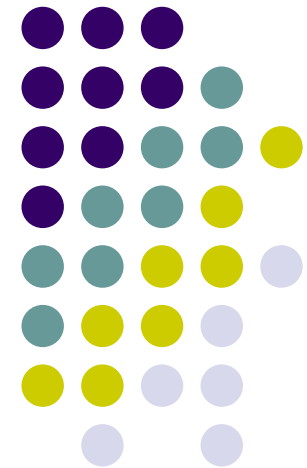


Einführung in die Systemprogrammierung

Prozesssteuerung



Überblick

- Prozesskennung und die Unix-Prozeshierarchie
- Kreieren von neuen Prozessen
- Die exec-Funktionen

Prozesskennungen

- Prozess-IDs
 - PID: eindeutige Prozesskennung
 - PPID (Parent Process ID): ID des Elternprozesses
 - User-ID
 - Group-ID
- Erfragen von PID und PPID
 - Funktionen *getpid* und *getppid*

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

pid_t getpid(void);
pid_t getppid(void);
```

Prozesskennungen

- Erfragen von User-ID und Group-ID
 - Funktionen *getuid* und *getgid*

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

pid_t getuid(void);
pid_t getgid(void);
```

- Ausgabe von PID, PPID:

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

int main (int argv, char *argv[])
{
    printf("PID/PPID: %d/%d\n", getpid(), getppid());
    exit (0);
}
```

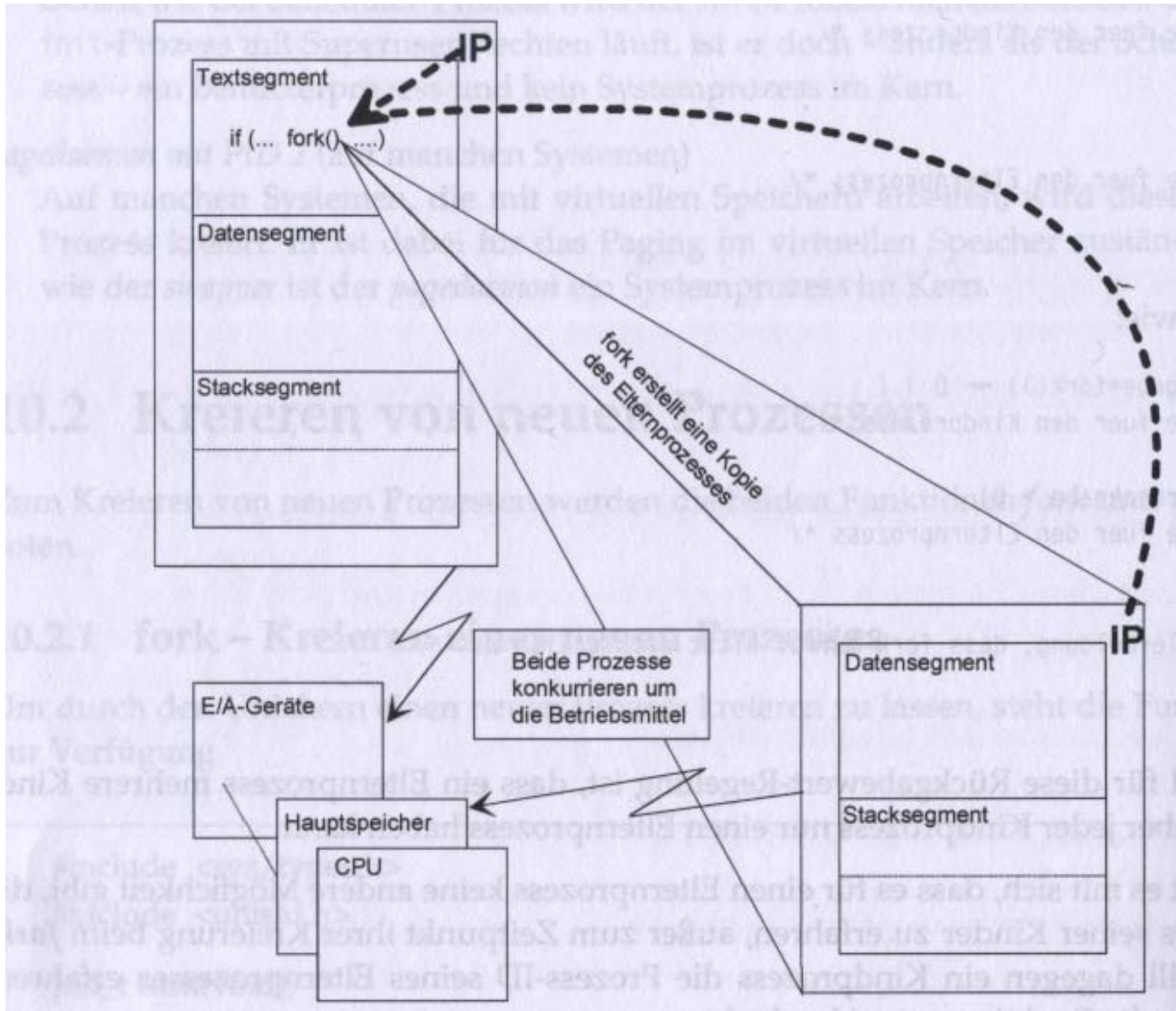
Unix-Prozess-Hierarchie

- Beim Start des Unix-Systems werden einige spezielle Prozesse erzeugt:
 - Scheduler-Prozess mit PID=0
 - Teil des Kerns
 - Wird auch als Swapper bezeichnet
 - init-Prozess mit PID=1
 - wird nach Booten vom Kern kreiert
 - erhält gewöhnlich die PID 1
 - Systemspezifische Initialisierung anhand der Dateien `/etc/rc*`
 - ist Benutzerprozess, kein Systemprozess
 - pagedaemon mit PID=2 (auf manchen Systemen)
 - Systemprozess
 - Zuständig für Paging im virtuellen Speicher

Kreieren von neuen Prozessen

- fork – Kreieren eines neuen Prozesses
 - Durch fork wird eine Kopie des Elternprozesses erstellt
 - Aufrufer von fork ist der Elternprozess
 - Der durch fork kreierte Prozess ist der Kindprozess
 - Nach Aufruf von fork haben Eltern- und Kindprozess
 - die gleichen offenen Dateien,
 - dieselbe User-ID
 - dasselbe Working-Directory
 -
 - fork wird einmal aufgerufen – kehrt aber 2x zurück:
 - Rückkehr zum Kindprozess mit Return-Wert 0
 - Rückkehr zum Elternprozess mit PID-des Kindprozesses als Return-Wert

Kreieren eines Prozesses mit fork



Kreieren von Prozessen mit fork

```
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    pid_t    pid;
    if ( (pid=fork()) < 0)
        /* Fehler bei fork */
    else if (pid == 0)
    {
        /* Code des Kindprozesses */
    }
    else if (pid > 0)
    {
        /* Code des Elternprozesses */
    }
    /* wird von Vater und Kind ausgefuehrt */
    return 0;
}
```

Beispielprogramm zu fork

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int global_var=100;

int
main(void)
{
    int lokal_var;
    pid_t pid;

    lokal_var=1;

    printf("---vor fork-Aufruf---\n");

    switch ( pid=fork() ) {
        case -1:
            printf( "Fehler bei fork\n");
            break;

        case 0:
            lokal_var++;
            global_var++;
            printf(".....Ich bin der Kindprozess.....\n");
            break;

        default:
            printf(".....Ich bin der Elternprozess.....\n");
            break;
    }

    printf("%s: global_var=%d, lokal_var=%d\n",
        (pid==0) ? "Kindprozess" : "Elternprozess", global_var, lokal_var);

    exit(0);
}
```

- Listing:
\$forkdemo
---vor fork-Aufruf---
.....Ich bin der Elternprozess.....
Elternprozess" : global_var=100, lokal_var=1
.....Ich bin der Kindprozess.....
Kindprozess" : global_var=101, lokal_var=2
- Hinweis:
 - die Ausführungsreihenfolge (insbesondere bei mehrfachem Ausführen von Kind und Elternprozess) hängt vom Scheduling-Algorithmus ab!

Mehrfachaufruf von fork

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

int main(void)
{
    char pid[256];
    fork();
    fork();
    fork();
    fork();

    sprintf(pid, "PID = %d\n",
            getpid());
    write(STDOUT_FILENO, pid,
          strlen(pid));
    /* Bei write keine Pufferung */
    exit(0);
}
```

- Listing:
\$mehrfork
PID = 441
PID = 442
PID = 443
PID = 444
PID = 445
PID = 447
PID = 448
PID = 450
PID = 451
PID = 453
PID = 454
PID = 446
PID = 449
PID = 452
PID = 455
PID = 456

Typische Anwendungen für fork

- Ein Programm soll zu einem Zeitpunkt "gleichzeitig" zwei verschiedene Code-Stücke ausführen
 - Bsp. Netzwerk-Server, der auf eine Anforderung eines Clients wartet und bei Eintreffen ein Kindprozess mit `fork` kreiert, der Client-Anforderung bedient
 - Elternprozess begibt sich wieder in Wartezustand
- Ein Prozess (z. B. Shell) möchte ein anderes Programm ausführen
 - dann ruft Kindprozess unmittelbar nach Rückkehr aus `fork` die Funktion `exec` auf

vfork

- Wird Kindprozess kreiert in dem sofort anschließend ein anderes Programm mit `exec` gestartet wird
- Kopie des Adressraums (Datensegment, Stacksegment, Heap) überflüssig
- => Benutzen der Funktion `vfork` anstatt `fork`
- `vfork` kreiert einen Kindprozess, kopiert aber nicht den Adressraum
- Kindprozess benutzt Elternprozessraum so lange mit, bis mit `exec` neues Programm gestartet wird oder Kind mit `exit` beendet wird.
- Bei `vfork` wird garantiert, dass Kindprozess zuerst ausgeführt wird, solange bis `exec` oder `exit` aufgerufen wird
- Achtung: Deadlock möglich!!!

Beispiel zu vfork

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

int global_var=100;

int main(void)
{
    int lokal_var, status;
    pid_t pid;

    lokal_var=1;

    printf("---vor vfork-Aufruf---\n");

    /* auf der nächsten Folie geht's weiter */
```

Beispiel zu vfork (Forts.)

```
switch ( pid=vmfork() ) {
    case -1:
        printf("Fehler bei vmfork\n");
        break;

    case 0:
        lokal_var++;
        global_var++;
        printf(".....Ich bin der Kindprozess.....\n");
        _exit(0);          /* Kindprozess beendet sich */

    default:
        break;
}

wait(&status);          /* Warten auf Ende des Kindprozesses */
printf(".....Ich bin der Elternprozess.....\n");
printf("%s: global_var=%d, lokal_var=%d\n",
        (pid==0) ? "Kindprozess" : "Elternprozess", global_var,
        lokal_var);

exit(0);
}
```

Die exec-Funktion

- Aufruf eines neuen Programmes
- Das aufgerufene Programm beginnt seine Ausführung in seiner main-Funktion
- Es gibt 6 verschiedenen exec-Funktionen

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

```
int execl (const char *pfadname, const char *arg0, /*...*/ ,NULL);
```

```
int execv (const char *pfadname, char *const arg[]);
```

```
int execl (const char *pfadname, const char *arg0, /* ... */ , NULL,  
char *const envp[]);
```

```
int execve (const char *pfadname, *const arg[], char *const envp[]);
```

```
int execlp (const char *dateiname, const char *arg0, /* ... */ , NULL);
```

```
int execvp (const char *dateiname, char *const arg[]);
```

Unterschiede der exec-Funktionen

- l: list
 - Übergabe der Kommandozeilenargumente in Form einer Liste
- v: vektor
 - Übergabe der Kommandozeilenargumente in als Vektor
- p: path
 - Funktion erwartet einen Dateinamen als Parameter
- e: environment
 - Funktion erwartet eine Environmentliste und benutzt nicht die Aktuellen Umgebungsparameter

Beispiele für exec-Aufrufe

```
if ( (pid=fork()) < 0)
    /*fehler*/;
else if (pid == 0) {
    if (execle("/home/hh/sysprog/kap10/arg_env",
              "arg_env", "Hallo", "", "Welt", NULL, neu_env) < 0)
        /*fehler*/;
}
if (waitpid(pid, NULL, 0) < 0)
    /*fehler*/;

/*----- Demo zu execlp -----*/
if ( (pid=fork()) < 0)
    /*fehler*/;
else if (pid == 0) {
    if (execlp("arg_env", "arg_env", "eins", "zwei", "drei",
              NULL) < 0)
        /*fehler*/;
}
```