

U7 POSIX-Prozesse

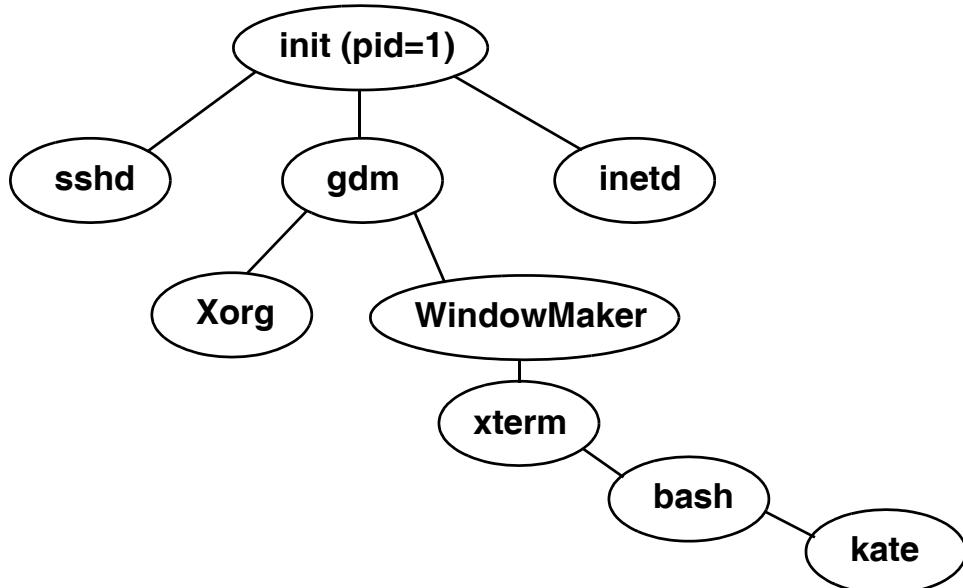
- Prozesse
- POSIX-Prozess-Systemfunktionen
- Aufgabe 7

U7-1 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 J)

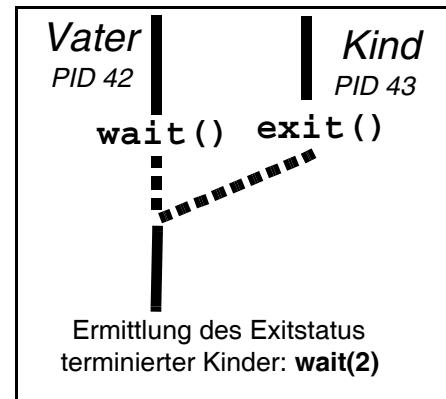
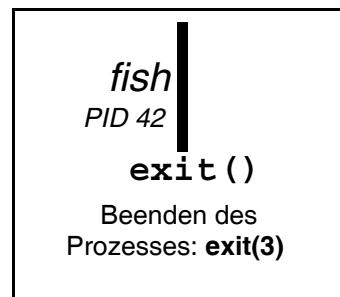
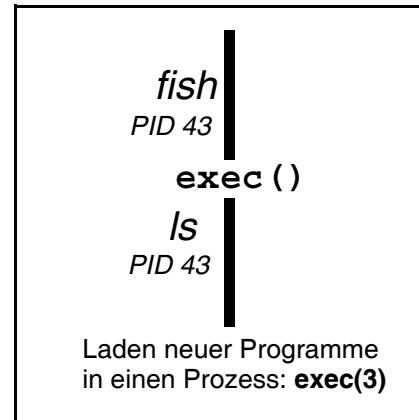
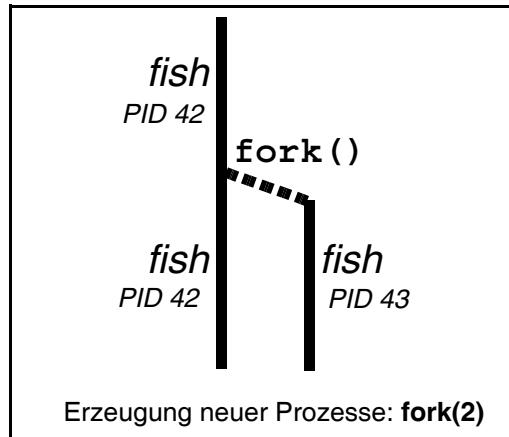
U7-1 UNIX-Prozesshierarchie

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



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

U7-2 POSIX Prozess-Systemfunktionen



1 fork(2): Erzeugung eines neuen Prozesses

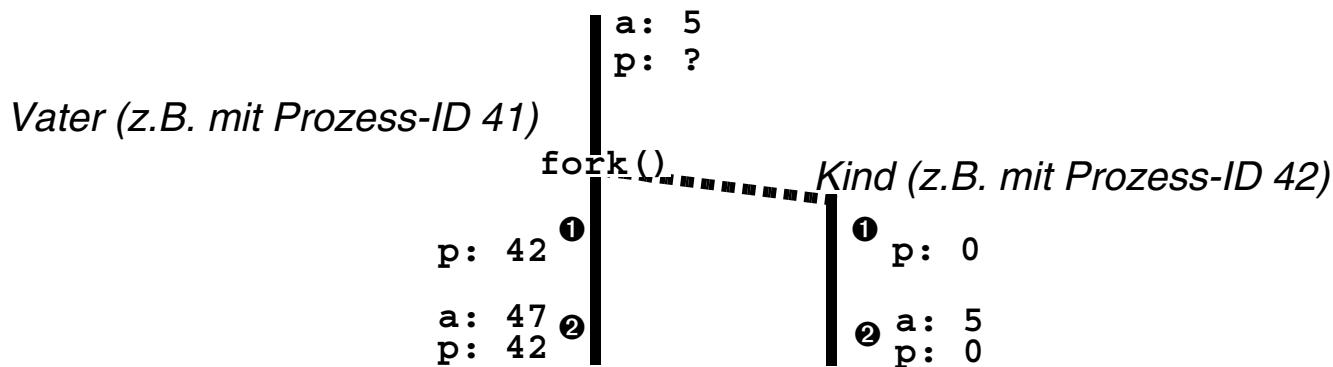
- Erzeugt einen neuen Kindprozess
- Exakte Kopie des Vaters...
 - ◆ Datensegment (neue Kopie, gleiche Daten)
 - ◆ Stacksegment (neue Kopie, gleiche Daten)
 - ◆ Textsegment (gemeinsam genutzt, da nur lesbar)
 - ◆ Filedeskriptoren (geöffnete Dateien)
 - ◆ ...
- ...mit Ausnahme der Prozess-ID
- Kind startet 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(); ①
a += p; ②
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
 - ```
exec1("/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;
 pid = fork();
 if (pid > 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 */
 }
}
```

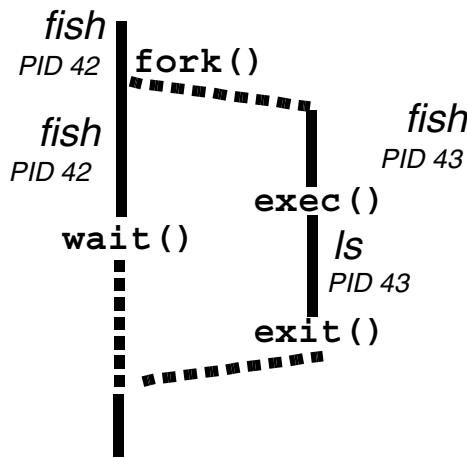
## 4 wait(2)

- `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`

# U7-3 Aufgabe 7: 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

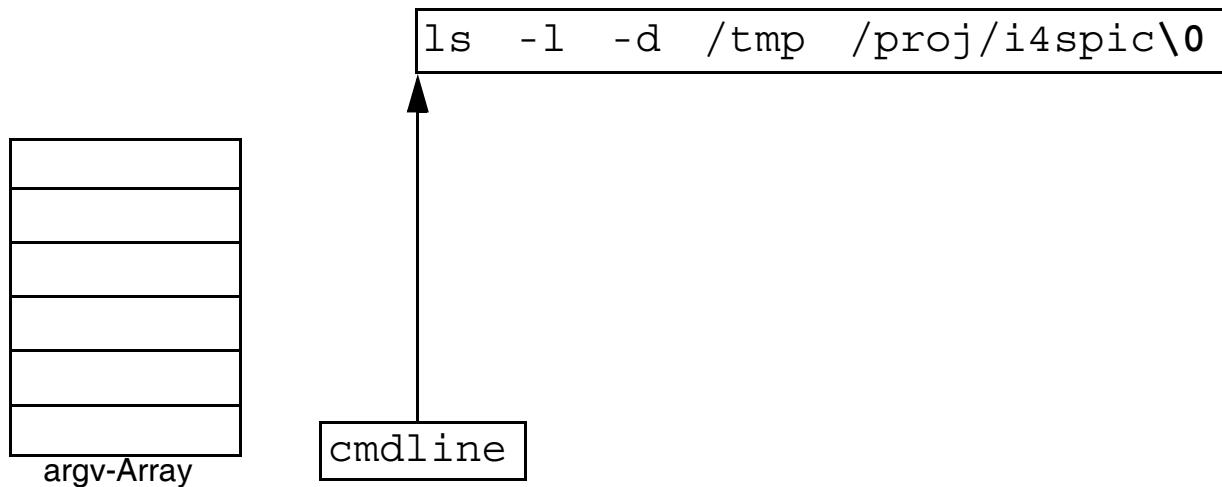
## 2 strtok

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

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

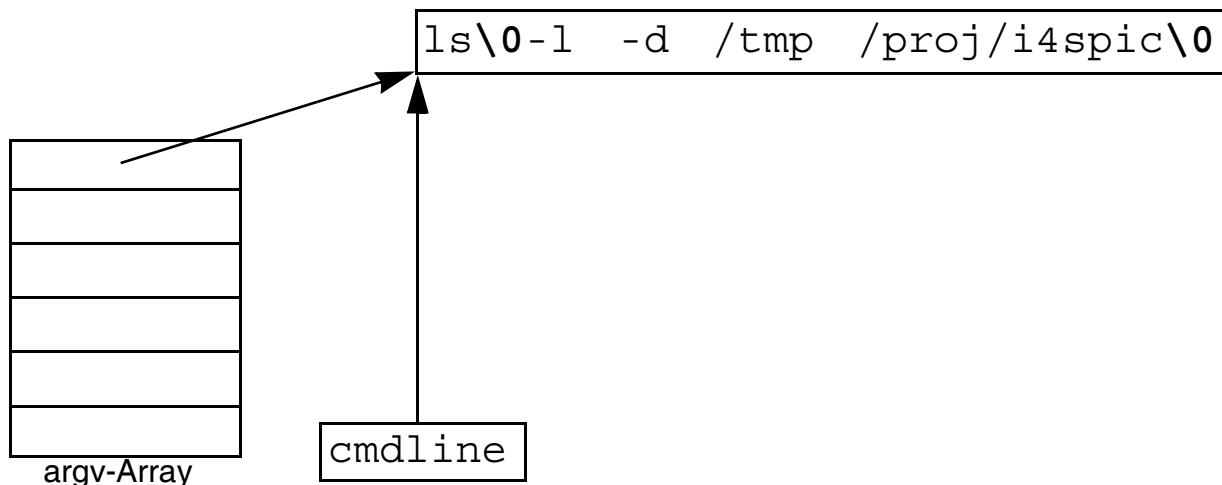
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token (mehrere aufeinanderfolgende Trennzeichen werden hierbei übersprungen)
  - ◆ str ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen NULL
  - ◆ delim ist ein String, der alle Trennzeichen 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

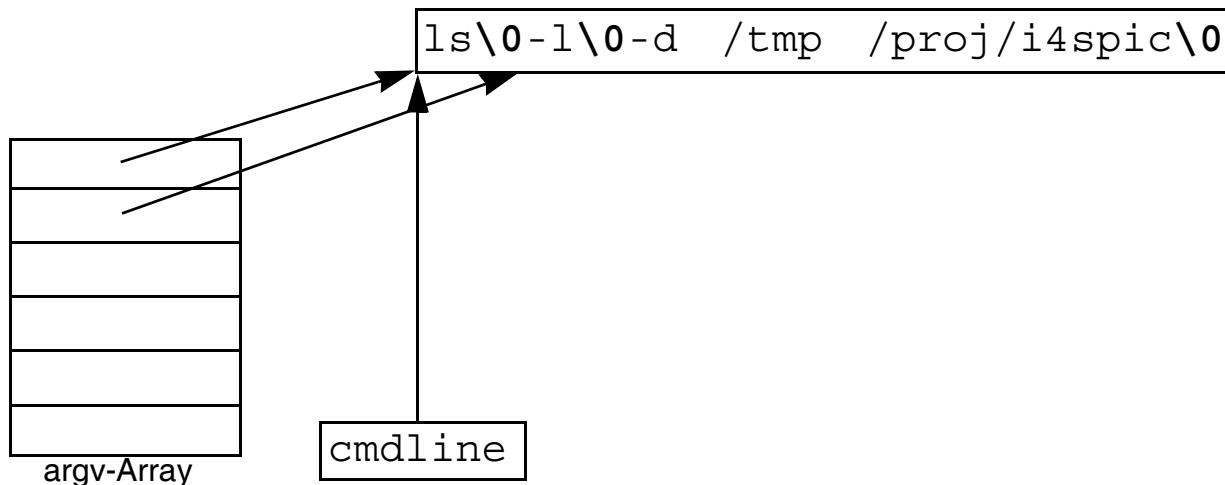


- Erster **strtok**-Aufruf mit dem Zeiger auf diesen Speicherbereich

```
my_argv[0] = strtok(cmdline, " \t\n");
```

- **strtok** liefert Zeiger auf erstes Token /s und ersetzt den Folgetrenner mit '\0'

## 2 strtok-Beispiel

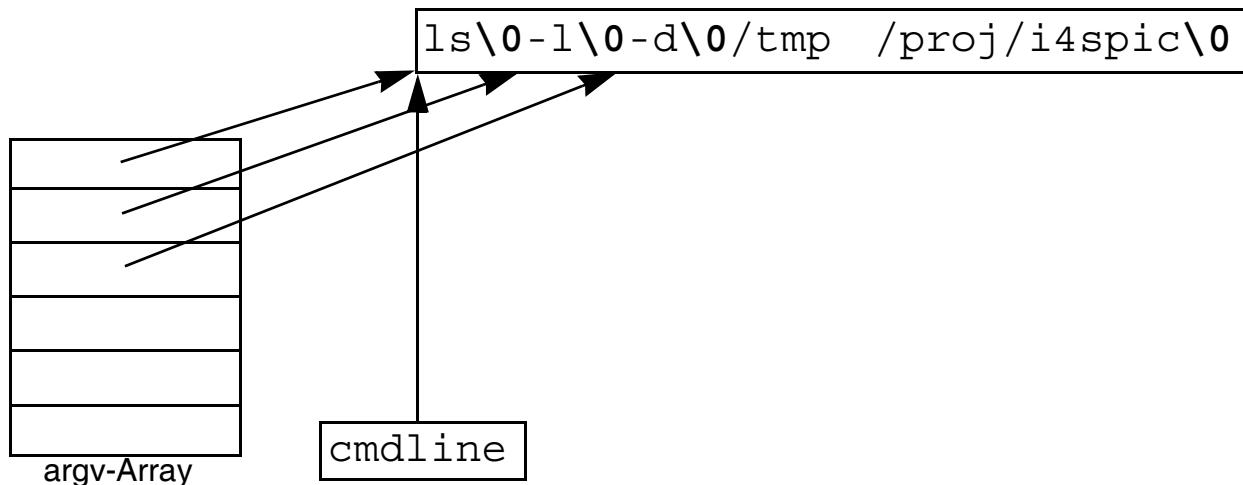


- Weitere Aufrufe von `strtok` nun mit einem NULL-Zeiger

```
while(my_argv[i] != NULL) {
 i++;
 my_argv[i] = strtok(NULL, " \t\n");
}
```

- `strtok` liefert jeweils Zeiger auf das nächste Token

## 2 strtok-Beispiel

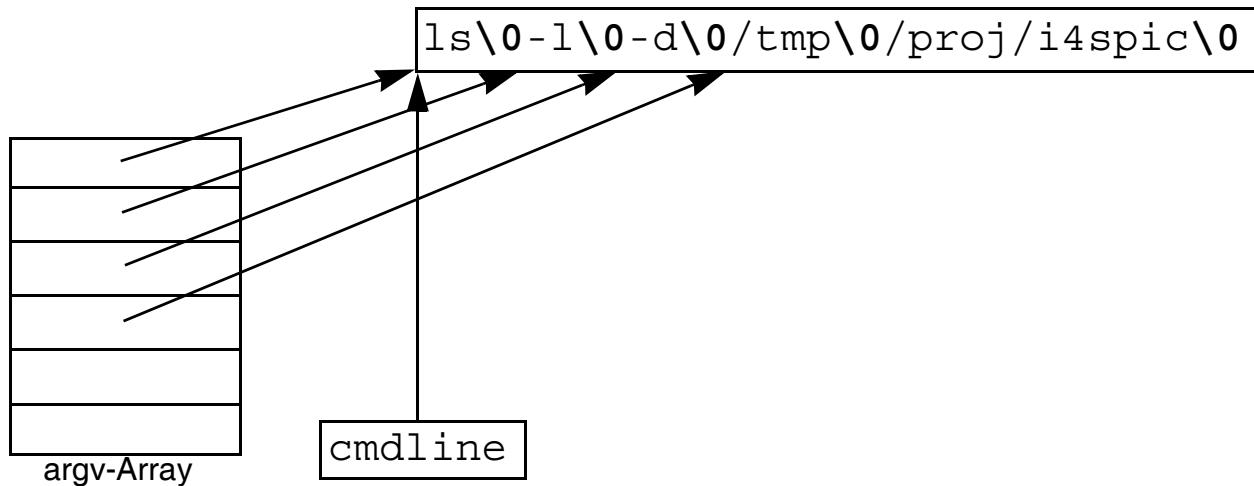


- Weitere Aufrufe von `strtok` nun mit einem NULL-Zeiger

```
while(my_argv[i] != NULL) {
 i++;
 my_argv[i] = strtok(NULL, " \t\n");
}
```

- `strtok` liefert jeweils Zeiger auf das nächste Token

## 2 strtok-Beispiel

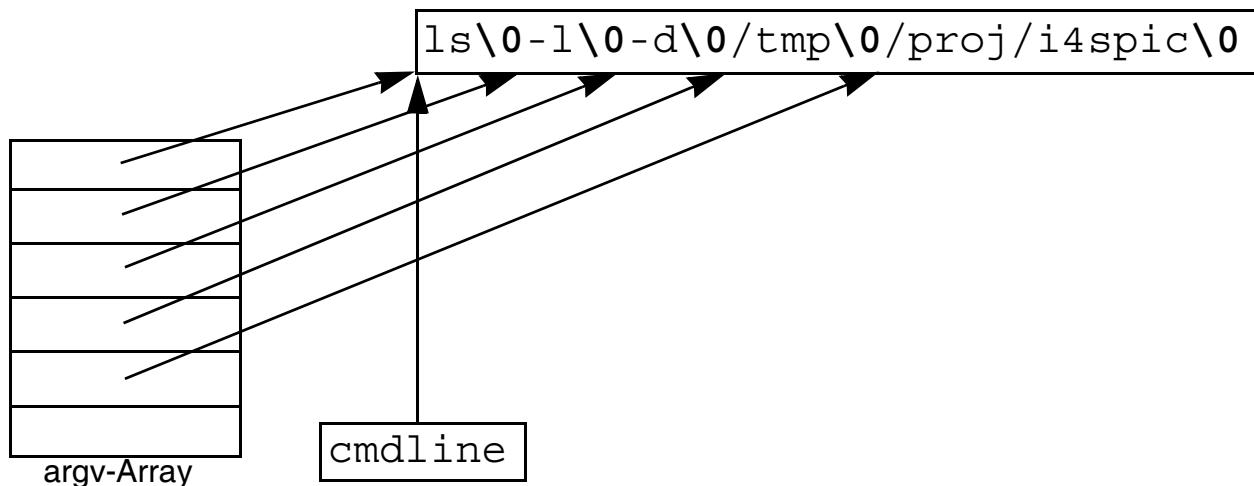


- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger

```
while(my_argv[i] != NULL) {
 i++;
 my_argv[i] = strtok(NULL, " \t\n");
}
```

- strtok** liefert jeweils Zeiger auf das nächste Token

## 2 strtok-Beispiel

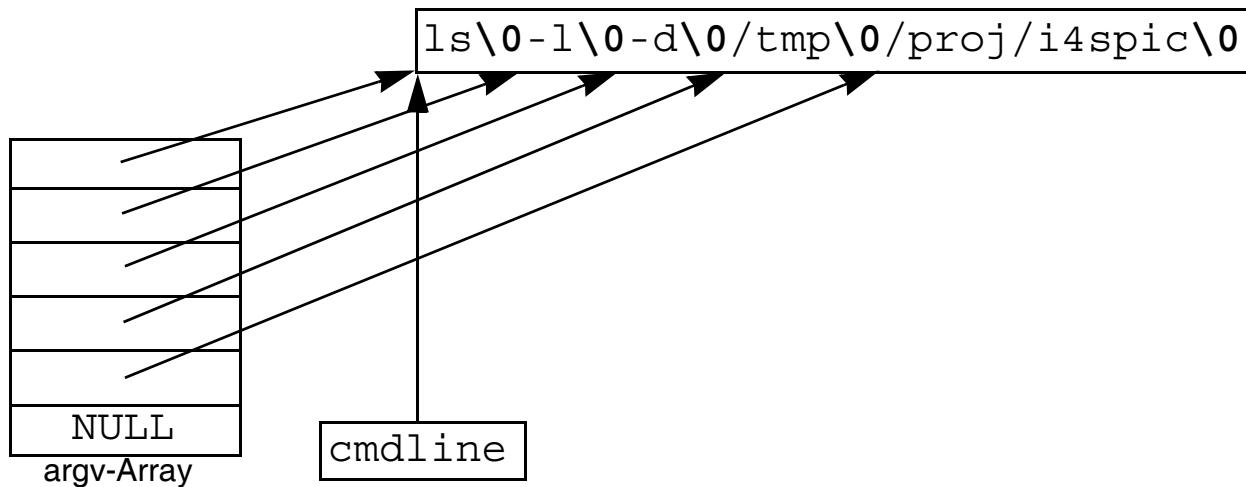


- Weitere Aufrufe von **strtok** nun mit einem NULL-Zeiger

```
while(my_argv[i] != NULL) {
 i++;
 my_argv[i] = strtok(NULL, " \t\n");
}
```

- **strtok** liefert jeweils Zeiger auf das nächste Token

## 2 strtok-Beispiel



- Weitere Aufrufe von `strtok` nun mit einem `NULL`-Zeiger

```
while(my_argv[i] != NULL) {
 i++;
 my_argv[i] = strtok(NULL, " \t\n");
}
```

- Am Ende liefert `strtok` `NULL` und das `argv-Array` hat die nötige Form