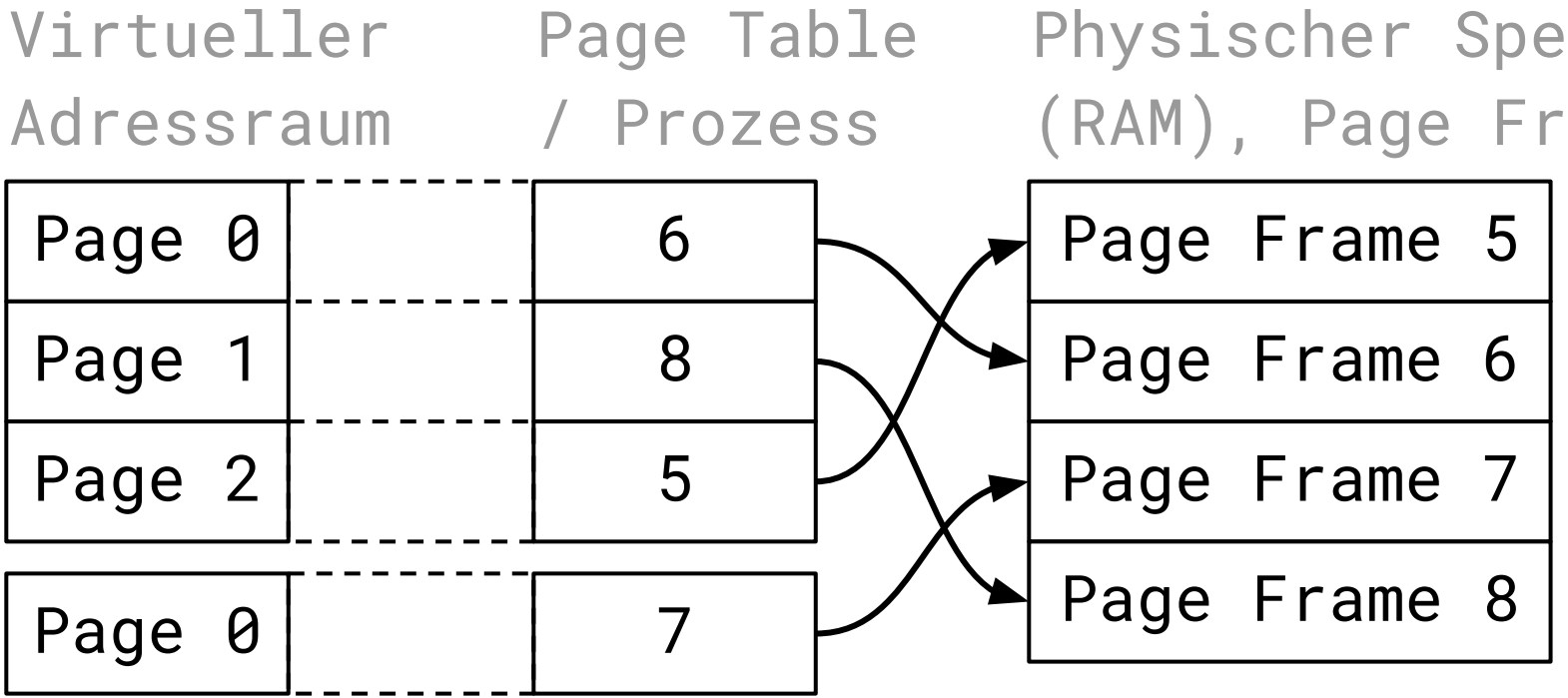
Linux managt Speicher als *Virtual Memory*, so wird eine effiziente Nutzung von CPU und RAM erreicht.

Dadurch muss nur ein Teil des Adressraums ins RAM geladen werden, eine *Page*, der Rest ist ausgelagert.

Das geht, weil aufeinanderfolgende Speicherzugriffe oft zeitlich und örtlich in der Nähe voneinander sind, z.B. bei *while*-Schleifen, oder Array-Zugriffen `a[i]`.

Prozess Page Table

Der Kernel hat für jeden Prozess eine *Page Table*:



Mapping auf physischen Speicher

Die Page Table beschreibt das Mapping von Pages im virtuellen Speicher auf physischen Speicher (RAM).

Unbenutzter virtueller *Adressraum* ist nicht gemappt, der gültige Adressraum kann sich ändern, wenn der Kernel zur Laufzeit Pages alloziert, wieder freigibt.

Wenn ein Prozess auf ungemappten Speicher zugreift, gibt es einen *Segmentation Fault* (SIGSEGV).

Vorteile von virtuellem Speicher

Prozesse sind voneinander und vom Kernel isoliert.

Programmtext kann (read-only) geshared werden.

Pages sind markierbar als read-/write-/executable.

Entwickler müssen physisches Layout nicht kennen.

Das Programm lädt/läuft schneller, und spart RAM.

Stack und Stack Frames

Pro Funktionsaufruf wird ein *Stack Frame* alloziert, das nach dem *return* wieder vom *Stack* entfernt wird.

Der Stack wächst mit jedem (verschachtelten) Aufruf und schrumpft beim *return* wieder um ein Frame.

Ein Stack Frame enthält Funktionsargumente, lokale "automatische" Variablen und CPU Register Kopien.

Command Line Argumente

args.c[01]

Die *main()* Funktion eines Programms wird von der Shell aufgerufen mit den Argumenten *argc* und *argv*:

```
int main(int argc, char* argv[]);
```

Der Name des Programms ist in *argv[0]* enthalten.

Die Elemente von *argv* sind Null-terminierte Strings.

Command Line eines Programms in Linux anzeigen:

```
$ cat -v /proc/PID/cmdline # mit ^@ für '\0'
```

Umgebungsvariablen environ.c

Environment, enthalten in einer globalen Variablen:

```
extern char **environ; // Liste von Strings
```

Auf die Umgebungsvariable namens *name* zugreifen:

```
char *getenv(const char *name); // oder NULL
int putenv(char *string); // != 0 => Error
```

Umgebungsvariablen anzeigen per Command Line:

```
$ sudo cat /proc/PID/environ
```

Dynamische Speicherallokation

Manchmal brauchen Programme neuen Speicher für dynamische Datenstrukturen wie Listen, Bäume, etc. deren Grösse erst zur Laufzeit bekannt ist.

Dieser Speicher kommt meistens vom Heap, dessen Grösse über System Calls verändert werden kann.

Wir betrachten die Library Calls *malloc()* und *free()* und deren Implementierung mit *sbrk()* und *brk()*.

Heap Speicher allozieren

Heap Speicher allozieren mit *malloc()*:

```
void *malloc(size_t size); // Zeiger oder NULL
```

Den *void*-Pointer darf man jedem Typ zuweisen, z.B.

```
struct rect *r = malloc(sizeof(struct rect));
```

Heap Speicher freigeben mit *free()*:

```
void free(void *p); // genau einmal pro Zeiger
```

Fehlerfälle *bei malloc()* und *free()*

Bei zu wenig Speicher liefert *malloc()* den Wert NULL:

```
void *p = malloc(size);
if (p == NULL) { perror("malloc"); exit(-1); }
```

Mehrfachaufruf von *free()* auf denselben Pointer:

```
free(p); // Verhalten undefiniert, z.B. SIGSEGV
```

Deshalb Pointer nach *free()* auf NULL setzen:

```
p = NULL; // Erneute Freigabe von p verhindern
free(p); // Freigabe von NULL, kein Effekt
```

Heap Grösse direkt setzen

Der *Program Break* markiert die Grenze des Heaps, *sbrk(i)* verschiebt ihn um i, gibt alte Adresse zurück:

```
void *sbrk(intptr_t increment);
```

brk() setzt ihn an eine neue Adresse im Speicher:

```
int brk(void *addr);
```

Diese System Calls erlauben es, die beiden Library Funktionen *malloc()* und *free()* zu implementieren.

Naive Implementierung malloc.c, _v2.c

1. *malloc(), free()* verschieben einfach Program Break:

```
void *p = sbrk(size); // ~= malloc(size)
brk(p); // ~= free(p)
```

Ist aber nur für genau eine Aufrufreihenfolge korrekt.

2. Besser: Library führt intern eine dynamische Liste, wie auf den nächsten beiden Seiten beschrieben. Aber für diese Liste bräuchte man auch wieder *malloc()*.

Implementierung von *malloc()*

Scanne Liste freier Blöcke, bis Block ≥ *n*, 1st-/best-fit.

Grösse ist *n*? *return* Block, sonst den zu grossen Block aufteilen, in einen *return*- und einen freien Block.

Kein freier Block? Heap mit *sbrk()* um $N \geq n$ Bytes vergrössern, mit N = ein Vielfaches der Page-Grösse, *return* Block, Rest als Block in die Liste freier Blöcke.

Implementierung von *free()*

Pointer *p* zeigt auf einen Block Grösse *n*.

Block in die Liste der freien Blöcke einfügen.

Block mit angrenzenden Blöcken verschmelzen*.

Program Break mit *brk()* verkleinern, falls möglich**.

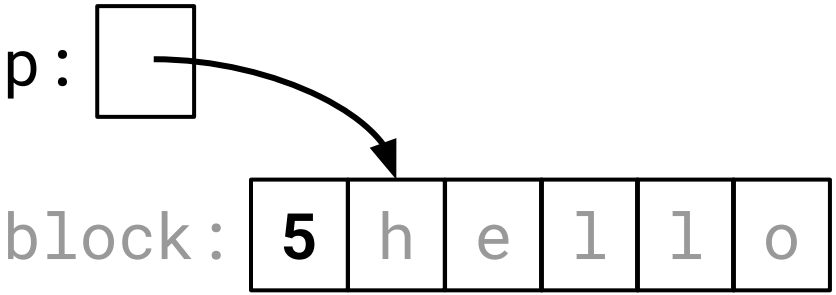
*Defragmentieren, **Heap schrumpfen

In-place Datenstruktur

impl.c

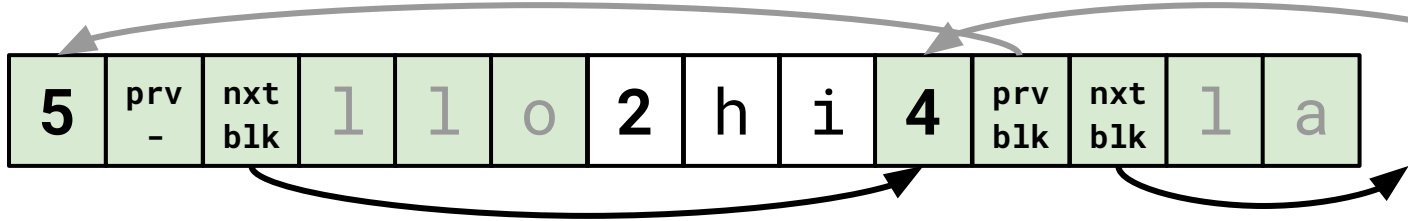
malloc() speichert die Block Länge links vom Pointer:

```
void *p = malloc(n); // see TLPI p. 144 ff.
```



Achtung: free() nur auf Pointer die von malloc() kommen

free() nutzt freie Blöcke für *prev*, *next* Pointers:



Signale

Ein *Signal* sagt dem Prozess, dass etwas passiert ist.

Signale sind eine Art Software Interrupts, da sie wie echte Interrupts den Programmablauf unterbrechen. Man weiss oft nicht genau, wann ein Signal kommt.

Prozesse können Signale an andere Prozesse senden, und an sich selbst, die meisten Signale kommen aber vom Kernel (HW Exceptions, Input, SW Events).

Signal Ursachen

Im Kernel ist eine *Hardware Exception* aufgetreten, z.B. Division durch Null oder Segmentation Fault.

Bei *User Input*, z.B. *CTRL-C* oder *CTRL-Z* gedrückt.

Software Events, z.B. wenn ein Timer abgelaufen ist, oder Input verfügbar wird an einer File Schnittstelle, oder wenn ein Prozess ein Signal verschickt hat.

Signal Ablauf

Eine Quelle (Kernel, Prozess) *generiert* ein Signal.

Bis zur Auslieferung ist das Signal *pending* (hängig).

Sobald der Prozess an der Reihe ist, wird es *geliefert*.

Der Prozess kann nun (mit Core Dump) terminieren, stoppen, wieder weiterlaufen, das Signal ignorieren, oder das Signal behandeln, mit einem *Handler*.

Signal Nr. und Symbole

Jedes Signal hat eine Nr. und ein *SIGxxxx* Symbol:

```
#define SIGINT 2 // in signal.h
```

Standard Signale vom Kernel an den Prozess: 1 - 31.

Daneben gibt es in Linux *real-time* Signale: 32 - 64, für Anwendungs-spezifische Use Cases: `SIGRTMIN+n`

Auf Signale sollte man immer per Symbol verweisen, weil sie je nach System verschiedene Nr. haben.

Signal Handler installieren

Der *signal()* Call setzt für ein Signal *s* den Handler *h*:

```
void (*signal(int s, void (*h)(int))) (int);
```

Ein *Signal Handler* hat demnach die folgende Form:

```
void handle(int signal) { … }
```

Der *return*-Wert ist die vorherige Handler-Funktion:

```
old_handle = signal(SIGINT, handle); // save
// do something else, handle handles SIGINT
… signal(SIGINT, old_handle); // restore
```

Hands-on, 15': Sig. Handler

sigint.!c

Schreiben Sie ein Programm *my_sigint.c*, welches das Signal *SIGINT* mit einem Handler behandelt.

Die Handler Funktion soll dabei *handle()* heissen.

Senden Sie dem Programm *SIGINT* mittels *CTRL-C*.

Signal Handler Konstanten

Bei einem Fehler ist der *return*-Wert *SIG_ERR*:

```
result = signal(SIGINT, handle);
if (result == SIG_ERR) { … }
```

Für default Signal Handler, *SIG_DFL* installieren:

```
result = signal(SIGINT, SIG_DFL);
```

Signal ignorieren, d.h. Handler *SIG_IGN* installieren:

```
result = signal(SIGINT, SIG_IGN);
```

Signal Handler Design

Signal Handler sollten so einfach wie möglich sein, um Race Conditions zu verhindern bei Reentrance.

Oft setzt der Handler einfach nur ein globales Flag:

```
volatile int flag; // portabel: sig_atomic_t
```

Statt signal() sollte man sigaction() verwenden, um den Handler zu übergeben, für portableren Code*.

*Selbes Prinzip, Semantik genauer spezifizierbar.

Signale senden mit *kill()*

Ein Signal *sig* an den Prozess *pid* senden, mit *kill()*:

```
int kill(pid_t pid, int sig); // or -1, errno
```

Falls *pid* = *0* ist, geht das Signal an alle in derselben Prozess-Gruppe, wie der aufrufende Prozess.

Mit *pid* = *-1* geht das Signal an alle Prozesse, an die der Aufrufer Signale senden darf, ausser an *init*.

Mit *sig* = *0* wird geprüft, ob Senden möglich ist.

Signale senden mit *raise()*

raise() sendet ein Signal *sig* an den eigenen Prozess:

```
int raise(int sig); // != 0 bei Error EINVAL
```

Das Aufrufen von *raise()* hat denselben Effekt wie:

```
kill(getpid(), sig);
```

Oder, in einem Programm mit mehreren Threads:

```
pthread_kill(pthread_self(), sig);
```

Der einzige Fehler ist *EINVAL*, falls *sig* ungültig.

Warten auf Signale

pause.c

Die *pause()* Funktion suspendiert den Prozess, bis ein Signal eintrifft, danach wird -1 zurückgegeben:

```
int pause(void); // -1, errno = EINTR
```

Prüfen ob ein vom Handler gesetztes Flag aktiv ist:

```
pause(); // wartet auf das nächste Signal
// installierter Handler wird aufgerufen
// der Handler setzt ein globales Flag
if (flag) { … } // siehe auch pause.c
```

Signal Beschreibung ausgeben

strsignal() liefert die Beschreibung des Signals *sig*:

```
char *strsignal(int sig); // oder NULL
```

Der String ist nur bis zum nächsten Aufruf gültig.

psignal() druckt eine Fehlermeldung *msg*, gefolgt von der Beschreibung von *sig* und \n auf *stderror*:

```
void psignal(int sig, const char *msg);
```

Signale Maskieren

Darf ein Prozess nicht unterbrochen werden, setzt man eine *Maske* um einzelne Signale abzublocken.

Die blockierten Signale eines Prozesses anzeigen:

```
$ cat /proc/PID/status # SigBlk
```

Signale bleiben *pending* bis sie entblockt werden:

```
$ cat /proc/PID/status # SigPnd, ShdPnd
```

Es gibt keine Queue, Signal-Masken sind nur Sets.

Signale maskieren mit *sigmask()*

Signal Set *old* durch *new* ersetzen mit *sigprocmask()*:

```
int sigprocmask(int how, // e.g. SIG_SETMASK
  const sigset_t *new, // e.g. NULL = get old
  sigset_t *old); // ergibt 0 oder -1, errno
```

Der *how* Parameter kann folgende Werte annehmen:

```
SIG_BLOCK; // Signale in new werden hinzugefügt
SIG_UNBLOCK; // Signale in new werden entfernt
SIG_SETMASK; // Signale in new werden gesetzt
```

Sets von Signalen

Für Sets von Signalen gibt's den System Typ *sigset_t*.

Mit *sigemptyset()* oder *sigfillset()* wird ein Set kreiert:

```
int sigemptyset(sigset_t *set); // leeres Set
int sigfillset(sigset_t *set); // alle im Set
```

sig{add|del}set() schliesst das Signal *sig* ein oder aus:

```
int sigaddset(const sigset_t *set, int sig);
int sigdelset(const sigset_t *set, int sig);
```

Selbststudium: Prozess Kreation

Als Vorbereitung auf die nächste Lektion, lesen Sie [TLPI] *Chapter 24: Process Creation*.

Das PDF des Kapitels 24 ist verfügbar als Leseprobe.

Die nächste Lektion fasst den Lesestoff zusammen, ohne Selbststudium wird das Tempo eher hoch sein.

PS. Eine gute Übersicht zu Signals finden Sie hier.

Feedback oder Fragen?

Gerne in Teams, oder per Email an

thomas.amberg@fhnw.ch

Danke für Ihre Zeit.

Raspberry on Twitter: "I love th

https://twitter.com/illusionfoxe/status/1041219052847263745

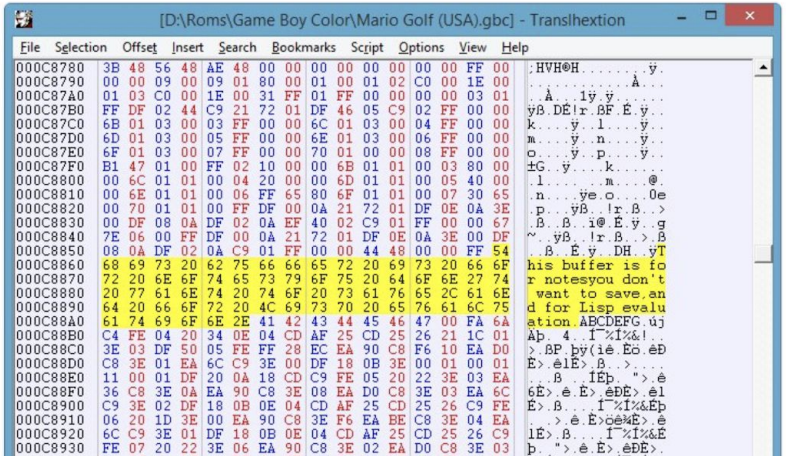
Home

Tweet

Raspberry
@illusionfoxe

Follow

I love that the default Emacs scratch buffer text got into Mario Golf (GBC) somehow. I wonder if this was a mis-copy-and-paste from a developer, or if it was another case of a compiler taking random memory contents and using them for padding in the ROM...



Raspberry
@illusionfoxe

(she/her) Persona
@raspberryfloof.
to be a game dev
Icon by @AruyaSe