Linux/Unix Systemprogrammierung

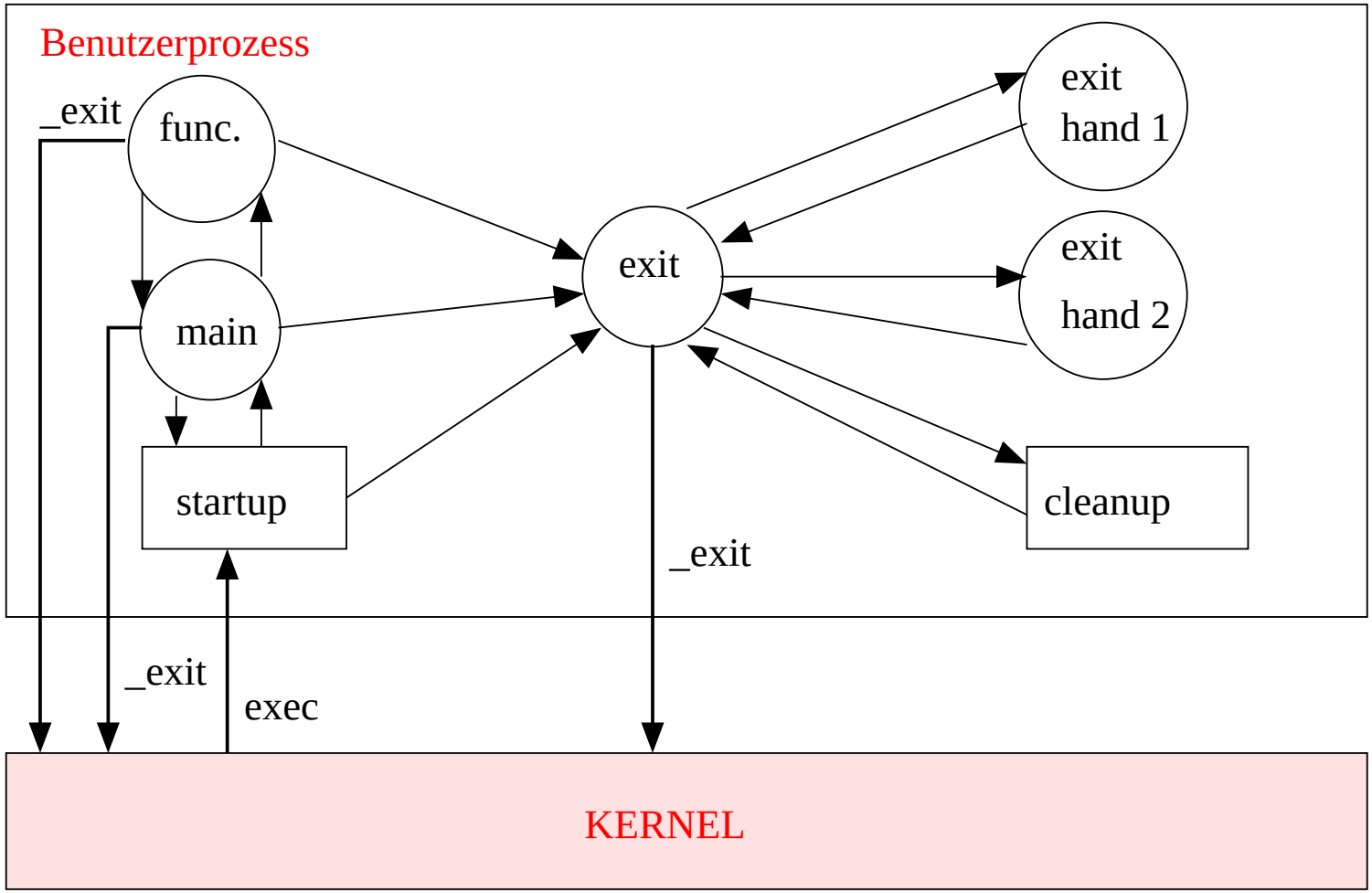
Pablo Yánez Trujillo

Poolmanager Team
Universität Freiburg

August 16, 2006

Der Unix Prozess

- Bisher haben wir Programme geschrieben, die über die `main` Funktion verfügen
- Aber wie wird ein Programm gestartet? Was macht der Kernel, damit ein Programm gestartet wird?
- Programme sind Prozesse, d.h. damit ein Programm gestartet wird, muss einen neuen Prozess erzeugt werden, in dem das Programm gestartet wird



Auszug: Linux/Unix Systemprogrammieru

- Die `main` Funktion kennen wir bereits
- Sie dient nicht nur dazu, das Programm zu starten, sondern sie setzt den **exit-Status** eines Prozesses
- Der **exit-Status** ist eine Zahl ungleich 0, wenn der Prozess nicht erfolgreich beendet wurde
- Der **exit-Status** ist 0 sonst
- Skripte richten sich danach, deswegen ist es wichtig den exit-Status **immer** anzugeben

exit: normales Beenden

```
#include <stdlib.h>

#define EXIT_FAILURE 1 /* Failing exit status.  */
#define EXIT_SUCCESS 0 /* Successful exit status.  */
void exit(int status);
```

_exit: sofortiges Beenden

```
#include <unistd.h>

void _exit(int status);
```

atexit: Registrierung von Funktionen

```
#include <stdlib.h>

int atexit(void (*funktion)(void));
```

- Jedes System verfügt über Environment Variablen, Unix macht da keine Ausnahme
- Es gibt Funktionen, mit denen diese Variablen zur Laufzeit manipuliert werden können

getenv: Variable lesen

```
#include <stdlib.h>

char *getenv(const char *name);
```

- Bei Erfolg: Zeiger auf String mit dem Wert. Sonst NULL
- Laut ANSI C sollte man den zurückgegebenen Speicher nicht modifizieren
- Lait ANSI C verwendet immer den selben Speicher, nach einem erneuten Aufruf ist der alte Inhalt weg.

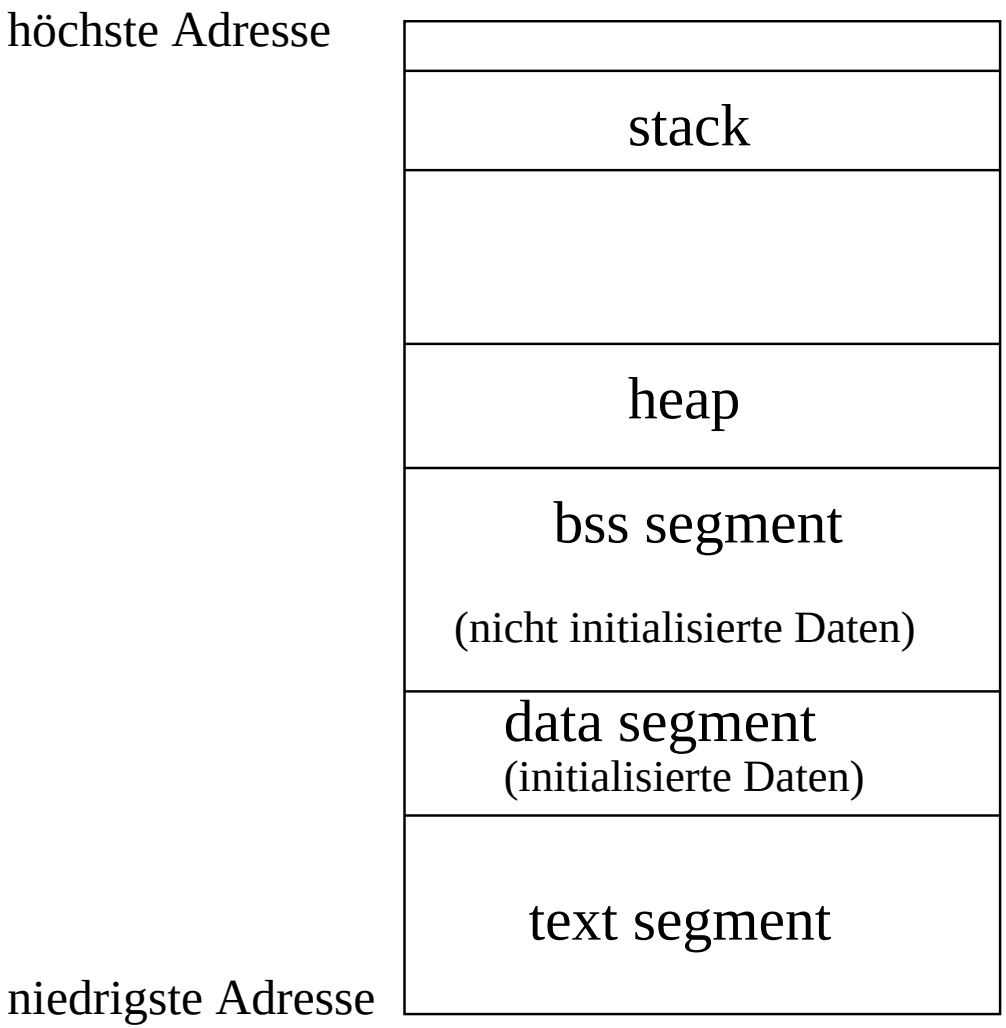
Variable schreiben

```
#include <stdlib.h>

int putenv(const char *eintrag); /* Format:  var=val */
int setenv(const char *var, const char *val, int overw);

void unsetenv(const char *var);
```

- Beide Funktionen liefern bei Erfolg 0 zurück
- Es sind äquivalent

```
putenv("PATH=/bin:/usr/bin:.");
setenv("PATH", "/bin:/usr/bin:.", 1);
```

- Ein Prozess muss im Speicher gespeichert werden, damit es ausführbar wird
- Ein Unix-Prozess besteht aus:
  - *text segment*
  - *data segment*
  - *bss segment*
  - *heap*
  - *stack*


höchste Adresse
stack
heap
bss segment
(nicht initialisierte Daten)
data segment
(initialisierte Daten)
text segment
niedrigste Adresse

- Wir wissen jetzt, was Prozesse sind. Wir wollen aber sie steuern
- Prozesse werden vom Systemkern durch die Prosess ID (pid) unterschieden
- Jeder Prozess hat einen Vater-Prozess (ppid)
- Jeder Prozess gehört einem Benutzer und erbt seine Rechte
- Jeder Prozess gehört einer Gruppe und erbt ihre Rechte
- Jeder Prozess hat eine efektive PID und GID

Erfragen der PID und PPID

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);
```

- Beide Funktionen liefern die PID bzw. PPID zurück

Erfragen der realen und effektiven User-ID

```
#include <sys/types.h>
#include <unistd.h>

pid_t getuid(void);

pid_t geteuid(void);
```

- Beide Funktionen liefern die reale User-ID bzw. effektive User-ID zurück

Erfragen der realen und effektiven Group-ID

```
#include <sys/types.h>
#include <unistd.h>

pid_t getgid(void);

pid_t getegid(void);
```

- Beide Funktionen liefern die reale Group-ID bzw. effektive Group-ID zurück

- *Scheduler-Prozess* mit PID 0: Systemprozess ist Teil des Kerns
- *init-Prozess* mit PID 1: Eltern-Prozess für user-land Prozesse. `init` wird vom Scheduler gestartet, nachdem der Kernel geladen wird. Die Konfigurationsdatei von `init` ist `/etc/inittab`. Genau wie der Schedular beendet `init` nicht, aber es ist ein Benutzerprozess und kein Systemprozess

- Prozesse werden nicht von der Luft erzeugt (gestartet)
- Alle user-land Prozesse (bis auf init) müssen von einem anderen Prozess erzeugt werden
- Schon mal überlegt: Wie schafft die Shell, dass bei Eingabe eines Programmnames ein Programm startet?
- Ganz einfach: die Shell erzeugt einen Prozess mittels `fork` und ruft dann `exec` aus (In Wahrheit macht die Shell viel mehr als das)

Einen neuen Prozess erzeugen

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

- `fork` erstellt eine Kopie (im Speicher) des aktellen Prozesses, und diese Kopie läuft als ein neuer Prozess (mit unterschiedlichen Instruction Pointer)
- `fork` gibt bei dem Kindprozess 0 zurück
- `fork` gibt bei dem Elternprozess die PID des Kindprozesses zurück
- `fork` gibt -1 zurück, wenn `fork` nicht in der Lage war, einen neuen Prozess zu erzeugen

fork Beispiel

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


int main(void)
{
    pid_t child;


    printf("Hallo\n");
    child = fork();


    switch(child) {
        case -1:
                fprintf(stderr, "Fehler bei fork()\n");
                return 1;
        case 0:  /* Kindprosess */
                printf("Ich bin das Kind meines Vaters (%d) mit pid %d\n", getppid(), getpid());
                break;
        default:  /* Elternprosess */
                printf("Ich bin der Vater mit pid %d\n", getpid()); break;
    }


    return 0;
}
```

- Untersuche das Programm `tag03/fork`
- Führe das Programm aus, wie sieht die Ausgabe aus?
- Was passiert, wenn man `fflush(stdout)` nach jeder `printf`-Anweisung schreibt? Warum?
- Was kann man dagegen machen?

- Manchmal will man, dass der Elternprozess auf seine Kinder wartet
- Wenn z.B. der Elternprozess wissen will, welchen exit-status sein Kind hat
- Der Elternprozess muss dann auf sein Kind warten

Auf Kind warten

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```

- Beide Funktionen halten den Elternprozess auf, bis das Kindprozess sein Status geändert hat
- `wait` wartet auf das aktuellste Kindprozess
- `waitpid` wartet auf das Kindprozess mit PID `pid`

Einen neuen Prozess erzeugen

```
#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);
```

- `vfork` tut dasselbe `fork` mit einem Unterschied
- `vfork` kopiert den Adressraum nicht sondern lässt das Kindprozess denselben Adressraum benutzen
- Das Beispiel `tag03/vfork` verdeutlicht diesen Unterschied, Kompiliere dieses Beispiel und führe es aus

- Wan braucht man einen neuen Prozess?
- Wenn man z.B. will, dass ein Prozess ein anderes Programm ausführt, seine Ausgabe liest oder seine Eingaben beeinflussen
- Dafür gibt es eine Familie von Funktionen, die genau das realisieren: `exec`

Die exec Familie

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- Der text-segment wird ersetzt, das bedeutet, dass exec den aktuallen Prozess durch das neue Programm ersetzt
- Deswegen muss man vor dem Aufruf von exec einen Prozess erzeugen, wenn man will, dass der Elternprozess nicht verschwindet

Vererbung

- IDs (PID, PPID, reale und effektive UID/GID)
- Working-Directory und Root-Directory
- umask
- Kontrollterminal
- Ressourcenlimits

Änderung der User-ID und Group-ID

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_tuid);

int setgid(gid_tgid);
```

- Beide Funktionen liefern bei Erfolg 0 zurück, -1 sonst
- Sowohl die reale als auch die effektive UID/GID wird geändert
- Nur superuser kann jede UID/GID wechseln

- Schreibe ein Programm, welches einen Befehl als einem anderen Benutzer ausführt
- `executeas user prog arg1 arg2 arg3 ...`

Aus Zeitgründen wird es erst morgen (schnell) behandelt