

# U3 3. Übung

---

- Besprechung 1. Aufgabe
- Infos zur Aufgabe 3: fork, exec
- Fehlersuche mit valgrind

# U3-1 Aufgabe 1

- Vorstellung einer Lösung
- Fehlerbehandlung nicht vergessen!

```
e = (struct listelement*) malloc(sizeof(struct listelement));  
if (e == NULL) {  
    perror("Kann Listenelement nicht anlegen.");  
    exit(EXIT_FAILURE);  
}
```

- Fehlermeldungen immer auf `stderr` ausgeben!

z.B. mit `fprintf`

```
fprintf(stderr, "%s(%d): %s\n", __FILE__, __LINE__, strerror(errno));
```

oder mit `perror`

```
perror("Beschreibung wobei");
```

## U3-2 Hinweise zur 3. Aufgabe

---

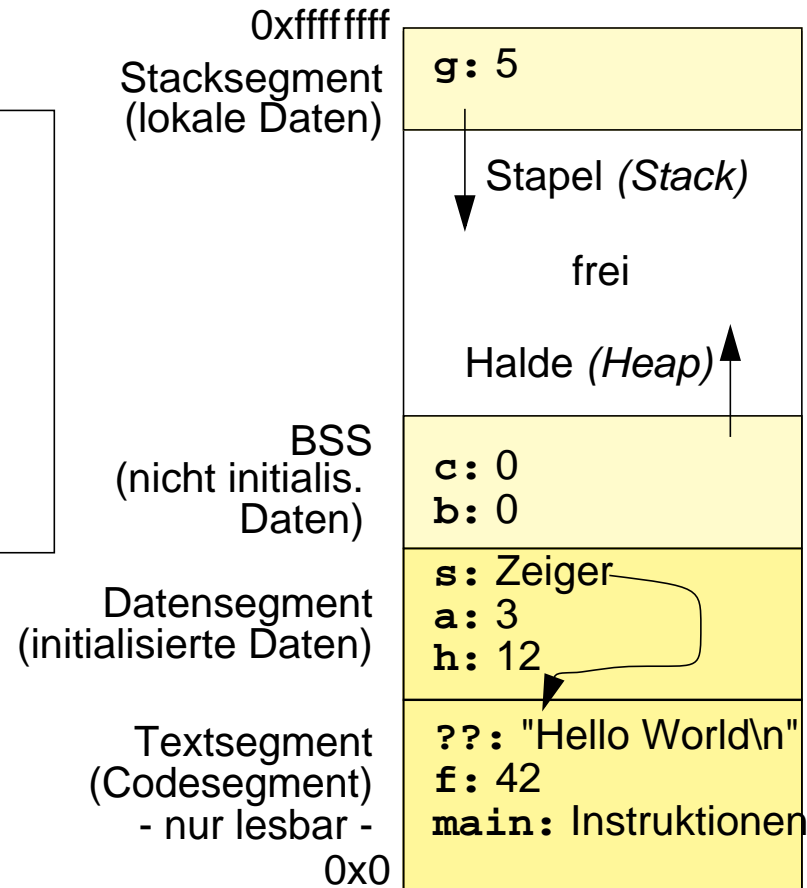
- Speicheraufbau eines Prozesses
- Prozesse
- fork, exec
- exit
- wait

# 1 Speicheraufbau eines Prozesses (UNIX)

## ■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```



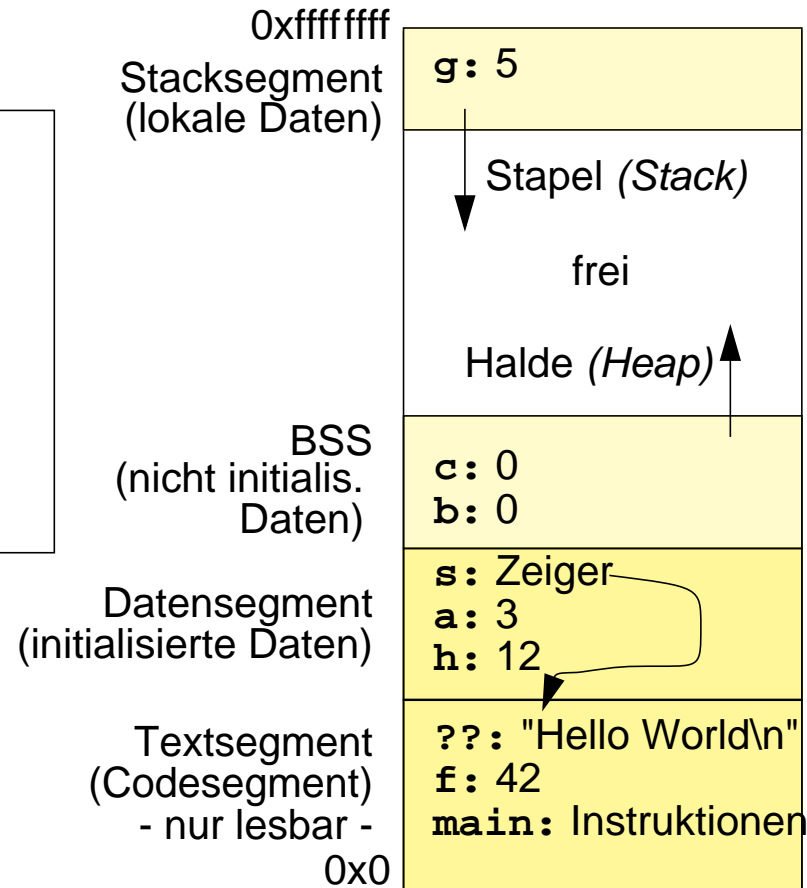
# 1 Speicheraufbau eines Prozesses (UNIX)

## ■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
s[1]= 'a';
f= 2;
```



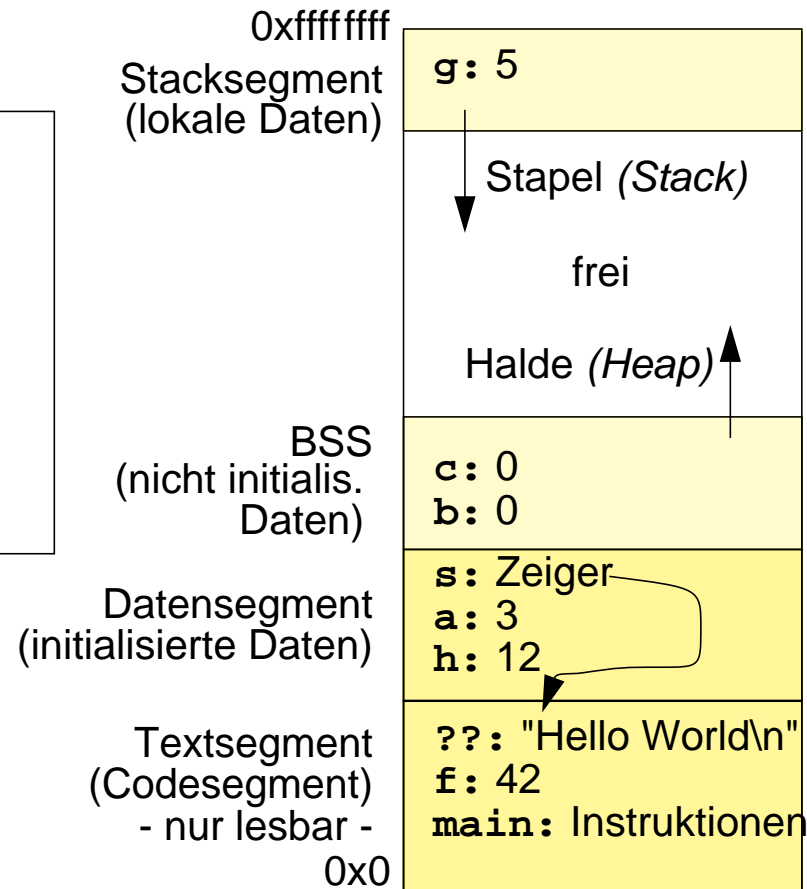
# 1 Speicheraufbau eines Prozesses (UNIX)

## ■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
s[1]= 'a'; /* cc error */
f= 2;     /* cc error */
```



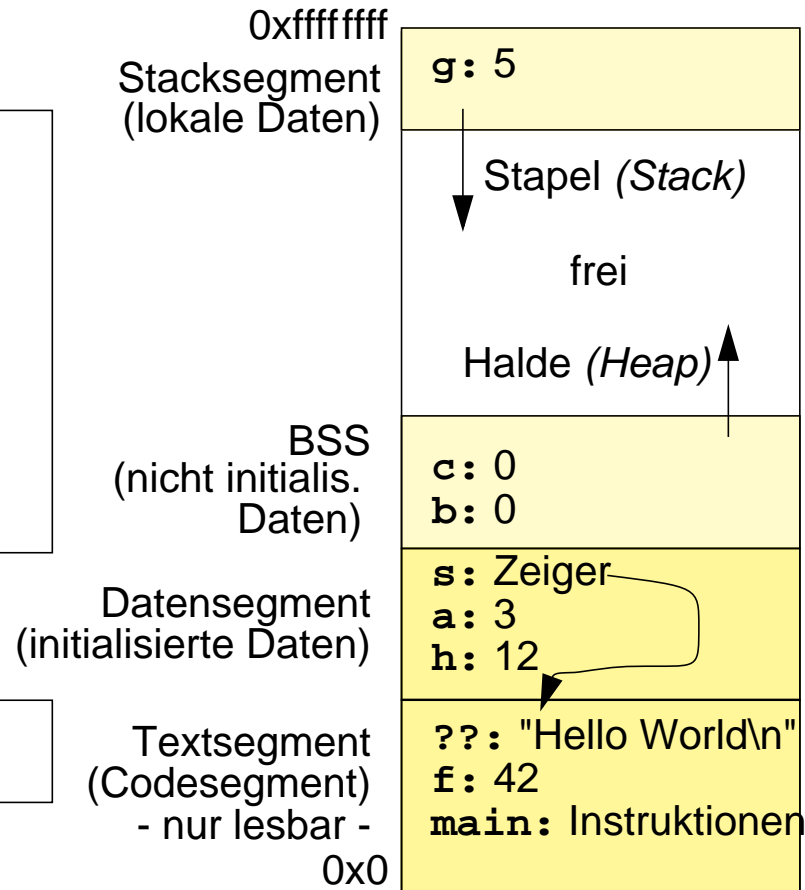
# 1 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
((char*)s)[1]= 'a';
*((int *)&f)= 2;
```



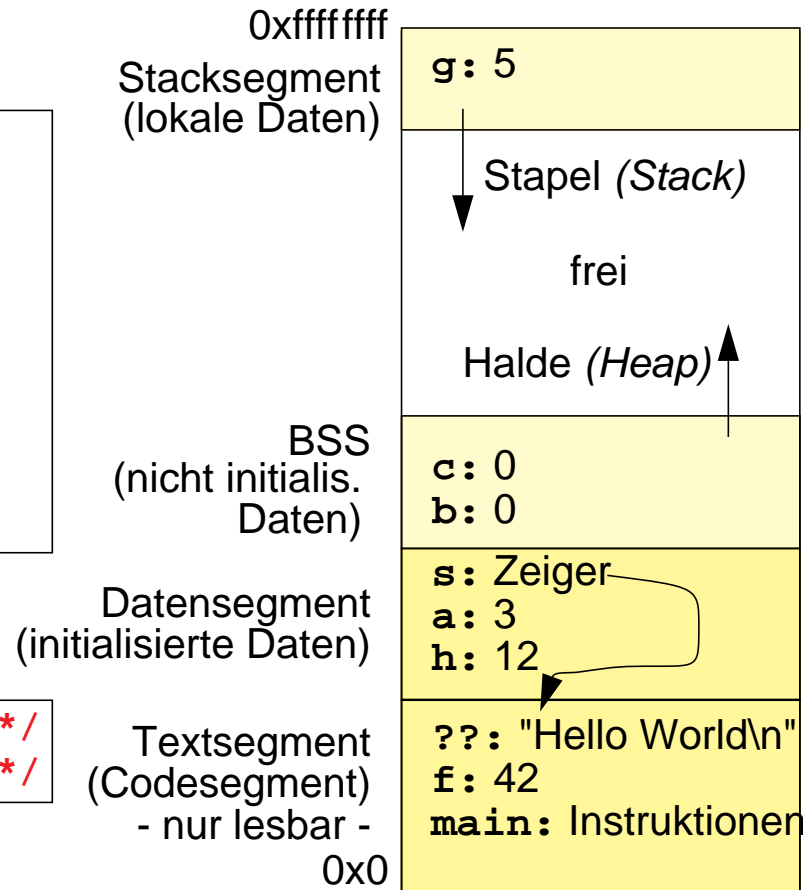
# 1 Speicheraufbau eines Prozesses (UNIX)

■ Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
static int a=3, b, c=0;
const int f=42;
const char *s="Hello World\n";

int main( ... ) {
    int g=5;
    static int h=12;
}
```

```
((char*)s)[1]= 'a'; /* SIGSEGV */
*((int *)&f)= 2; /* SIGSEGV */
```





## U3-3 fork

---

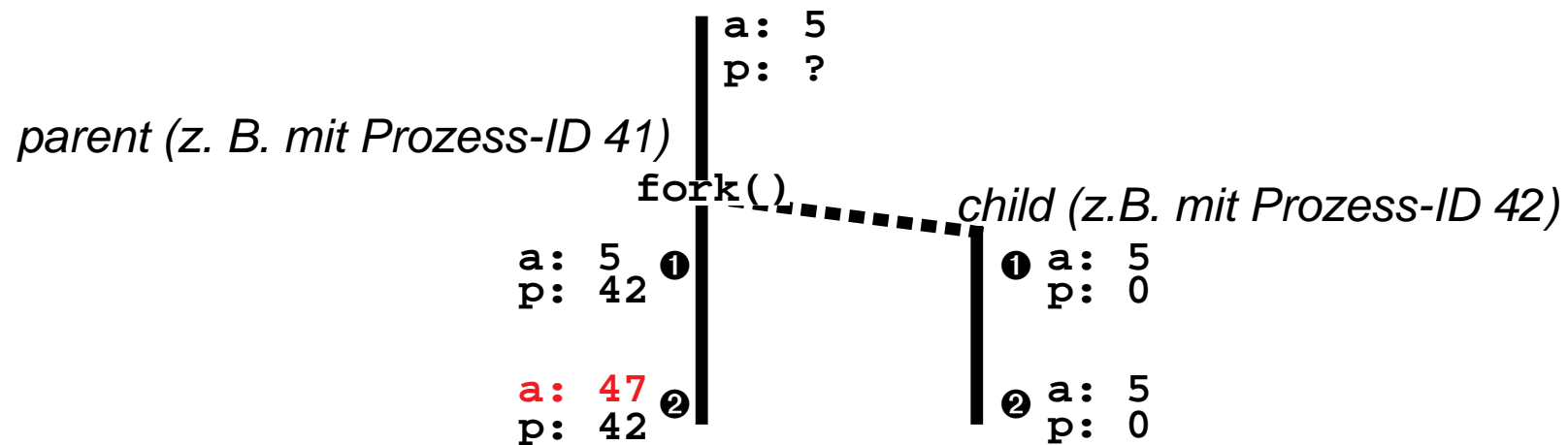
- Vererbung von
  - ◆ Datensegment (neue Kopie, gleiche Daten)
  - ◆ Stacksegment (neue Kopie, gleiche Daten)
  - ◆ Textsegment (gemeinsam genutzt, da nur lesbar)
  - ◆ Filedeskriptoren (geöffnete Dateien)
  - ◆ Arbeitsverzeichnis
  - ◆ Benutzer- und Gruppen-ID (uid, gid)
  - ◆ Umgebungsvariablen
  - ◆ Signalbehandlung
  - ◆ ...
  
- Neu:
  - ◆ Prozess-ID

# U3-3 fork

```

int a; pid_t p;
a = 5;
p = fork();
  ①
a += p;    ②
if (p == 0) {
    ...
} else {
    ...
}

```



## U3-4 exec

---

- Lädt Programm zur Ausführung in den aktuellen Prozess
- ersetzt Text-, Daten- und Stacksegment
- behält: Filedeskriptoren (= geöffnete Dateien), Arbeitsverzeichnis, ...
  - Vererbung von stdin, stdout und stderr!
- Aufrufparameter:
  - ◆ Dateiname des neuen Programmes (z.B. `"/bin/cp"`)
  - ◆ Argumente, die der `main`-Funktion des neuen Programms übergeben werden (z.B. `"cp"`, `"/etc/passwd"`, `"/tmp/passwd"`)
  - ◆ evtl. Umgebungsvariablen

- Beispiel

```
execl("/bin/cp", "cp", "/etc/passwd", "/tmp/passwd", NULL);
```

## U3-4 exec Varianten

- mit Angabe des vollen Pfads der Programm-Datei in `path`

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

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

- mit Umgebungsvariablen in `envp`

```
int execlp(const char *path, char *const arg0, ... , const char
           *argn, char * /*NULL*/, char *const envp[]);
```

```
int execve (const char *path, char *const argv[], char *const
            envp[]);
```

- zum Suchen von `file` wird die Umgebungsvariable `PATH` verwendet

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

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

## U3-5 exit

---

- beendet aktuellen Prozess
- gibt alle Ressourcen frei, die der Prozess belegt hat, z.B.
  - ◆ Speicher
  - ◆ Filedesriptoren (schließt alle offenen Files)
  - ◆ Kerndaten, die für die Prozessverwaltung verwendet wurden
- Prozess geht in den *Zombie*-Zustand über
  - ◆ ermöglicht es dem Vater auf den Tod des Kindes zu reagieren (wait)

## U3-6 wait

- warten auf Statusinformationen von Kind-Prozessen (Rückgabe: PID)

- ◆ `wait(int *status)`

- ◆ `waitpid(pid_t pid, int *status, int options)`

- Beispiel:

```
int main(int argc, char *argv[]) {
    pid_t pid;
    if ((pid=fork()) > 0) {
        /* parent */
        int status;
        wait(&status); /* ... Fehlerabfrage */
        printf("Kindstatus: %x", status); /* nackte Status-Bits ausg. */
    } else if (pid == 0) {
        /* child */
        execl("/bin/cp", "cp", "/etc/passwd", "/tmp/passwd", 0);
        /* diese Stelle wird nur im Fehlerfall erreicht */
    } else {
        /* pid == -1 --> Fehler bei fork */
    }
}
```

## U3-7 wait

---

- `wait` blockiert den aufrufenden Prozess so lange, bis ein Kind-Prozess im Zustand "terminiert" existiert oder ein Kind-Prozess gestoppt wird
  - ◆ `pid` dieses Kind-Prozesses wird als Ergebnis geliefert
  - ◆ als Parameter kann ein Zeiger auf einen `int`-Wert mitgegeben werden, in dem der Status (16 Bit) des Kind-Prozesses abgelegt wird
  - ◆ in den Status-Bits wird eingetragen "was dem Kind-Prozess zugestossen ist", Details können über Makros abgefragt werden:
    - Prozess "normal" mit `exit()` terminiert: `WIFEXITED(status)`
    - exit-Parameter (nur das unterste Byte): `WEXITSTATUS(status)`
    - Prozess durch Signal abgebrochen: `WIFSIGNALED(status)`
    - Nummer des Signals, das Abbruch verursacht hat: `WTERMSIG(status)`
    - Prozess wurde gestoppt: `WIFSTOPPED(status)`
    - Prozess hat core-dump geschrieben: `WCOREDUMP(status)`
    - weitere siehe `man 2 wait` bzw. `man wstat` (je nach System)

## U3-8 valgrind

---

- Baukasten von Debugging- und Profiling-Werkzeugen (ausführbarer Code wird durch synthetische CPU auf Softwareebene interpretiert → Ausführung erheblich langsamer!)
  - ◆ Memcheck: erkennt Speicherzugriff-Probleme
    - Nutzung von nicht-initialisiertem Speicher
    - Zugriff auf freigegebenen Speicher
    - Zugriff über das Ende von allokierten Speicherbereichen
    - Zugriff auf ungültige Stack-Bereiche
    - ...
  - ◆ Helgrind: erkennt Koordinierungsprobleme zwischen mehreren Threads
    - siehe Aufgabe 8
    - in valgrind 3.1.X nicht verfügbar
  - ◆ Cachegrind: zur Analyse des Cache-Zugriffsverhaltens eines Programms
- Aufrufbeispiel: `valgrind --tool=memcheck wsort` oder `valgrind wsort`