U3 3. Übung

- Besprechung 1. Aufgabe
- Fehlerbehandlung
- Speicheraufbau eines Prozesses
- Prozesse: fork, exec, wait
- Aufgabe 3

U3-1 Fehlerbehandlung

- Fehler können aus unterschiedlichsten Gründen im Programm auftreten
  - ➤ Systemressourcen erschöpft
    ➠ **malloc(3)** schlägt fehl
  - ➤ Fehlerhafte Benutzereingaben (z.B. nicht existierende Datei)
    ➠ **open(2)** schlägt fehl
  - ➤ Transiente Fehler (z.B. nicht erreichbarer Server)
    ➠ **connect(2)** schlägt fehl
  - ➤ ...
- Gute Software **erkennt Fehler**, führt eine **angebrachte Behandlung** durch und gibt eine **aussagekräftige Fehlermeldung** aus
  - ◆ Kann das Programm trotz des Fehlers sinnvoll weiterlaufen?
  - ◆ Beispiel 1: Ermittlung des Hostnamens zu einer IP-Adresse für Logeintrag
    ➠ Fehlerbehandlung: IP-Adresse im Log eintragen, Programm läuft weiter
  - ◆ Beispiel 2: Öffnen einer zu kopierenden Datei schlägt fehl
    ➠ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden

1 Fehler in Bibliotheksfunktionen

- Fehler treten häufig in Funktionen der C-Bibliothek auf
  - ➤ erkennbar i.d.R. am Rückgabewert (Manpage!)
- Die Fehlerursache wird über die globale Variable **`errno`** übermittelt
  - ➤ Fehlercode für jeden möglichen Fehler (siehe **errno(3)**)
  - ➤ Der Wert **`errno=0`** bedeutet Erfolg, alles andere ist ein Fehlercode
  - ➤ Bibliotheksfunktionen setzen **`errno`** im Fehlerfall
  - ➤ Bekanntmachung im Programm durch Einbinden von **`errno.h`**
- Fehlercodes können mit den Funktionen **perror(3)** und **strerror(3)** ausgegeben bzw. in lesbare Strings umgewandelt werden

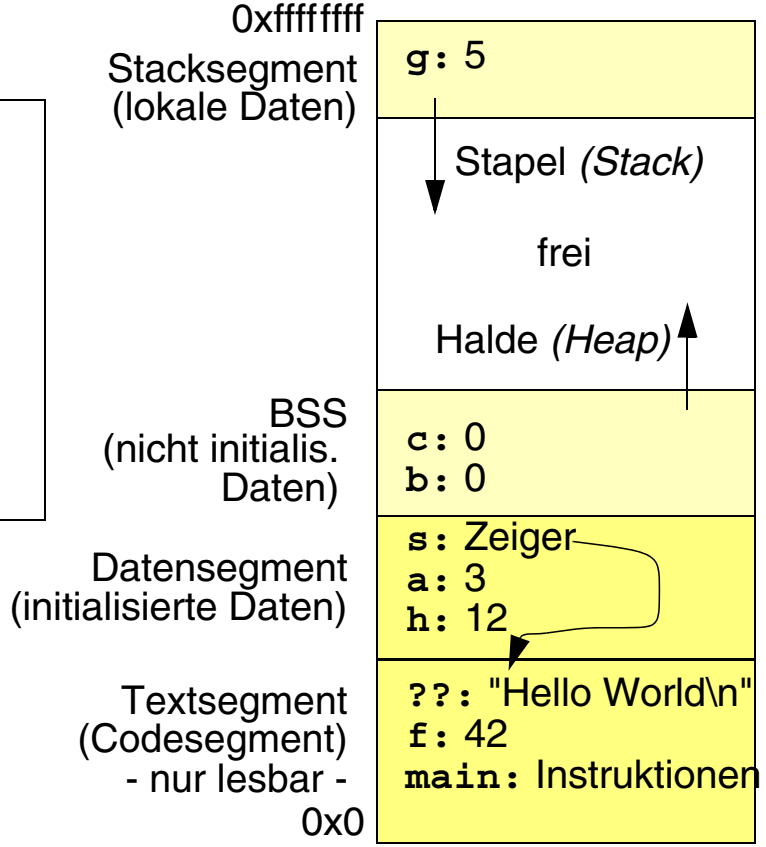
```
char *mem = malloc(...); /* malloc gibt im Fehlerfall */
if(NULL == mem) {        /* NULL zurück */
    fprintf(stderr, "%s:%d: malloc failed with reason: %s\n",
        __FILE__, __LINE__-3, strerror(errno));
    perror("malloc"); /* Alternative zu strerror + fprintf */
    exit(EXIT_FAILURE); /* Programm mit Fehlercode beenden */
}
```

U3-2 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;
}
```



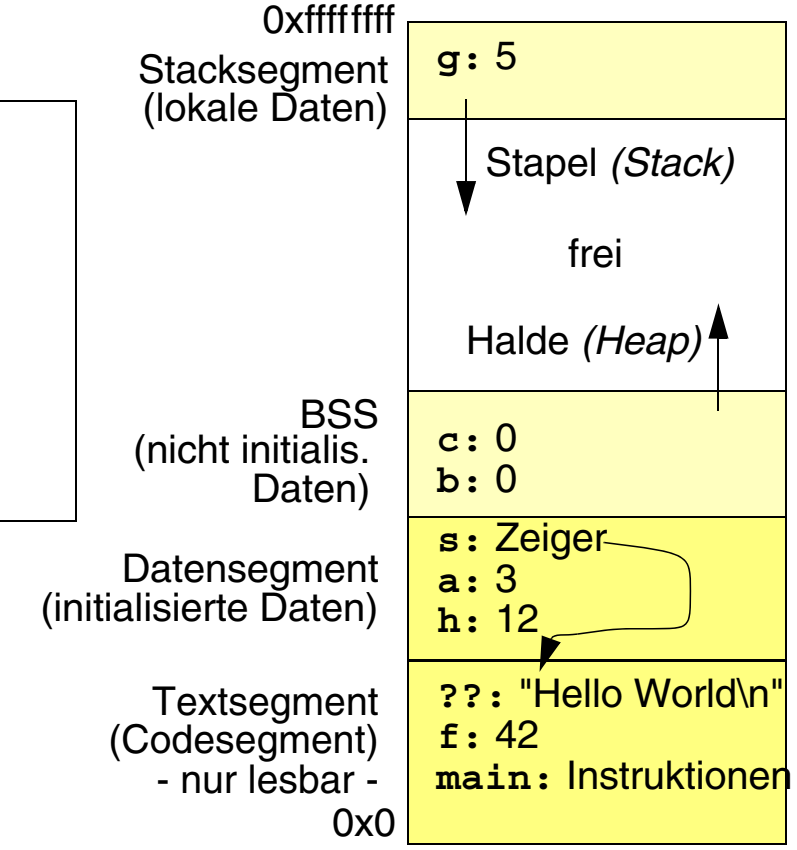
U3-2 Speicheraufbau eines Prozesses (UNIX)

- 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;
```



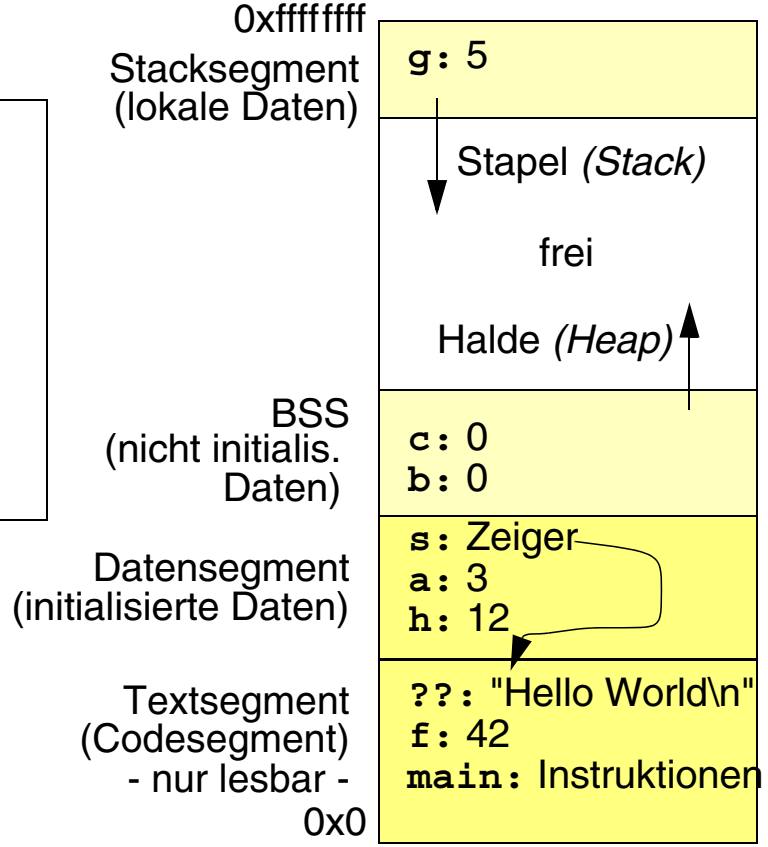
U3-2 Speicheraufbau eines Prozesses (UNIX)

- 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 */
```



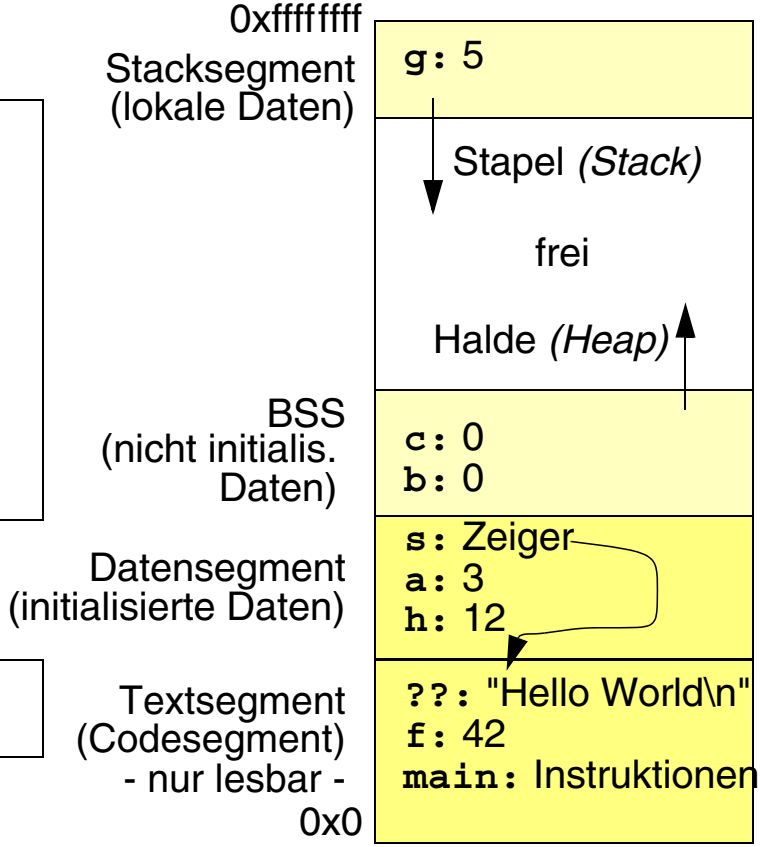
U3-2 Speicheraufbau eines Prozesses (UNIX)

- 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;
```



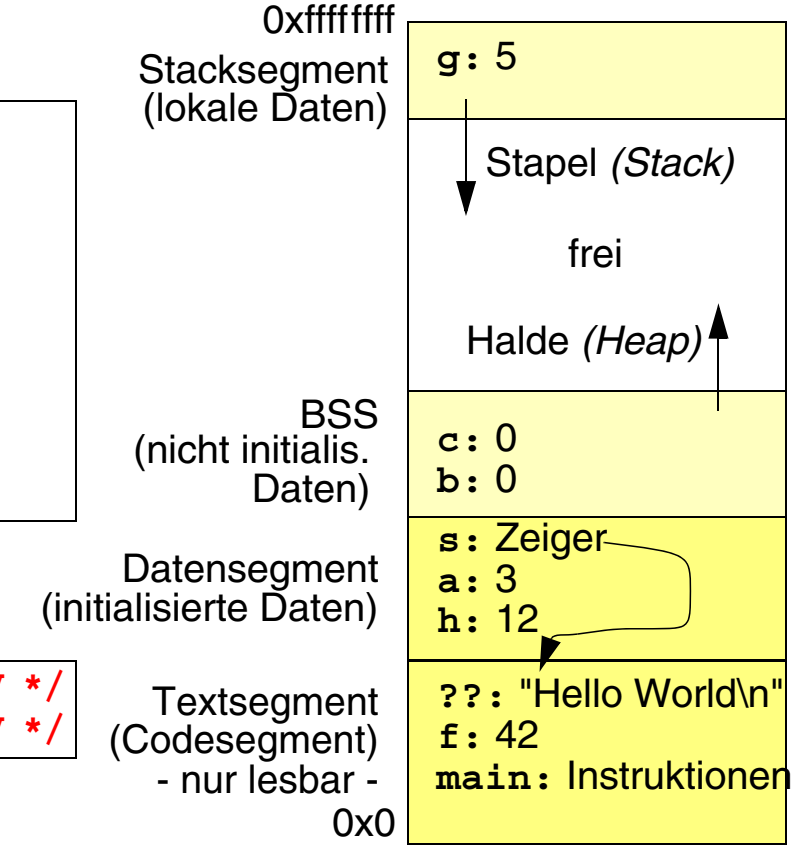
U3-2 Speicheraufbau eines Prozesses (UNIX)

- 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 */
```

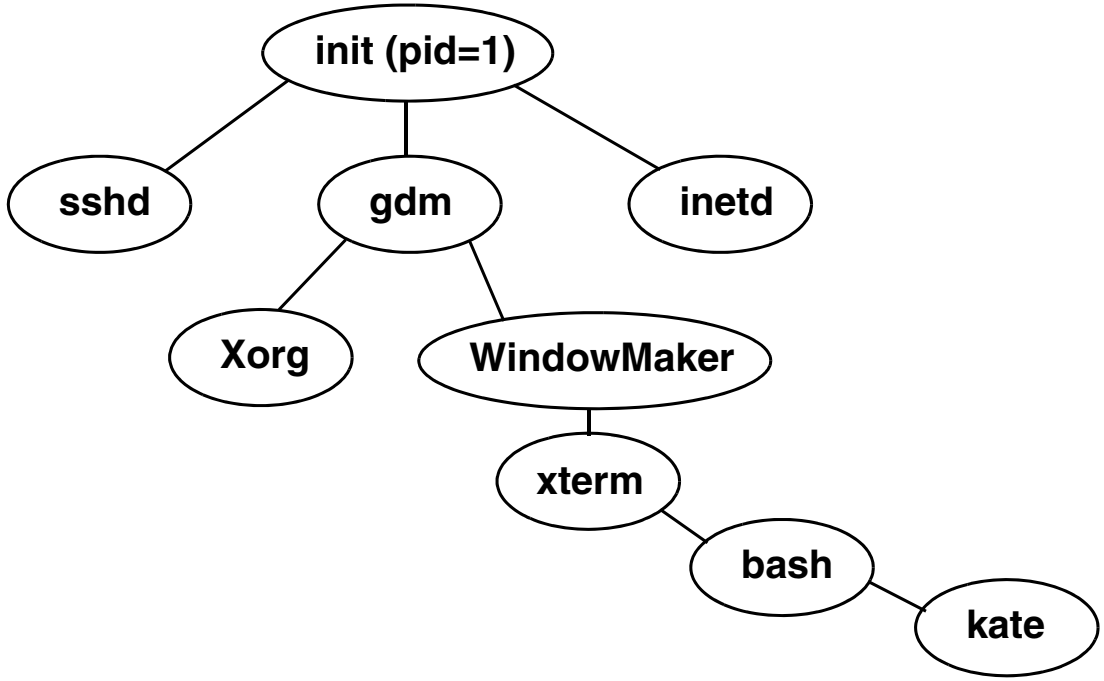


U3-3 Prozesse: Überblick

- Prozesse sind eine Ausführumgebung für Programme
  - ➤ haben eine Prozess-ID (PID, ganzzahlig positiv)
  - ➤ führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft (s. Vorlesung ab **A 3-6**)

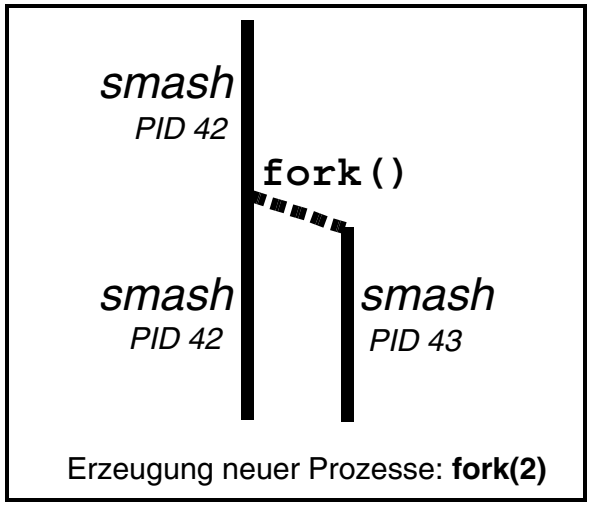
U3-3 UNIX-Prozesshierarchie

- Zwischen Prozessen bestehen Vater-Kind-Beziehungen
  - der erste Prozess wird direkt vom Systemkern gestartet (z.B. System V *init*)
  - es entsteht ein Baum von Prozessen bzw. eine Prozesshierarchie

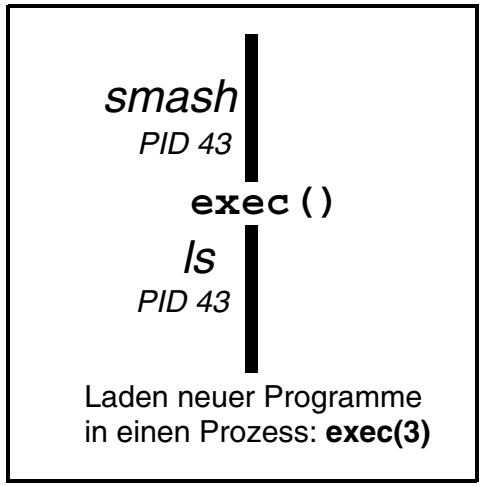


  - Beispiel: **kate** ist ein Kind von **bash**, **bash** wiederum ein Kind von **xterm**

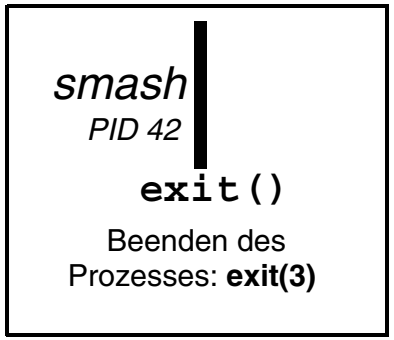
U3-4 POSIX Prozess-Systemfunktionen



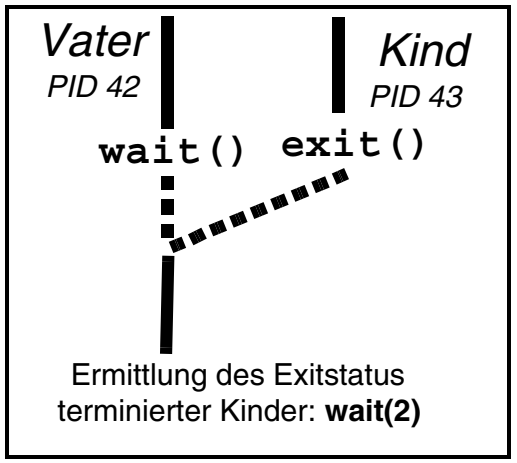
Erzeugung neuer Prozesse: **fork(2)**



Laden neuer Programme in einen Prozess: **exec(3)**



Beenden des Prozesses: **exit(3)**



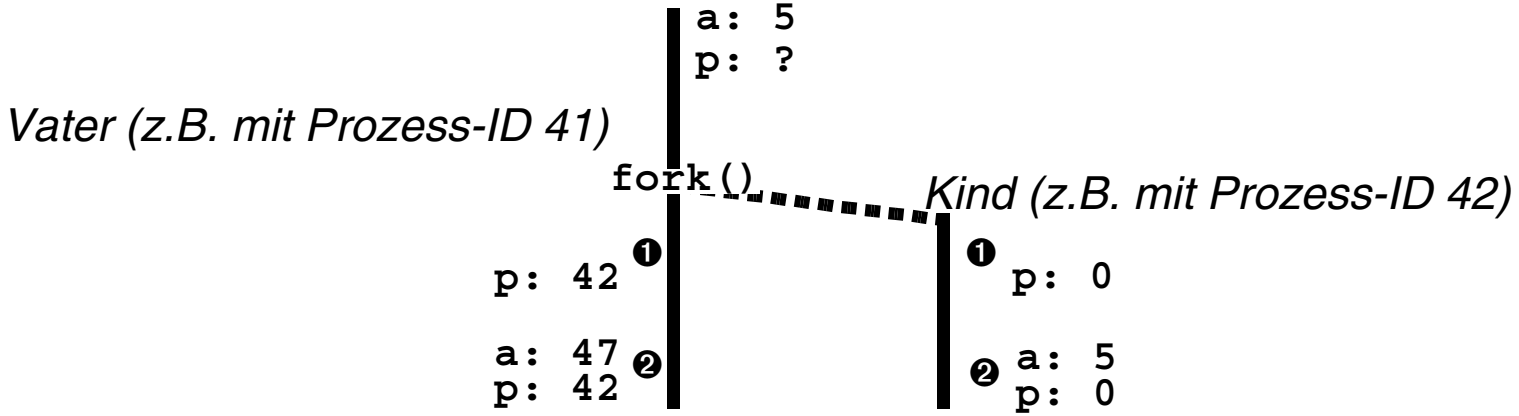
Ermittlung des Exitstatus terminierter Kinder: **wait(2)**

1 fork(2): Erzeugung eines neuen Prozesses

- Erzeugt einen neuen Kindprozess
- Exakte Kopie des Vaters...
  - Datensegment (neue Kopie, gleiche Daten)
  - Stacksegment (neue Kopie, gleiche Daten)
  - Textsegment (gemeinsam genutzt, da nur lesbar)
  - Filedeskriptoren (geöffnete Dateien)
  - ...
- ...mit Ausnahme der Prozess-ID
- Kind startet Ausführung hinter dem fork() mit dem geerbten Zustand
  - ➤ das ausgeführte Programm muss anhand der PID (Rückgabewert von **fork()**) entscheiden, ob es sich um den Vater- oder den Kindprozess handelt

1 fork(2): Beispiel

```
int a=5; pid_t p = fork();❶
a += p; ❷
switch(p) {
    case -1: /* fork-Fehler, es wurde kein Kind erzeugt */
        ...
    case 0: /* Hier befinden wir uns im Kind */
        ...
    default: /* Hier befinden wir uns im Vater */
        ...
}
```



2 exec(3)

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

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

- exec kehrt nur **im Fehlerfall** zurück

2 exec(3) 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[]);
```

- 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[]);
```

3 exit(3)

- beendet aktuellen Prozess mit einem Status-Byte
  - ➤ Konvention: Status 0 bedeutet Erfolg, alles andere eine Fehlernummer
  - ➤ Bedeutung der Exitstatus üblicherweise in Manpage dokumentiert
  - ➤ Exitstatus `EXIT_FAILURE` und `EXIT_SUCCESS` vordefiniert
- gibt alle Ressourcen frei, die der Prozess belegt hat, z.B.
  - ◆ Speicher
  - ◆ Filedeskriptoren (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(2)**)
  - ◆ Zombie-Prozesse belegen Systemressourcen und sollten schnellstmöglich beseitigt werden!
  - ◆ ist der Vater schon vor dem Kind terminiert, so wird der Zombie an den Prozess mit PID 1 (z.B. *init*) weitergereicht, welcher diesen sofort beseitigt

4 wait(2)

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

```
pid_t wait(int *status);
```

- Beispiel:

```
int main(int argc, char *argv[]) {
  pid_t pid;
  if ((pid=fork()) > 0) { /* Vater */
    int status;
    wait(&status); /* Fehlerbehandlung nicht vergessen! */
    printf("Kindstatus: %x", status); /* nackte Status-Bits ausg.*/

  } else if (pid == 0) { /* Kind */
    execl("/bin/cp","/bin/cp","x.txt","y.txt", NULL);
    /* diese Stelle wird nur im Fehlerfall erreicht */
    perror("exec /bin/cp"); exit(EXIT_FAILURE);
  } else {
    /* pid == -1 --> Fehler bei fork */
  }
}
```

4 wait(2)

- **`wait`** blockiert den Vater, bis ein Kind terminiert oder gestoppt wird
  - *pid* dieses Kind-Prozesses wird als Ergebnis geliefert
  - als Parameter kann ein Zeiger auf einen *int*-Wert mitgegeben werden, in dem der Exitstatus (**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 mit exit() terminiert: **`WIFEXITED(status)`**
      - exit-Parameter (unteres Byte): **`WEXITSTATUS(status)`**
    - ➤ Prozess durch Signal abgebrochen: **`WIFSIGNALED(status)`**
      - Nummer des Signals: **`WTERMSIG(status)`**
  - weitere siehe `man 2 wait`

5 waitpid(2)

- Mächtigere Variante von wait(2)

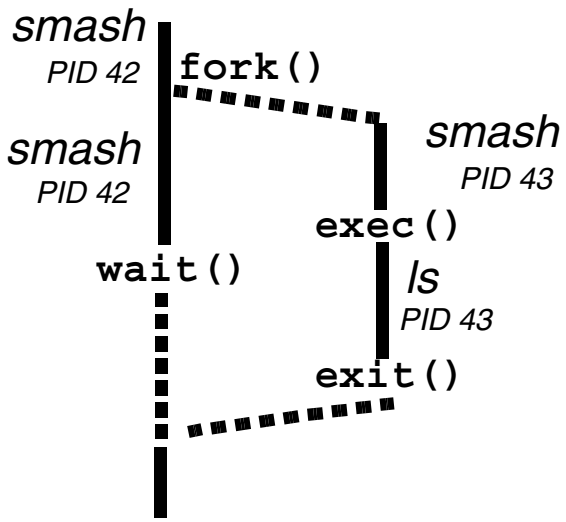
```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Wartet auf Statusänderung
  - *bestimmten* Prozesses: `pid>0`
  - *beliebigen* Kindprozesses: `pid==-1`
  - eines Prozesses derselben Prozessgruppe: `pid==0`
  - eines Prozesses einer *bestimmten* Prozessgruppe: `pid<-1`
- Verhalten mit *Optionen* anpassbar
  - `WNOHANG`: **waitpid** kehrt sofort zurück, wenn kein passender Zombie verfügbar ist
  - eignet sich zum Polling nach Zombieprozessen

U3-5 Aufgabe 3: Einfache Shell im Eigenbau

1 Funktionsweise

- Eingabezeile, aus der der Benutzer Programme starten kann



- Erzeugt einen neuen Prozess und startet in diesem das Programm
- Wartet auf Ende des Prozesses und gibt dann dessen Exitstatus aus

2 Aufteilung der Kommandozeile

- Anzahl der Kommandoparameter beim Programmieren
  - ➤ gibt der Benutzer mit der Eingabe vor
  - ➤ können von Kommando zu Kommando unterschiedlich sein
  - ➤ die l-Varianten von exec können nicht verwendet werden
- Die v-Varianten von exec erhalten ein Argumentenarray als Parameter
  - ➤ dieses kann dynamisch konstruiert werden
  - ➤ hierzu muss die Kommandozeile in aufgeteilt werden (Trenner '\t' und ' ')
  - ➤ das Argumentenarray ist ein Feld von Zeigern auf die einzelnen Token
  - ➤ terminiert mit einem `NULL`-Zeiger
- Zum Aufteilen der Kommandozeile kann die Funktion **strtok(3)** benutzt werden

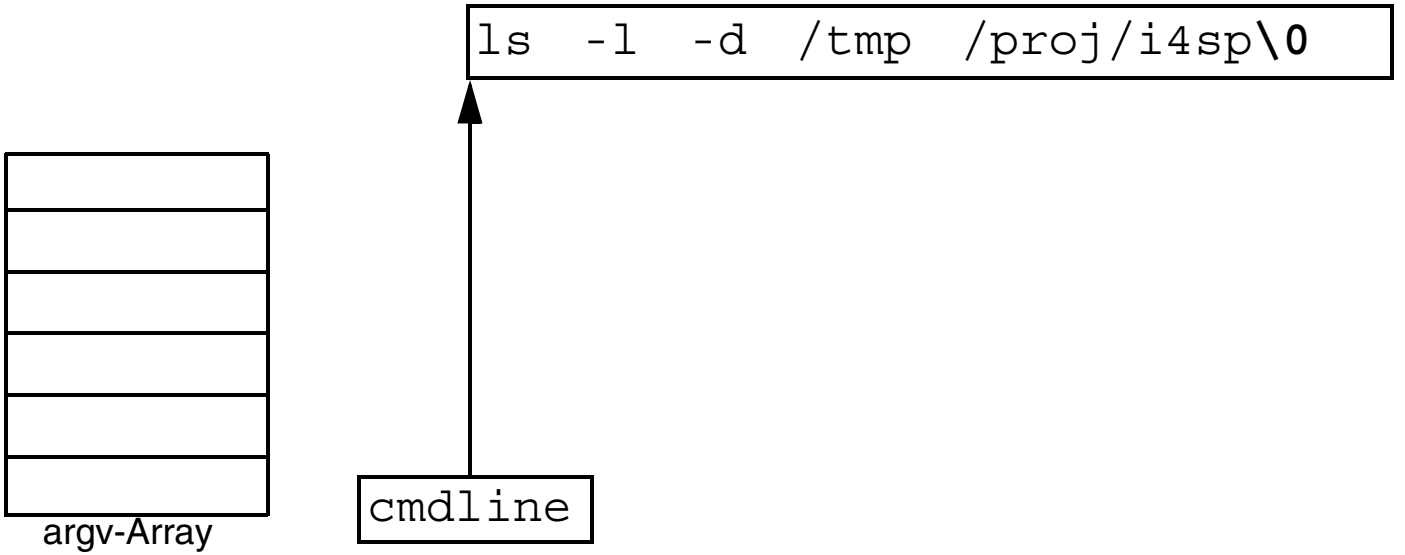
2 strtok

- **strtok(3)** teilt einen String in *Tokens* auf, die durch bestimmte Trennzeichen getrennt sind

```
char *strtok(char *str, const char *delim);
```

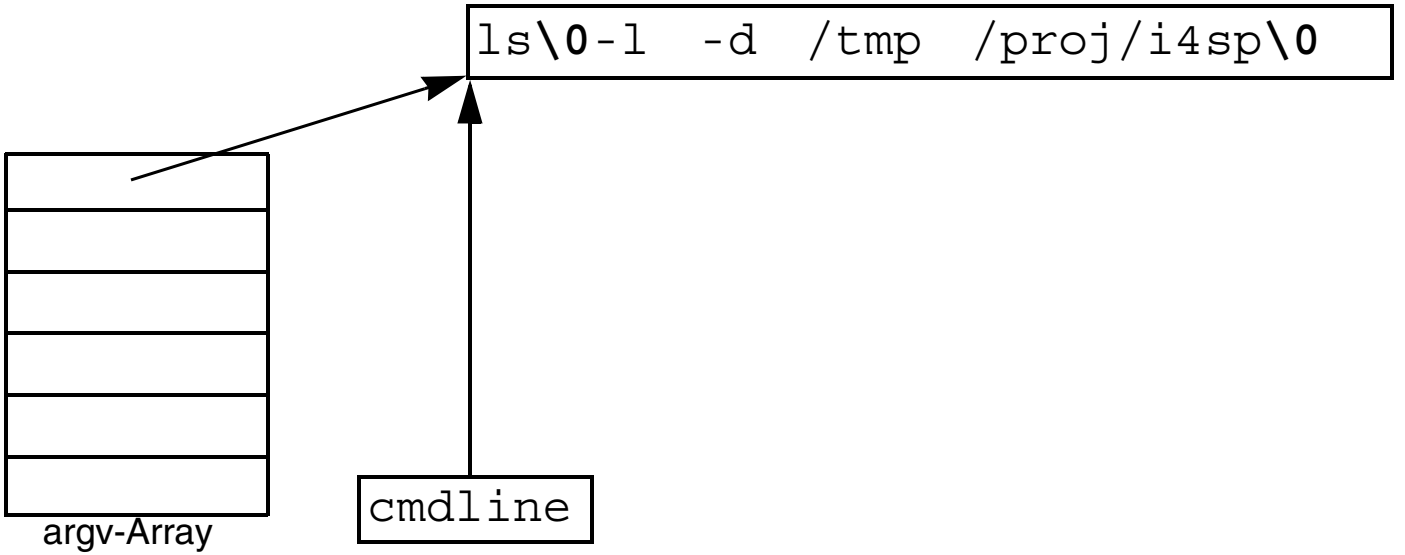
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token (mehrere aufeinanderfolgende Trennzeichen werden hierbei übersprungen)
  - `str` ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen `NULL`
  - `delim` ist ein String, der alle Trennzeichen enthält, z.B. `" \t\n"`
- Bei jedem Aufruf wird das einem Token folgende Trennzeichen durch `'\0'` ersetzt
- Ist das Ende des Strings erreicht, gibt **strtok** `NULL` zurück

2 strtok-Beispiel



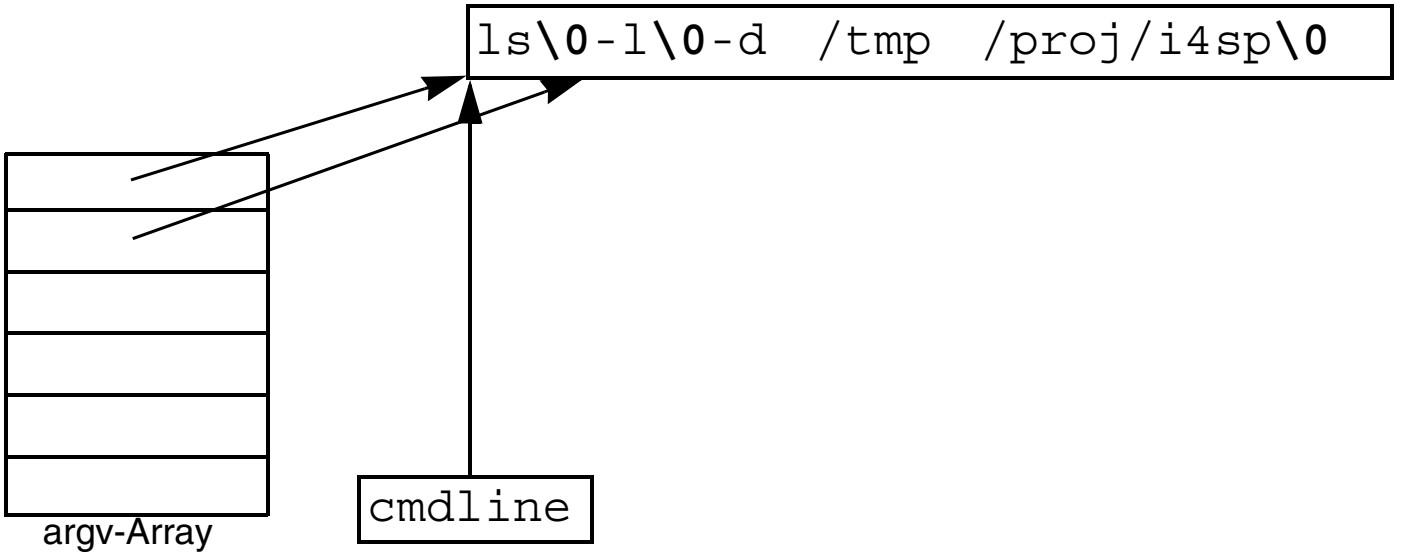
- Kommandozeile befindet sich als `'\0'`-terminierter String im Speicher

2 strtok-Beispiel



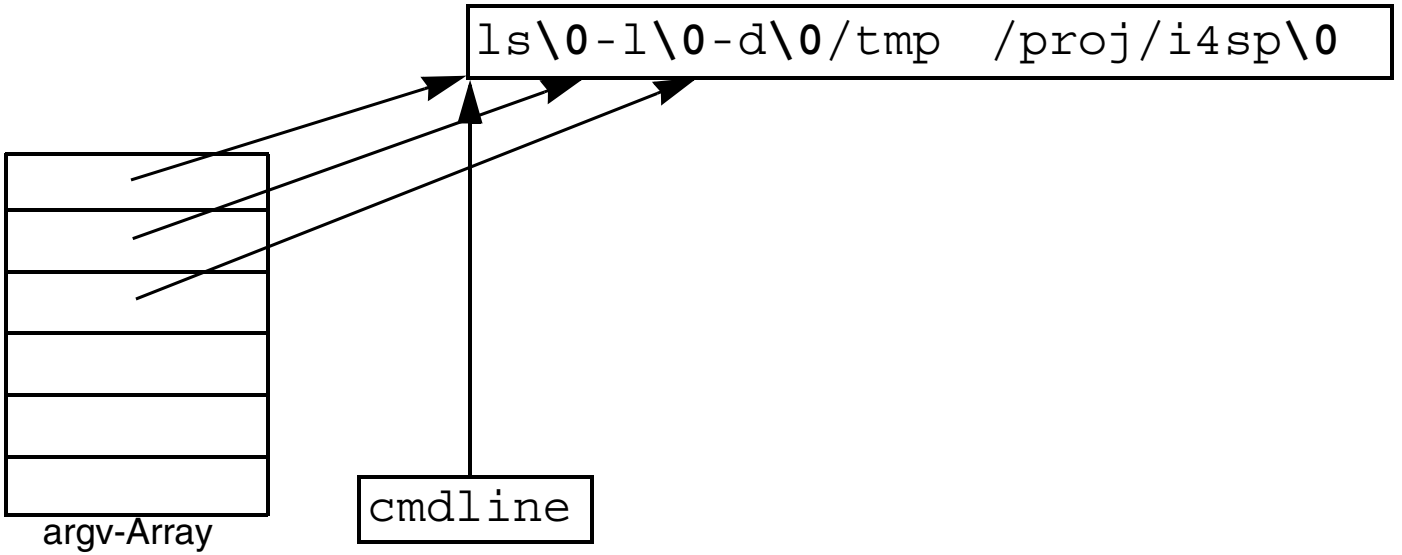
- Erster **strtok**-Aufruf mit dem Zeiger auf diesen Speicherbereich
- **strtok** liefert Zeiger auf erstes Token *ls* und ersetzt den Folgetrenner mit `'\0'`

2 strtok-Beispiel



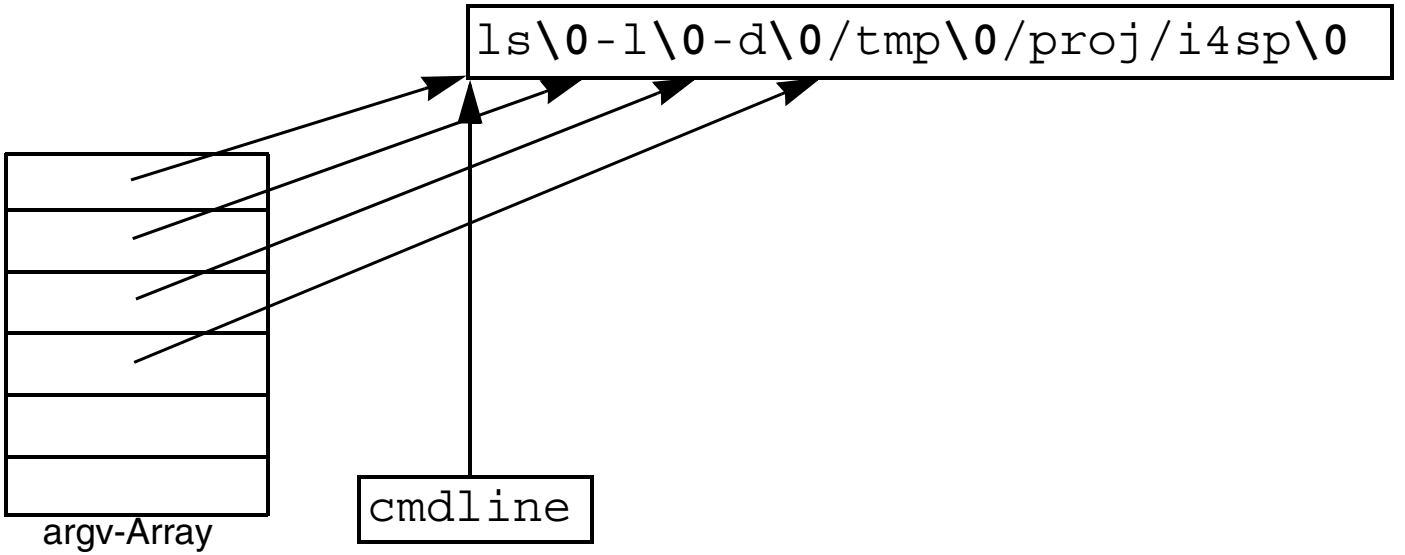
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



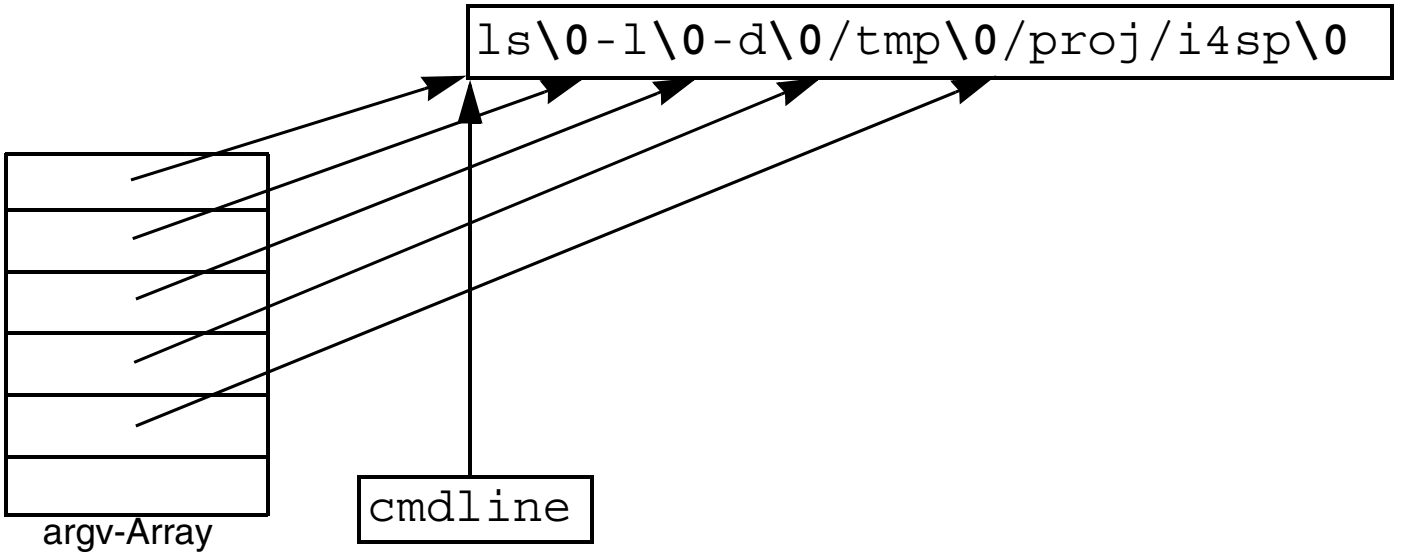
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



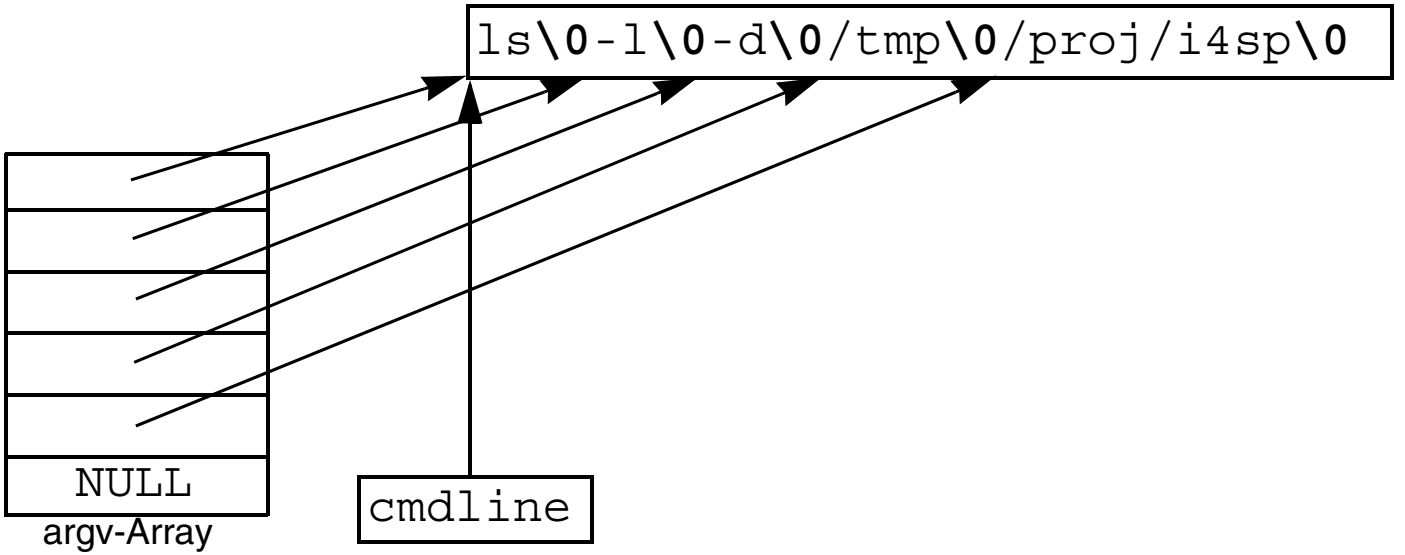
- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- **strtok** liefert jeweils Zeiger auf das nächste Token

2 strtok-Beispiel



- Weitere Aufrufe von **strtok** nun mit einem `NULL`-Zeiger
- Am Ende liefert **strtok** `NULL` und das argv-Array hat die nötige Form

3 Ermitteln von Systemlimits

- Funktion **sysconf(3)**

```
long sysconf(int name);
```

- Abfrage von Konfigurationsoptionen des Betriebssystems, z.B.
  - _SC_ARG_MAX: Maximale Länge der Kommandozeile für **exec(3)**
  - _SC_LINE_MAX: Maximale Länge einer Eingabezeile (**stdin** oder Datei)
- Abfrage zur Laufzeit durch Aufruf von sysconf mit Options-ID (s. oben)
- Alternativ Verwendung von Makros zur Übersetzungszeit
  - ARG_MAX
  - LINE_MAX